Camil Demetrescu
Irene Finocchi
Giuseppe F. Italiano

Algoritmi e strutture dati

McGraw-Hill

Milano • New York • San Francisco • Washington D.C. • Auckland
Bogotá • Lisboa • London • Madrid • Mexico City • Montreal
New Delhi • San Juan • Singapore • Sydney • Tokyo • Toronto

Copyright © 2004 The McGraw-Hill Companies, S.r.l.
Publishing Group Italia
via Ripamonti 89 – 20139 Milano

McGraw-Hill
A Division of The McGraw-Hill Companies

I diritti di traduzione, di riproduzione, di memorizzazione elettronica e di adattamento totale e parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Editor: Chiara Tartara
Produzione: Donatella Giuliani
Impaginazione: a cura degli Autori
Grafica di copertina: G & G
Stampa: Cromografica Europea, Rho (MI)

ISBN 88-386-6161-8
Printed in Italy
23456789CROERR987654

A teacher can never truly teach unless he is still learning himself. A lamp can never light another lamp unless it continues to burn its own flame. The teacher who has come to the end of his subject, who has no living traffic with his knowledge but merely repeats his lesson to his students, can only load their minds, he cannot quicken them.

(Rabindranath Tagore, premio Nobel per la letteratura, 1913)

Indice

Prefazione xv

1 Un'introduzione informale agli algoritmi 1
1.1 I numeri di Fibonacci . . . 2
1.2 Un algoritmo numerico . . . 4
1.3 Un algoritmo ricorsivo . . . 6
1.4 Un algoritmo iterativo . . . 9
1.5 Occupazione di memoria . . . 10
1.6 Notazione asintotica . . . 12
1.7 Un algoritmo basato su potenze ricorsive . . . 13
1.8 Problemi . . . 18
1.9 Sommario . . . 19
1.10 Note bibliografiche . . . 20

2 Modelli di calcolo e metodologie di analisi 23
2.1 Modelli di calcolo . . . 23
2.1.1 Criteri di costo uniforme e logaritmico . . . 24
2.2 La notazione asintotica O, Ω, Θ . . . 25
2.3 Delimitazioni inferiori e superiori . . . 27
2.4 Metodi di analisi . . . 29
2.4.1 Caso peggiore, caso migliore e caso medio . . . 30
2.4.2 Analisi della ricerca sequenziale . . . 31
2.4.3 Un algoritmo più veloce: la ricerca binaria . . . 34
2.5 Analisi di algoritmi ricorsivi . . . 36
2.5.1 Metodo dell'iterazione . . . 37
2.5.2 Metodo della sostituzione . . . 38
2.5.3 Il teorema fondamentale delle ricorrenze . . . 40
2.5.4 Altre tecniche utili . . . 44
2.6 Analisi di algoritmi randomizzati . . . 45
2.7 * Analisi ammortizzata . . . 47
2.7.1 Il metodo dei crediti . . . 50
2.7.2 Il metodo del potenziale . . . 51
2.8 * Modelli evoluti: la gerarchia di memoria . . . 53
2.9 Problemi . . . 55
2.10 Sommario . . . 57
2.11 Note bibliografiche . . . 58

3 Strutture dati elementari 61
3.1 Tecniche per rappresentare collezioni di oggetti . . . 62
3.1.1 Strutture indicizzate: array . . . 62
3.1.2 Strutture collegate: record e puntatori . . . 66
3.2 Pile e code . . . 68
3.3 Alberi . . . 69
3.3.1 Rappresentazioni indicizzate . . . 71
3.3.2 Rappresentazioni collegate . . . 73
3.3.3 Visite di alberi . . . 74
3.4 Problemi . . . 77
3.5 Sommario . . . 78
3.6 Note bibliografiche . . . 79

4 Ordinamento 81
4.1 Una delimitazione inferiore al numero di confronti . . . 82
4.1.1 Alberi di decisione . . . 83
4.1.2 La delimitazione inferiore nel caso peggiore . . . 84
4.2 Ordinare in tempo quadratico . . . 85
4.2.1 Ordinamenti incrementali . . . 86
4.2.2 Ordinamento a bolle . . . 88
4.3 Heapsort . . . 90
4.3.1 Struttura dati heap . . . 90
4.3.2 Ordinare *in loco* mediante heap . . . 94
4.4 Mergesort . . . 95
4.5 Quicksort . . . 98
4.5.1 Analisi di Quicksort . . . 99
4.6 Bucketsort . . . 102
4.7 Radixsort . . . 106
4.8 Problemi . . . 108
4.9 Sommario . . . 111
4.10 Note bibliografiche . . . 111

5 Selezione e statistiche di ordine 115
5.1 Selezione per piccoli valori di k . . . 116
5.1.1 Ricerca del secondo minimo . . . 116
5.1.2 L'algoritmo Heapselect . . . 119
5.2 Calcolo randomizzato del mediano . . . 119
5.3 Calcolo deterministico del mediano . . . 124
5.3.1 Mediano dei mediani . . . 125
5.4 Problemi . . . 129
5.5 Sommario . . . 130
5.6 Note bibliografiche . . . 131

6 Alberi di ricerca 133
6.1 Alberi binari di ricerca . . . 134
6.2 Alberi AVL . . . 139
6.2.1 Altezza di un albero AVL . . . 140
6.2.2 Ribilanciamento tramite rotazioni . . . 142
6.2.3 Modifiche del dizionario . . . 143
6.3 * Alberi auto-aggiustanti . . . 146
6.3.1 L'operazione splay . . . 146
6.3.2 Analisi basata sul potenziale . . . 147
6.4 Alberi 2-3 . . . 152
6.4.1 Fusioni e separazioni di nodi . . . 154
6.5 B-alberi . . . 156
6.5.1 Definizioni e proprietà . . . 157
6.5.2 Inserimenti e cancellazioni di chiavi . . . 160
6.6 Alberi 2-3-4 ed alberi red-black . . . 162
6.7 Problemi . . . 165
6.8 Sommario . . . 166
6.9 Note bibliografiche . . . 167

7 Tavole hash 169
7.1 Tavole ad accesso diretto . . . 169
7.2 Tavole hash . . . 170
7.2.1 Definizione di funzioni hash . . . 173
7.3 Risoluzione delle collisioni . . . 175
7.3.1 Liste di collisione . . . 175
7.3.2 Indirizzamento aperto . . . 177
7.4 Problemi . . . 183
7.5 Sommario . . . 184
7.6 Note bibliografiche . . . 185

8 Code con priorità 187
8.1 d-heap . . . 188
8.2 Heap binomiali . . . 192
8.3 * Heap di Fibonacci . . . 196
8.3.1 Heap binomiali rilassati . . . 197
8.3.2 Heap di Fibonacci . . . 201
8.4 Problemi . . . 207
8.5 Sommario . . . 207
8.6 Note bibliografiche . . . 209

9 Union-find 211
9.1 Approcci elementari al problema union-find . . . 212
9.1.1 Algoritmi di tipo QuickFind . . . 213
9.1.2 Algoritmi di tipo QuickUnion . . . 215
9.2 Euristiche di bilanciamento nell'operazione union . . . 217
9.2.1 Bilanciamento per algoritmi di tipo QuickFind . . . 217

9.2.2 Bilanciamento per algoritmi di tipo QuickUnion 220
9.3 Euristiche di compressione nell'operazione find 225
9.4 * Union-find con bilanciamento e compressione 226
9.4.1 Proprietà del rank 227
9.4.2 Un'analisi preliminare 227
9.4.3 Un'analisi più raffinata 228
9.5 Problemi 233
9.6 Sommario 234
9.7 Note bibliografiche 235

10 Tecniche algoritmiche **237**
10.1 Tecnica *divide et impera* 238
10.1.1 Moltiplicazione di interi di grandezza arbitraria 239
10.1.2 Moltiplicazione tra matrici 241
10.2 Programmazione dinamica 242
10.2.1 La distanza tra due stringhe di caratteri 244
10.2.2 Associatività del prodotto tra matrici 249
10.3 Tecnica golosa (o *greedy*) 253
10.3.1 Il distributore automatico di resto 254
10.3.2 Problemi di sequenziamento 256
10.4 Problemi 258
10.5 Sommario 259
10.6 Note bibliografiche 260

11 Grafi e visite di grafi **263**
11.1 Definizioni preliminari sui grafi 265
11.2 Strutture dati per rappresentare grafi 268
11.3 Visite di grafi 274
11.3.1 Visita in ampiezza 277
11.3.2 Visita in profondità 282
11.3.3 Visite in ampiezza ed in profondità su grafi orientati . . . 285
11.4 Componenti connesse di un grafo non orientato 288
11.5 Componenti fortemente connesse di un grafo orientato 289
11.6 Problemi 296
11.7 Sommario 298
11.8 Note bibliografiche 299

12 Minimo albero ricoprente **301**
12.1 Proprietà dei minimi alberi ricoprenti 302
12.2 Algoritmo di Kruskal 305
12.2.1 Implementazione mediante union-find 307
12.3 Algoritmo di Prim 308
12.3.1 Implementazione mediante code con priorità 311
12.4 Algoritmo di Borůvka 313
12.5 Problemi 314

12.6 Sommario 316
12.7 Note bibliografiche 317

13 Cammini minimi **319**
13.1 Cammini minimi e distanze in un grafo 319
13.1.1 Cammini minimi in grafi con cicli 321
13.1.2 Distanza fra vertici in un grafo 322
13.1.3 Costruire cammini minimi a partire da distanze 323
13.1.4 Alberi di cammini minimi 323
13.1.5 Varianti del problema dei cammini minimi 324
13.2 La tecnica del rilassamento 325
13.3 Algoritmo di Bellman e Ford 326
13.4 Algoritmo per grafi diretti aciclici 327
13.4.1 Ordinamento topologico 327
13.4.2 Rilassamento in ordine topologico 329
13.5 Algoritmo di Dijkstra 330
13.5.1 Implementazione mediante code con priorità 332
13.6 Algoritmo di Floyd e Warshall 336
13.7 Problemi 338
13.8 Sommario 339
13.9 Note bibliografiche 341

14 Flusso **347**
14.1 Reti di flusso 347
14.2 Metodo delle reti residue 350
14.2.1 Versione ricorsiva del metodo delle reti residue 353
14.3 Metodo dei cammini aumentanti 353
14.4 Algoritmo di Ford e Fulkerson 355
14.5 Algoritmo di Edmonds e Karp 356
14.6 Flusso massimo e taglio minimo 358
14.7 Problemi 360
14.8 Sommario 361
14.9 Note bibliografiche 362

15 Algoritmi geometrici **365**
15.1 Inviluppo convesso 366
15.1.1 Un approccio "induttivo" 366
15.1.2 Il metodo dell'incartamento dei regali 369
15.1.3 La scansione di Graham 370
15.2 Localizzazione di punti in suddivisioni planari 372
15.2.1 La sequenza di triangolazioni 374
15.2.2 La struttura dati 375
15.2.3 L'algoritmo di interrogazione 376
15.3 Problemi di ricerca multidimensionali 377
15.3.1 Il caso di una dimensione 378
15.3.2 Il range tree 380

15.3.3 Il priority search tree 382
15.4 Intersezione di rettangoli 386
15.4.1 Il metodo di plane sweep 387
15.4.2 L'albero degli intervalli 387
15.5 Problemi 387
15.6 Sommario 391
15.7 Note bibliografiche 393
. 394

16 Teoria della NP-completezza 397
16.1 Complessità di problemi decisionali 398
16.1.1 Decidere, ricercare, ottimizzare 398
16.1.2 Classi di complessità 399
16.2 La classe NP 402
16.2.1 Non determinismo 402
16.2.2 Uno sguardo alla gerarchia 404
16.3 Problemi NP-completi 405
16.3.1 Riducibilità polinomiale 406
16.3.2 Il teorema di Cook 407
16.3.3 Dimostrazioni di NP-completezza 409
16.4 Algoritmi di approssimazione 413
16.5 Problemi 415
16.6 Sommario 416
16.7 Note bibliografiche 417

17 Appendice 419
17.1 Logaritmi e numero di Nepero 419
17.2 Serie e successioni 421
17.2.1 Serie aritmetica e geometrica 421
17.2.2 Calcolo di somme per integrazione 423
17.2.3 La successione di Fibonacci 425
17.3 Elementi di calcolo della probabilità 426
17.4 Elementi di calcolo combinatorio 428
17.5 Elementi di teoria dei grafi 430
17.5.1 Alberi 430
17.5.2 Grafi planari 432
17.6 Note bibliografiche 434

Prefazione

Questo libro offre un'introduzione allo studio degli algoritmi e delle strutture dati, cercando di conciliare comprensibilità, chiarezza di esposizione e rigore matematico. Particolare enfasi è posta sull'astrazione delle tecniche e delle metodologie generali di progetto e analisi di algoritmi, stimolandone la comprensione intuitiva dei principi fondamentali.

Il testo, pur essendo indipendente dalla scelta di un particolare linguaggio di programmazione, adotta un approccio orientato agli oggetti sia nella descrizione delle strutture dati che nello pseudocodice utilizzato per descrivere gli algoritmi. In tal modo, pur astraendo dai dettagli implementativi di basso livello, gli algoritmi presentati non risultano troppo distanti da una loro reale implementazione: ci si può aspettare di tradurre ogni istruzione dello pseudocodice in un numero limitato di linee di codice di un linguaggio di programmazione, come C++ o Java, senza troppe difficoltà ed in modo quasi meccanico. Tutti gli algoritmi del libro, anche i più sofisticati, vengono descritti in pseudocodice, in modo da incoraggiare il lettore a realizzarne e a studiarne delle implementazioni pratiche.

Il libro è concepito soprattutto per corsi universitari delle Facoltà di Ingegneria e di Scienze Matematiche, Fisiche e Naturali, e nasce dall'esperienza diretta degli autori, maturata negli ultimi anni, nell'insegnare corsi di algoritmi e strutture dati in entrambe le Facoltà.

Per il docente

Lo svolgimento degli argomenti esposti nel testo non avviene necessariamente in un unico corso, ma può avvenire in vari momenti dei percorsi formativi interni ai corsi di laurea in Informatica ed in Ingegneria Informatica, nell'ambito sia della laurea triennale che della laurea specialistica.

L'organizzazione del volume consente infatti percorsi di lettura diversificati, corrispondenti a diversi moduli in cui il relativo materiale, o parte di esso, può essere proposto agli studenti. Ad esempio, i Capitoli 1-11 possono essere di riferimento per un modulo di base su Algoritmi e Strutture Dati, così come viene tipicamente svolto al primo o al secondo anno di un corso di laurea. Materiale per un modulo più avanzato di algoritmica, per la laurea triennale o per la laurea specialistica, è invece contenuto nei Capitoli 8-16. Per evidenziare il materiale più avanzato all'interno di un capitolo, verrà utilizzato un asterisco (*).

Non dovrebbe essere quindi difficile organizzare il proprio percorso formativo scegliendo opportunamente il materiale dai capitoli necessari. A titolo di

esempio, negli ultimi anni gli autori hanno utilizzato materiale incluso in questo volume per varie tipologie di corsi di algoritmi e strutture dati che hanno avuto il piacere di insegnare presso diverse università italiane e straniere, sia in modalità tradizionale che per *e-learning*:

- Un corso di Algoritmi e Strutture Dati di 50 ore di lezione, tenuto al secondo anno del corso di laurea in Ingegneria Informatica dell'Università di Roma "Tor Vergata".
- Un corso di Algoritmi e Strutture Dati erogato in modalità interamente on-line (tramite piattaforma di *e-learning*), tenuto al secondo anno del corso di laurea in Ingegneria Informatica on-line dell'Università di Roma "Tor Vergata".
- Un corso di *Design and Analysis of Algorithms* a livello *undergraduate* (quarto anno) della Columbia University, della durata di 40 ore di lezione.
- Un corso di *Design and Analysis of Algorithms* a livello *graduate* (*Master*) erogato in modalità *e-learning* dalla *Columbia University Video Network*.
- Un corso di *Design and Analysis of Algorithms* a livello *graduate* (*Master*) erogato in modalità *e-learning* dalla *National Technical University*.
- Un corso di *Graph Algorithms* di 40 ore di lezione a livello *graduate* (*Ph.D.*) tenuto presso l'*Hong Kong University of Science & Technology*.

Per lo studente

Ci auguriamo che questo libro possa rappresentare per te una piacevole introduzione agli algoritmi ed alle strutture dati. Probabilmente il tuo corso affronterà soltanto una parte del materiale contenuto in questo libro. È anche probabile che affronterai il resto del materiale in un corso avanzato, oppure nella tua futura carriera professionale. Gli unici prerequisiti per leggere e comprendere il materiale di questo libro sono:

- Conoscenza di un linguaggio di programmazione, come ad esempio il `C`, `C++` o `Java`, acquisita in corsi come "Fondamenti di Informatica", "Programmazione", o "Laboratorio di Informatica".
- Conoscenza di nozioni di base relative alla ricorsione e alle strutture dati elementari, come ad esempio liste ed array.
- Familiarità con dimostrazioni e concetti matematici di base, come ad esempio le dimostrazioni per induzione, acquisiti in corsi di "Analisi Matematica".

Ogni capitolo del libro si conclude con un sommario, che tenta di riepilogare i concetti più importanti introdotti nel capitolo, e con un insieme di esercizi e problemi, che sono utili nel verificare il grado di comprensione del materiale approfondito nel capitolo. Per evidenziare i problemi più difficili, saranno utilizzati uno o più asterischi (*, **).

Brevi cenni storico-bibliografici, al termine di ogni capitolo, consentono anche al lettore interessato di approfondire gli aspetti culturali dell'algoritmica, di porre in relazione il materiale del libro con argomenti propri di altre discipline, e di comprendere il ruolo dei diversi concetti nell'evoluzione delle discipline algoritmiche.

Supporto sul sito Web

Il Sito Web del libro di testo, disponibile all'indirizzo

`http://www.ateneonline.it/demetrescu`

contiene varie informazioni utili per il docente e per lo studente.

Per il docente

Nel Sito Web del docente troverai:

- Copia delle trasparenze utilizzate dagli autori del testo in un corso della laurea triennale.
- Soluzione completa di un numero selezionato di problemi nel testo.

Per lo studente

Nel Sito Web dello studente troverai:

- Esercizi di autovalutazione organizzati in base ai capitoli del libro.
- Animuzioni dei principali algoritmi descritti nel testo, sviluppate nel sistema Leonardo Web.

Sia lo studente che il docente possono interagire con le animazioni: questo consentirà al docente di utilizzare le animuzione a scopi didattici, ed allo studente di approfondire indipendentemente e più rapidamente il materiale del libro.

Prevediamo che in futuro il Sito Web conterrà molto altro materiale, progettato anche per corsi in modalità *e-learning*: ad esempio, prevediamo di mettere a disposizione degli studenti informuzioni su come acquisire e correggere (automaticamente) alcuni esercizi di autovalutazione relativi al testo, così da poter verificare in maniera autonoma il proprio livello di apprendimento del testo.

Ringraziamenti

Molti amici e colleghi hanno contribuito enormemente alla realizzazione e alla qualità di questo libro. In particolare, desideriamo ringraziare Alberto Apostolico, Giorgio Ausiello, Gianfranco Bilardi, Andrea Clementi, Fabrizio d'Amore, Paolo G. Franciosa, Giorgio Gambosi, Raffaele Giancarlo, Fabrizio Grandoni, Stefano Leonardi, Giuseppe Liotta, Fabrizio Luccio, Angelo Monti, Alberto Marchetti Spaccamela, Alessandro Panconesi, Rossella Petreschi, Riccardo Silvestri, ed Ugo Vaccaro per i loro commenti e per discussioni su argomenti relativi a questo libro. Un ringraziamento speciale a Maurizio Lenzerini, per le sue lunghe e pazienti attese di fronte alla stampante del Dipartimento in qualche fine settimana.

Ringraziamo inoltre Vincenzo Bonifaci, Benedetto A. Colombo e Luigi Laura per il loro prezioso supporto nella realizzazione del sistema di visualizzazione

Leonardo Web con cui sono state realizzate le animazioni a corredo di questo testo.

Molti dei nostri studenti hanno inoltre avuto un ruolo fondamentale, aiutandoci soprattutto a correggere alcuni errori nei nostri appunti delle lezioni e a migliorare l'esposizione dei contenuti.

Ringraziamo infine le Università di Roma "La Sapienza" e di Roma "Tor Vergata" per l'ambiente stimolante che ci hanno fornito. Ringraziamo inoltre Columbia University e Hong Kong University of Science and Technology, che ci hanno ospitato generosamente, e in cui abbiamo sviluppato parte del materiale in questo libro.

È stato un vero piacere lavorare con McGraw-Hill. Ringraziamo in particolare Chiara Tartara, Rossana Cecchi, Alessandra Porcelli e Daniela Cipollone per il loro continuo incoraggiamento e supporto.

Infine desideriamo ringraziare le nostre famiglie, che ci hanno supportato con amore, affetto e infinita pazienza durante la scrittura di questo libro. Questo libro è dedicato soprattutto a loro, con amore e riconoscenza.

Avremmo voluto scrivere un libro perfetto. Ma siamo autori imperfetti, e quindi perfettamente consapevoli che questo libro conterrà errori ed imprecisioni, di cui siamo ovviamente gli unici responsabili. Saremo grati a chiunque vorrà segnalarceli per posta elettronica all'indirizzo

`algoritmi@algonet.uniroma2.it`

Roma, Aprile 2004

Camil Demetrescu
Irene Finocchi
Giuseppe F. Italiano

1

Un'introduzione informale agli algoritmi

Quelli che s'innamorano di pratica, sanza scienza, son come 'l nocchiere, ch'entra in navilio sanza timone o bussola, che mai ha certezza dove si vada.

(Leonardo Da Vinci)

Intuitivamente, un algoritmo è un insieme di istruzioni, definite passo per passo, in modo tale da poter essere eseguite meccanicamente, e tali da produrre un determinato risultato. Sempre in modo informale potremmo definire un algoritmo come una sequenza di passi di calcolo che, ricevendo in *ingresso* un valore (od un insieme di valori), restituisce in *uscita* un altro valore (od un insieme di valori), trasformando quindi i dati in ingresso in dati in uscita. Gli algoritmi non sono ristretti soltanto alle discipline informatiche o matematiche, e probabilmente sono esistiti ancor prima che si coniasse un termine speciale per indicarli. Tutti noi utilizziamo, più o meno consapevolmente, algoritmi nella nostra vita quotidiana. Ad esempio, l'algoritmo illustrato nella Figura 1.1 mostra come preparare del caffè a partire da una caffettiera, e da un'opportuna quantità di acqua e di polvere di caffè. Notiamo che l'algoritmo di Figura 1.1 è definito mediante una sequenza *finita* di

algoritmo `preparaCaffè`
1. Svita la caffettiera.
2. Riempi d'acqua il serbatoio della caffettiera.
3. Inserisci il filtro.
4. Riempi il filtro con la polvere di caffè.
5. Avvita la parte superiore della caffettiera.
6. Metti la caffettiera, così predisposta, su un fornello acceso.
7. Spegni il fornello quando il caffè è pronto.
8. Versa il caffè nella tazzina.

Figura 1.1 Algoritmo per preparare il caffè.

passi elementari, descritti in un linguaggio naturale, in modo tale da poter essere interpretati da un essere umano. Molti degli algoritmi che vedremo in questo libro sono concepiti per essere eseguiti sulla CPU di un sistema di elaborazione,

e quindi saranno descritti in *pseudocodice*, che ricorda linguaggi di programmazione come C, C++, o Java, ma contiene alcune frasi in italiano (o in inglese) piuttosto che direttamente in un linguaggio di programmazione.

In questo libro ci occuperemo del progetto di algoritmi per vari tipi di problemi, e della loro analisi matematica, analisi che condurremo anche indipendentemente da eventuali esperimenti sulle implementazioni degli algoritmi. Lo scopo di progettare algoritmi dovrebbe essere evidente: abbiamo bisogno di algoritmi ogni qualvolta dobbiamo scrivere un programma. La necessità di analizzare algoritmi sembrerebbe a prima vista meno ovvia, anche se alcune motivazioni possono facilmente rendere conto della sua importanza.

Innanzitutto, cosa ci può offrire l'analisi di un algoritmo rispetto alla valutazione sperimentale delle prestazioni di un programma? L'analisi sembrerebbe essere, almeno in linea di principio, più affidabile: nella sperimentazione, possiamo solo studiare il comportamento di un programma in un numero limitato di casi, mentre al contrario l'analisi può offrirci delle precise garanzie matematiche sulle prestazioni di un algoritmo per ogni possibile distribuzione dei dati di ingresso. Inoltre, l'analisi ci può aiutare a scegliere tra diverse soluzioni allo stesso problema. Un'analisi attenta può esserci utile nel decidere quale soluzione possa essere più adeguata ai nostri obiettivi, senza costringerci a costose implementazioni ed ai relativi test dei programmi prima di accorgerci che abbiamo scelto di percorrere una strada sbagliata. Possiamo anche predire le prestazioni di un programma software, prima ancora di scriverne le prime linee di codice. In grossi progetti software, è di vitale importanza avere una stima delle prestazioni già a livello progettuale: attendere che tutto il codice sia scritto e scoprire soltanto allora che qualcosa non raggiunge i requisiti prestazionali, potrebbe portare a conseguenze disastrose o per lo meno molto costose. Quando analizziamo un algoritmo, abbiamo la possibilità di scoprire eventuali problemi di prestazioni già in una prima fase di analisi preliminare, e possiamo quindi tentare di eliminarli direttamente in quella fase. Infine, durante l'analisi di un algoritmo, riusciamo ad avere un'idea migliore di quali siano le sue componenti più lente e di quali siano le sue componenti più veloci, e di conseguenza possiamo metterci al lavoro sulle componenti più lente già in questa fase per rendere tutta l'implementazione globalmente più veloce.

1.1 I numeri di Fibonacci

In questo paragrafo introdurremo alcune delle problematiche relative al progetto e all'analisi di algoritmi tramite un problema "giocattolo": il calcolo dei numeri di Fibonacci. Probabilmente non avremo mai bisogno di risolvere questo problema nella nostra vita professionale, ma lo consideriamo qui perché ha una buona valenza didattica, è abbastanza semplice da comprendere e, sorprendeatemente, può essere risolto con molti approcci distinti.

L'isola dei conigli. Leonardo di Pisa (noto anche come Fibonacci) è an noto matematico italiano, vissuto intorno al 1200. Fibonacci si interessò di molte cose, e

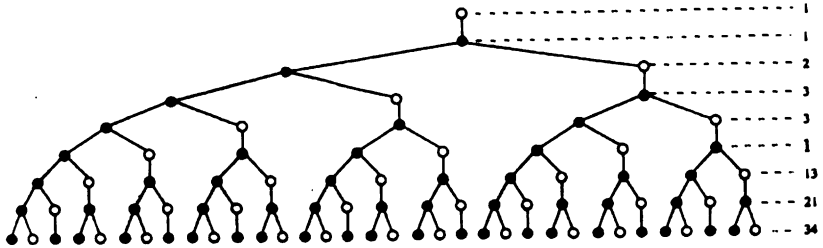


Figura 1.2 I conigli di Fibonacci. Ogni livello dell'albero rappresenta un anno. I nodi neri e bianchi rappresentano rispettivamente coppie di conigli adulti e giovani.

tra le altre studiò anche vari problemi che al giorno d'oggi definiremmo di "dinamica delle popolazioni". Ad esempio, studiò come si espande una popolazione di conigli, sotto opportune condizioni. Immaginiamo di fare il seguente esperimento: partiamo da una singola coppia di conigli in un'isola deserta, e proviamo a studiarne l'evoluzione nel corso degli anni. Come avviene spesso in matematica – e l'analisi di algoritmi utilizza strumenti matematici – rendiamo il problema più astratto, per avere un'idea delle sue caratteristiche generali senza rischiare di perderci troppo nei dettagli di basso livello. In particolare, assumiamo che:

(1) Una coppia di conigli genera due coniglietti ogni unao.
(2) I conigli cominciano a riprodursi soltanto al secondo unao dopo la loro nascita.
(3) I conigli sono immortali.

L'ultima ipotesi è sicuramente irrealistica, ma indubbiamente rende il problema più semplice da analizzare. Dopo averlo analizzato nella sua versione semplificata, potremmo tornare indietro e sostituire questa ipotesi con una più realistica, del tipo "I conigli vivono in media dieci anni". Così facendo, ci renderemmo conto che questo non cambierebbe troppo il risultato.

Proviamo quindi ad esprimere il numero di coppie di conigli in funzione del tempo, ovvero in funzione del numero di unai trascorsi dall'inizio dell'esperimento. Sia F_t il numero di coppie di conigli all'unao t:

$F_1 = 1$: Partiamo con una coppia
$F_2 = 1$: Nel secondo anno, la coppia è ancora troppo giovane per riprodursi
$F_3 = 2$: Nel terzo anno, la coppia iniziale dà alla luce un'altra coppia
$F_4 = 3$: Nel quarto anno nasce un'altra coppia di coniglietti
$F_5 = 5$: Nel quinto anno nascono i primi nipoti della coppia iniziale!

Questo fenomeno riproduttivo è illustrato in Figura 1.2. In generale, nell'anno n sono presenti tutti i conigli dell'anno precedente (F_{n-1} coppie) ed inoltre abbiamo una coppia di conigli per ogni coppia presente due anni prima (F_{n-2} coppie). Avremo quindi:

$$\boxed{F_n = F_{n-1} + F_{n-2}}$$

Il problema che vogliamo risolvere è, dato un intero $n \geq 1$, come calcolare il valore di F_n. In pratica, vogliamo trovare la soluzione della seguente *relazione di ricorrenza*:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n = 1, 2 \end{cases}$$

Faremo riferimento a tale relazione come *relazione di Fibonacci*. I primi numeri della sequenza di Fibonacci $\{F_n\}_{n \geq 1}$ sono i seguenti:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F_n	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

Nei paragrafi seguenti affronteremo il problema del calcolo dei numeri di Fibonacci, presentando diverse soluzioni ed evidenziandone pregi e difetti.

1.2 Un algoritmo numerico

Per ottenere un primo algoritmo, cerchiamo una funzione matematica che calcoli direttamente i numeri di Fibonacci. A questo scopo, consideriamo innanzitutto funzioni che soddisfino la relazione di ricorrenza $F_n = F_{n-1} + F_{n-2}$. Proviamo con funzioni esponenziali della forma a^n con $a \neq 0$, e vediamo per quali valori di a esse soddisfano la relazione di Fibonacci. Studiamo quindi l'equazione:

$$a^n = a^{n-1} + a^{n-2}$$

ovvero

$$a^n - a^{n-1} - a^{n-2} = 0$$

che è equivalente a

$$a^{n-2} \cdot (a^2 - a - 1) = 0$$

Per l'ipotesi $a \neq 0$, cerchiamo i valori di a che soddisfano l'equazione:

$$a^2 - a - 1 = 0$$

Questa equazione di secondo grado ammette due radici reali [1]:

$$\begin{aligned} \phi &= \frac{1+\sqrt{1+4}}{2} = \frac{1+\sqrt{5}}{2} \approx +1.618 \\ \hat{\phi} &= \frac{1-\sqrt{1+4}}{2} = \frac{1-\sqrt{5}}{2} \approx -0.618 \end{aligned}$$

[1] Il numero $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ è importante in matematica, ma anche nel mondo dell'arte, dove appare sin dall'antichità come rapporto dimensionale di particolare pregio estetico nei progetti architettonici. Si denota tale grandezza come *sezione aurea*, e si usa la lettera greca ϕ in onore dello scultore Fidia, che si dice ne abbia fatto un particolare uso nelle sue opere.

algoritmo `fibonacci1`(*intero* n) $\rightarrow$ *intero*

1. **return** $\frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$

Figura 1.3 Algoritmo `fibonacci1` per il calcolo dell'n-esimo numero di Fibonacci.

Abbiamo quindi trovato due funzioni ϕ^n e $\hat{\phi}^n$ che soddisfano la relazione di Fibonacci. Purtroppo, però, nessuna delle due calcola i numeri di Fibonacci come vorremmo: infatti, ad esempio, $\phi^2 \neq F_2$. Il punto sembra essere che, nel definire ϕ^n e $\hat{\phi}^n$, non abbiamo considerato i casi base della definizione ricorsiva.

Per risolvere il problema, osserviamo che una qualunque combinazione lineare di funzioni che soddisfano la relazione di Fibonacci soddisfa anch'essa la relazione di Fibonacci. Proviamo quindi a vedere se un'opportuna combinazione lineare $c_1 \cdot \phi^n + c_2 \cdot \hat{\phi}^n$ di ϕ^n e $\hat{\phi}^n$ può darci la funzione F_n cercata. Imponendo che vengano soddisfatti i casi base della definizione ricorsiva $F_1 = 1$ ed $F_2 = 1$, otteniamo il sistema di due equazioni in due incognite c_1 e c_2:

$$\begin{cases} c_1 \cdot \phi + c_2 \cdot \hat{\phi} = 1 \\ c_1 \cdot \phi^2 + c_2 \cdot \hat{\phi}^2 = 1 \end{cases}$$

Risolvendo il sistema otteniamo:

$$c_1 = +\frac{1}{\sqrt{5}} \qquad c_2 = -\frac{1}{\sqrt{5}}$$

La soluzione della relazione di Fibonacci risulta pertanto essere:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$$

Tornando al nostro problema originario, ovvero calcolare l'n-esimo numero di Fibonacci, un primo algoritmo risulta quindi essere quello mostrato in Figura 1.3.

Osservando che i numeri coinvolti nel calcolo sono reali, sorge spontaneo chiedersi: quale deve essere l'accuratezza di ϕ e $\hat{\phi}$ per ottenere la riposta esatta? Ad esempio, usando tre cifre decimali, ovvero $\phi \approx 1.618$ e $\hat{\phi} \approx -0.618$, otteniamo:

`fibonacci1`$(3) = 1.99992$: arrotondando siamo in grado di ottenere il valore corretto $F_3 = 2$.

`fibonacci1`$(16) = 986.698$: arrotondando siamo in grado di ottenere il valore corretto $F_{16} = 987$.

`fibonacci1`$(18) = 2583.1$: arrotondando otteniamo 2583, mentre invece il valore corretto è $F_{18} = 2584$.

```
algoritmo fibonacci2(intero n) → intero
    if ( n ≤ 2 ) then return 1
    else return fibonacci2(n − 1) + fibonacci2(n − 2)
```

Figura 1.4 Algoritmo `fibonacci2` per il calcolo dell'n-esimo numero di Fibonacci.

Il limite dell'algoritmo `fibonacci1`, che utilizza direttamente la soluzione della relazione di Fibonacci, sembra evidente: siamo costretti a lavorare con numeri reali, che sono rappresentati nei calcolatori con una precisione limitata, e corriamo quindi il rischio di fornire risposte non corrette a causa degli errori di arrotondamento. Dato che i numeri di Fibonacci sono interi, non sarebbe più semplice progettare un algoritmo che lavora soltanto su interi? Vedremo vari esempi di algoritmi che lavorano soltanto con numeri interi nelle pagine seguenti.

1.3 Un algoritmo ricorsivo

La relazione di Fibonacci sembra essere naturalmente ricorsiva. Sembrerebbe quindi naturale risolvere il nostro problema con un algoritmo ricorsivo, così come probabilmente abbiamo già visto in un corso introduttivo di programmazione. Lo pseudocodice di questo approccio è mostrato in Figura 1.4. Una tipica domanda che ci porremo frequentemente in questo libro è: "Quanto tempo richiede questo algoritmo?" Per rispondere a questa domanda dobbiamo innanzitutto chiarire come misurare il tempo. La risposta più ovvia sembrerebbe "in secondi", ma sarebbe importante riuscire ad ottenere una risposta che sia indipendente dalle tecnologie, ossia una risposta che non cambia ogni volta che si mette in produzione una nuova CPU. Potremmo quindi pensare di misurare il tempo in termini di istruzioni macchina; dividendo per la velocità della macchina (istruzioni/secondo) otteremmo i tempi esatti. Tuttavia, è estremamente complicato inferire da un pezzo di codice o di pseudocodice il numero esatto di istruzioni che saranno generate da un particolare compilatore. Per avere una prima approssimazione, potremmo quindi decidere di misurare la velocità di un algoritmo in termini di linee di codice eseguite.

Analizziamo quindi quante linee di codice richiede l'algoritmo `fibonacci2`. Notiamo innanzitutto che ogni chiamata alla funzione `fibonacci2` coinvolge una oppure due linee di codice.

- Se $n \leq 2$, viene eseguita solamente una linea di codice, e più precisamente l'istruzione **if** $(n \leq 2)$ **return** 1.
- Se $n = 3$, vengono eseguite 2 linee di codice per la chiamata `fibonacci2(3)`, più una linea di codice per la chiamata `fibonacci2(2)` ed una per la chiamata `fibonacci2(1)`. Questo comporta un totale di 4 linee di codice.

- Se $n = 4$, vengono eseguite 2 linee di codice per la chiamata `fibonacci2(4)`, più 4 linee di codice per la chiamata `fibonacci2(3)` ed una per la chiamata `fibonacci2(2)`. In totale, 7 linee di codice.

Cominciamo ad intuire che le linee di codice della funzione `fibonacci2` si riproducono in modo molto simile ai conigli di Fibonacci! Oltre alle due linee in ogni chiamata, il numero di linee di codice mandate in esecuzione in occasione di una chiamata alla funzione `fibonacci2`(n) è dato dalla somma delle linee di codice mandate in esecuzione dalle due chiamate ricorsive:

$$T(n) = 2 + T(n-1) + T(n-2)$$

In generale, ogni algoritmo ricorsivo come `fibonacci2` può essere analizzato tramite una *relazione di ricorrenza*: il tempo richiesto da una routine è uguale al tempo speso all'interno della routine più il tempo speso per le chiamate ricorsive. Questo produce quasi automaticamente una relazione di ricorrenza, che possiamo risolvere per trovare il tempo di esecuzione. In questo caso particolare, la relazione di ricorrenza è molto simile alla definizione dei numeri di Fibonacci.

Con un po' di lavoro, possiamo risolvere l'equazione almeno in termini dei numeri di Fibonacci F_n. Possiamo rappresentare le chiamate ricorsive con una struttura ad un albero: l'*albero della ricorsione*. In dettaglio, usiamo un nodo, la radice dell'albero, per la prima chiamata, e generiamo un figlio per ogni chiamata ricorsiva. L'albero delle chiamate ricorsive a partire dalla chiamata `fibonacci2(8)` è mostrato in Figura 1.5. Ognuno dei venti nodi interni dell'albero per `fibonacci2(8)` richiede due linee di codice, mentre le ventuno foglie richiedono una linea ciascuna. Il numero totale di linee di codice mandato in esecuzione è quindi $20 \cdot 2 + 21 = 61$. Come facciamo ad analizzare il numero di

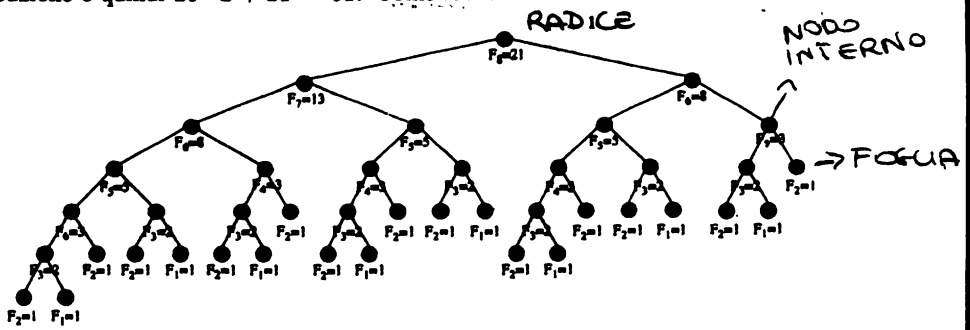


Figura 1.5 Chiamate ricorsive della funzione `fibonacci2` per il calcolo di F_8.

linee di codice mandato in esecuzione per una generica chiamata alla funzione `fibonacci2`(n)? I seguenti lemmi rispondono a questa domanda.

Lemma 1.1 *Sia T_n l'albero delle chiamate ricorsive della funzione* `fibonacci2`(n). *Il numero di foglie in T_n è esattamente uguale al numero di Fibonacci F_n.*

Dimostrazione. Dimostriamo il lemma per induzione. La base è banalmente verificata: se $n = 1$, $F_1 = 1$ e l'albero della ricorsione T_1 contiene un solo nodo e quindi una sola foglia. Anche per $n = 2$, $F_2 = 1$ e l'albero della ricorsione T_2 contiene un solo nodo ed una sola foglia.

Dimostriamo ora il passo induttivo. Sia $n > 2$ ed assumiamo induttivamente che l'enunciato del lemma sia verificato per ogni $k \leq (n-1)$. Vogliamo dimostrare che il lemma vale anche per $k = n$. Se $n > 2$, sappiamo che l'albero della ricorsione T_n ha come sottoalbero sinistro T_{n-1} (l'albero delle chiamate ricorsive della funzione `fibonacci2`$(n-1)$) e come sottoalbero destro T_{n-2} (l'albero delle chiamate ricorsive della funzione `fibonacci2`$(n-2)$). Per l'ipotesi induttiva, il numero delle foglie di T_{n-1} è esattamente uguale a F_{n-1}, mentre il numero delle foglie di T_{n-2} è esattamente uguale a F_{n-2}. Il numero totale delle foglie in T_n sarà quindi uguale a $F_{n-1} + F_{n-2} = F_n$. □

Come conseguenza del Lemma 1.1, il numero totale di foglie contenute nell'albero della ricorsione della funzione `fibonacci2`(n) è dato esattamente da F_n. Nell'albero della ricorsione della funzione `fibonacci2`(n) ogni foglia è responsabile di una linea di codice (base della ricorsione), mentre ogni nodo interno dell'albero è responsabile di due linee di codice (chiamata ricorsiva). Per contare il numero di nodi interni, possiamo usare le proprietà di alberi in cui ogni nodo interno ha due figli. In particolare, il seguente lemma dimostra che il numero di nodi interni di un tale albero binario è sempre uguale al numero di foglie meno uno.

Lemma 1.2 *Sia T un albero binario in cui ogni nodo interno ha esattamente due figli. Allora il numero di nodi interni di T è sempre uguale al numero di foglie diminuito di uno.*

Dimostrazione. Ancora una volta la dimostrazione procederà per induzione. Sia n il numero di nodi dell'albero T. Esaminiamo la base dell'induzione: se $n = 1$, T ha solo una foglia e nessun nodo interno, quindi la condizione è banalmente verificata.

Supponiamo ora per ipotesi che la condizione valga per tutti gli alberi con meno di n nodi, e dimostriamo che deve valere anche per T. Se f e i sono il numero di foglie e di nodi interni di T, rispettivamente, vogliamo quindi dimostrare che $i = f - 1$. Sia $\widehat{T}$ un albero ottenuto da T rimuovendo una qualunque coppia di foglie con lo stesso padre. Questo albero avrà $i - 1$ nodi interni e $f - 2 + 1 = f - 1$ foglie. Per l'ipotesi induttiva, poiché $\widehat{T}$ ha meno nodi di T, si ha che $(i-1) = (f-1) - 1$. Sommando a entrambi i membri 1 si ottiene proprio l'uguaglianza $i = f - 1$ che si voleva dimostrare. □

Sia T_n l'albero di ricorsione della funzione `fibonacci2`(n). In base al Lemma 1.1 il numero di foglie di T_n è esattamente uguale a F_n, mentre per il Lemma 1.2 il numero di nodi interni di T_n è esattamente uguale a $(F_n - 1)$. Notiamo che ogni foglia dell'albero di ricorsione T_n corrisponde al caso base (linea 1) della funzione ricorsiva `fibonacci2`(n)): per ogni foglia in T_n viene quindi mandata in esecuzione una sola linea di codice, ed in particolare la linea 1. In maniera


```
    algoritmo fibonacci3(intero n) → intero
1.     Sia Fib un array di n interi
2.     Fib[1] ← Fib[2] ← 1
3.     for i = 3 to n do
4.         Fib[i] ← Fib[i − 1] + Fib[i − 2]
5.     return Fib[n]
```

Figura 1.6 Algoritmo `fibonacci3` per il calcolo dell'n-esimo numero di Fibonacci.

analoga, ogni nodo interno dell'albero di ricorsione T_n corrisponde alla ricorsione nella funzione `fibonacci2`(n)): per ogni foglia in T_n vengono quindi mandate in esecuzione due linee di codice, la linea 1 e la linea 2.

In sintesi, vengono dunque mandate in esecuzione F_n linee di codice (una per ogni foglia dell'albero di ricorsione), più $(2 \cdot F_n - 2)$ linee di codice (due per ogni nodo interno dell'albero di ricorsione), per un totale di $(3 \cdot F_n - 2)$ linee di codice. L'algoritmo `fibonacci2`(n) sembra indubbiamente molto lento: il numero di linee di codice mandate in esecuzione cresce come i conigli di Fibonacci! Ad esempio, per $n = 8$ vengono mandate in esecuzione

$$3 \cdot F_8 - 2 = 3 \cdot (21) - 2 = 61$$

linee di codice. Per $n = 45$ l'algoritmo richiede

$$3 \cdot F_{45} - 2 = 3 \cdot 1.134.903.170 - 2 = 3.404.709.508$$

ovvero più di tre miliardi di linee di codice! Possiamo fare di meglio? La risposta è affermativa, e la vedremo nel prossimo paragrafo.

1.4 Un algoritmo iterativo

Perché l'algoritmo `fibonacci2` è lento? Perché continua a ricalcolare ripetutamente la soluzione dello stesso sottoproblema! Ad esempio, consideriamo l'albero delle chiamate ricorsive per il calcolo di F_8. La seconda volta che compare, ad esempio, F_4, perdiamo tempo a ricalcolarlo: in realtà, lo abbiamo già calcolato, e la risposta non cambierà di certo. Per fare di meglio, potremmo quindi risolvere ogni sottoproblema una volta soltanto, memorizzare questa soluzione, ed usarla nel seguito invece di ricalcolarla.

Questa semplice idea è alla base di una tecnica algoritmica, chiamata *programmazione dinamica*, che vedremo più in dettaglio nel Capitolo 10, e che può produrre algoritmi molto più complicati di questo. Nel nostro caso, la soluzione è molto semplice, come può desumersi dall'esame della Figura 1.6. L'algoritmo `fibonacci3` è un algoritmo iterativo, in cui utilizziamo dei cicli al posto della ricorsione. Per questo motivo, dobbiamo analizzarlo in modo un po' diverso da un

	fibonacci2(58)	fibonacci3(58)
Pentium IV 1700MHz	15820 sec. ($\simeq$ 4 ore)	0.7 milionesimi di secondo
Pentium III 450MHz	43518 sec. ($\simeq$ 12 ore)	2.4 milionesimi di secondo
PowerPC G4 500MHz	58321 sec. ($\simeq$ 16 ore)	2.8 milionesimi di secondo

Figura 1.7 Tempo richiesto su varie architetture dall'implementazione in C degli algoritmi `fibonacci2` e `fibonacci3` su input $n = 58$.

algoritmo ricorsivo. In pratica, per ogni linea di codice calcoliamo quante volte essa è eseguita, esaminando a quali cicli appartiene e quante volte tali cicli sono eseguiti.

In ogni caso, si mandano in esecuzione sempre tre linee di codice (righe 1, 2 e 5). La prima linea del ciclo (riga 3) viene eseguita $(n-1)$ volte, ad eccezione del caso $n = 1$. La seconda linea di codice del ciclo (riga 4) viene eseguita $(n-2)$ volte, ad eccezione del caso $n = 1$. Quindi, indicando con $T(n)$ il numero di linee di codice mandate in esecuzione dalla funzione `fibonacci3`(n), si ha

$$T(n) = n - 1 + n - 2 + 3 = 2n$$

ad eccezione del caso $T(1) = 4$.

Ad esempio, per $n = 45$ ci vogliono 90 passi, e l'algoritmo `fibonacci3` è all'incirca 38 milioni di volte più veloce dell'algoritmo `fibonacci2`, che invece richiede $3 \cdot F_{45} - 2 = 3.404.709.508$ di passi. Per $n = 58$ l'algoritmo `fibonacci3` richiede 116 passi, mentre l'algoritmo `fibonacci2` richiede $3 \cdot F_{58} - 2 = 1.773.860.189.635$, ovvero più di 1700 miliardi di passi! Quindi, nel caso $n = 58$, ci aspettiamo che l'algoritmo `fibonacci3` sia all'incirca 15 miliardi di volte più veloce dell'algoritmo `fibonacci2`. La Figura 1.7 evidenzia, per $n = 58$, le differenze di efficienza degli algoritmi `fibonacci2` e `fibonacci3` implementati in C su architetture disponibili in commercio nel momento in cui questo libro è stato scritto. I dati riportati confermano la nostra predizione teorica sul numero di linee di codice mandate in esecuzione: nelle varie architetture, l'implementazione di `fibonacci3` risulta essere in pratica da 18 a 22 miliardi di volte più veloce di quella di `fibonacci2`.

1.5 Occupazione di memoria

Il tempo di esecuzione non è l'unica misura che ci interessa, o l'unica metrica che può essere analizzata per valutare l'efficienza di un algoritmo. Anche il tempo richiesto per scrivere il codice (tempo di programmazione) e la lunghezza del codice prodotto sono parametri importanti, ma sono nozioni che saranno approfondite in maggior dettaglio nei corsi di ingegneria del software. In questo libro, analizzeremo invece frequentemente, oltre al tempo di esecuzione, anche la quantità di memoria utilizzata da un programma. Ci sono varie ragioni per preoccuparsi della quantità di memoria richiesta da un programma. Innanzitutto,

se un programma richiede molto tempo, possiamo sempre attendere una quantità sufficiente di tempo per ottenere i risultati dell'elaborazione. A titolo di esempio, se vogliamo calcolare F_{58} con l'algoritmo `fibonacci2` avendo a disposizione una CPU di 1700 MHz, possiamo attendere all'incirca 4 ore e 24 minuti, come riportato in Figura 1.7.

Nel caso in cui un programma richiede una quantità di memoria eccessiva, invece, non si ha neppure la garanzia di ottenere i risultati della sua elaborazione, anche attendendo tempi elevati. Ad esempio, il programma potrebbe richiedere più spazio di quello offerto dal disco rigido, oppure il continuo trasferimento di dati da memoria secondaria (disco rigido) a memoria principale (RAM) potrebbe non fare terminare affatto la sua esecuzione. Anche l'occupazione di memoria, quindi, sembra un parametro molto importante nell'analisi degli algoritmi.

Ancora una volta, anche per l'occupazione di memoria analizzeremo gli algoritmi in modo diverso a seconda che siano iterativi oppure ricorsivi. Per programmi iterativi, di solito è sufficiente esaminare la dichiarazione delle variabili e le chiamate di allocazione (come ad esempio la funzione `malloc()` in C). Per esempio, l'algoritmo `fibonacci3` dichiara solo un array di n numeri interi, e quindi possiamo concludere che la sua richiesta di memoria è proporzionale ad n. L'analisi di programmi ricorsivi è invece più complicata: lo spazio usato in un certo momento è lo spazio totale usato da tutte le chiamate ricorsive attive in quell'istante. Ad esempio, ogni chiamata ricorsiva dell'algoritmo `fibonacci2` richiede una quantità costante di spazio: una quantità costante per le variabili locali e per gli argomenti della funzione, ed una quantità costante per memorizzare l'indirizzo di ritorno. Le chiamate attive ad un certo istante formano un cammino nell'albero della ricorsione che abbiamo visto prima, dove l'argomento di ogni nodo è di una o due unità più piccolo dell'argomento nel nodo padre.

La lunghezza di ogni cammino nell'albero delle chiamate ricorsive è al più n, e quindi lo spazio richiesto dall'algoritmo ricorsivo è ancora proporzionale ad n, modulo fattori moltiplicativi costanti. Abbreviamo la frase "modulo fattori moltiplicativi costanti", usando la notazione asintotica "O" (O grande), di cui daremo una definizione più precisa nel seguito: diciamo che lo spazio richiesto dall'algoritmo `fibonacci2` è $O(n)$, ovvero "O grande di n".

L'algoritmo `fibonacci3` può essere modificato così da richiedere meno spazio. Basta osservare che in realtà ogni iterazione del ciclo utilizza soltanto i due valori precedenti di F_n, ovvero F_{n-1} ed F_{n-2}. Quindi possiamo usare soltanto due variabili al posto dell'intero array. Questa modifica richiede alcuni spostamenti perché tutto risieda al suo posto, come illustrato nella Figura 1.8. In questo codice, la variabile a rappresenta $Fib[i-2]$, b rappresenta $Fib[i-1]$ e c rappresenta $Fib[i]$. Le due assegnazioni dopo la somma preparano questi valori per l'iterazione successiva. Questo algoritmo richiede approssimativamente $4n$ linee di codice per calcolare F_n, e quindi è più lento dell'algoritmo `fibonacci3` per via del fattore moltiplicativo, ma richiede meno spazio: spazio costante invece di spazio $O(n)$.

```
algoritmo fibonacci4(intero n) → intero
    a ← 1, b ← 1
    for i = 3 to n do
        c ← a + b
        a ← b
        b ← c
    return b
```

Figura 1.8 Algoritmo `fibonacci4` per il calcolo dell'n-esimo numero di Fibonacci.

1.6 Notazione asintotica

Prima di descrivere algoritmi più efficienti per calcolare i numeri di Fibonacci, rendiamo la nostra analisi un po' più generale. Un problema con l'analisi che abbiamo condotto finora è definire che cos'è esattamente una linea di codice. Se spezziamo una linea di codice in due, non cambiamo certo la velocità del programma, ma cambiamo il numero di linee di codice eseguite. E se compriamo un computer più veloce, cambiamo la velocità del programma, ma non certo la sua analisi.

Per evitare di analizzare specificità e dettagli non necessari, usiamo la notazione asintotica. L'idea di fondo alla base della nozione di definizione asintotica è la seguente: scriviamo già i tempi in funzione dell'input n e trattiamo due funzioni come uguali, in prima istanza, se una è c volte l'altra, dove c è una constante che non dipende da n. Ad esempio, potremmo rimpiazzare $3F_n - 2$ con $O(F_n)$ e $2n$ e $4n$ con $O(n)$. In modo più formale, diciamo che $f(n) \in O(g(n))$ se esistono due costanti positive c ed n_0 tali che $f(n) \leq cg(n)$ per ogni $n \geq n_0$, ovvero la funzione $f(n)$ cresce al più come $cg(n)$ a partire da un certo valore $n = n_0$ (si veda la Figura 1.9). Ad esempio, dato che $F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$, F_n cresce esponenzialmente in n, e quindi il fatto che $\phi < 2$ implica $F_n = O(2^n)$.

Perché usare la notazione asintotica? Innanzitutto, la utilizzeremo per semplificare le formule, ignorando i fattori costanti ed altri dettagli inlnfluenti. Questo ci renderà la vita più semplice: non dovremo preoccuparci di tutti i dettagli di basso livello sul comportamento di un algoritmo. Inotre, cosa più rilevante, la notazione asintotica ci consentirà di confrontare due algoritmi in modo semplice. Ad esempio, gli algoritmi `fibonacci3` e `fibonacci4` richiedono entrambi l'esecuzione di $O(n)$ linee di codice. Analizzando esattamente il numero di linee di codice eseguite, un algoritmo è circa due volte più veloce dell'altro, ma questo rapporto non varia al variare di n. Dall'esame di altri fattori, come ad esempio il tempo necessario per l'allocazione dell'array nell'algoritmo `fibonacci3`, può venir fuori che i due algoritmi in realtà possano essere più vicini come prestazioni: per determinare quale dei due sia preferibile, servirà un'analisi più attenta e sperimentale. D'altronde, sappiamo che $4n$ è molto minore di $(3F_n - 2)$ per

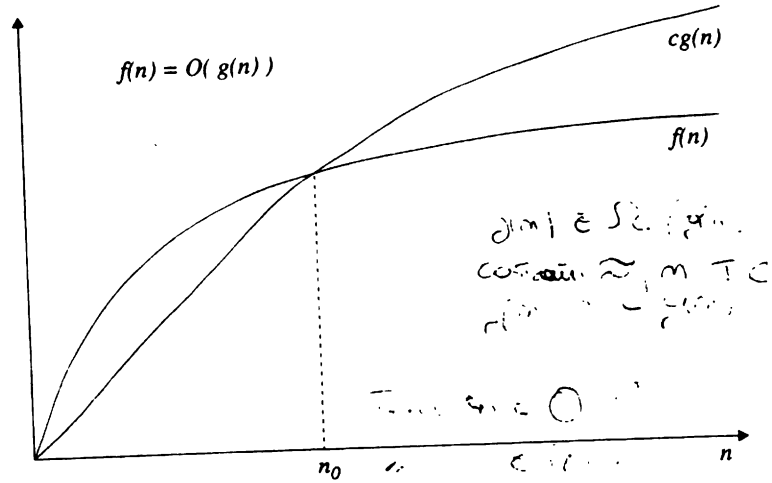


Figura 1.9 Notazione asintotica O.

ogni valore ragionevole di n, e questo non dipende dalla costante moltiplicativa 4: sarebbe vero anche per $8n$ o $15n$. Al crescere di n, il rapporto F_n/n diventa talmente grande da rendere sicuramente l'algoritmo `fibonacci3` preferibile all'algoritmo `fibonacci2`. Sostituire $4n$ con $O(n)$ è un'astrazione che ci consente di confrontarlo facilmente con altre funzioni senza tenere in considerazione dettagli poco rilevanti, quali la costante moltiplicativa 4.

1.7 Un algoritmo basato su potenze ricorsive

Forse sembra difficile crederlo, ma i precedenti algoritmi, `fibonacci3` e `fibonacci4`, non sono i più efficienti possibili. Sfruttando semplici proprietà delle matrici si può dimostrare il seguente lemma.

Lemma 1.3 *Sia* $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. *Allora*

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

Dimostrazione. La dimostrazione procede per induzione su n. Per convenzione, fissiamo $F_0 = 0$. Per $n = 2$ il passo base è banalmente verificato:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$$

```
algoritmo fibonacci5(intero n) → intero
    M ← ( 1 0 )
        ( 0 1 )
    for i = 1 to n − 1 do
        M ← M · ( 1 1 )
                ( 1 0 )
    return M[0][0]
```

Figura 1.10 Algoritmo `fibonacci5` per il calcolo dell'n-esimo numero di Fibonacci.

Assumiamo ora che valga induttivamente l'uguaglianza per $n-1$, ovvero:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix}$$

Moltiplicando entrambi i membri dell'uguaglianza per la matrice A si ottiene

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} F_{n-1} + F_{n-2} & F_{n-1} \\ F_{n-2} + F_{n-3} & F_{n-2} \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

□

Il Lemma 1.3 ci consente di definire un altro algoritmo iterativo basato su moltiplicazioni di matrici. Come può desumersi dall'esame della Figura 1.10, inizializziamo M alla matrice identità (potenza zero di A), e poi moltiplichiamo ripetutamente M per A fino ad ottenere la potenza $(n-1)$-esima. Quindi, secondo la formula precedente, l'elemento in alto a sinistra è esattamente F_n, il valore da restituire. Questo è esattamente quello che fa l'algoritmo `fibonacci5` in Figura 1.10.

Osserviamo che la moltiplicazione di matrici non è un'istruzione primitiva di linguaggi di programmazione come C, C++ e Java, e quindi il nostro pseudocodice comincia ad allontanarsi da un linguaggio di programmazione reale. Questo ci consente di mantenere un buon livello di astrazione ignorando dettagli irrilevanti e focalizzando la nostra attenzione sugli aspetti essenziali del procedimento risolutivo.

È facile verificare che l'algoritmo `fibonacci5` richiede tempo $O(n)$ ed è quindi molto più veloce dell'algoritmo `fibonacci2`. Ci aspettiamo invece che sia più lento in pratica degli algoritmi `fibonacci3` o `fibonacci4`, a causa delle operazioni per la gestione delle matrici. La notazione asintotica nasconde le differenze tra questi algoritmi, e quindi bisogna essere cauti nel decidere chi di essi è in pratica il più veloce. Ancora una volta, `fibonacci5` ha bisogno di una


```
algoritmo fibonacci6(intero n) → intero
    M ← ( 1 0 )
        ( 0 1 )
    potenzaDiMatrice(M, n − 1)
    return M[0][0]

procedura potenzaDiMatrice(matrice M, intero n)
    if ( n > 1 ) then
        potenzaDiMatrice(M, n/2)
        M ← M · M
    if ( n è dispari ) then M ← M · ( 1 1 )
                                    ( 1 0 )
```

Figura 1.11 Algoritmo `fibonacci6` per il calcolo dell'n-esimo numero di Fibonacci.

quantità costante di memoria, ovvero spazio $O(1)$. Qual è dunque il vantaggio nell'usare moltiplicazione di matrici rispetto all'algoritmo `fibonacci4`?

In realtà, possiamo calcolare la matrice M^{n-1} molto più efficientemente. L'idea di base è la seguente: per calcolare, ad esempio, 3^8, si può moltiplicare 3 per otto volte (e.g., $3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 6561$) oppure ricorrere a quadrati ripetuti (ovvero, $3^2 = 9$, $9^2 = 3^4 = 81$, $81^2 = 3^8 = 6561$). Con il metodo dei quadrati ripetuti si effettua un numero molto minore di moltiplicazioni. Con un po' di attenzione, lo stesso metodo può essere esteso a esponenti che non sono potenze di due e può anche essere adattato a matrici, come mostrato nella Figura 1.11. In pratica, tutto il tempo richiesto dall'algoritmo `fibonacci6` di Figura 1.11 è speso nella procedura `potenzaDiMatrice`, che calcola ricorsivamente la potenza n-esima della matrice M elevando a quadrato la sua potenza $(n/2)$-esima. Se n è dispari, l'arrotondamento di $n/2$ produce la potenza $(n-1)$-esima: possiamo aggiustare questo moltiplicando ancora una volta per A. Questo algoritmo è ricorsivo, e per analizzarlo usiamo di nuovo una relazione di ricorrenza. Se $n \leq 1$ l'algoritmo richiede un tempo di esecuzione costante. D'altro canto, il tempo relativo ad una chiamata a `potenzaDiMatrice` con argomento $n > 0$ è $O(1)$, più il tempo di una chiamata ricorsiva con argomento $n/2$:

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

Dimostriamo ora il seguente lemma.

Lemma 1.4 *La relazione di ricorrenza*

$$T(n) = \begin{cases} O(1) + T(n/2) & \textit{se } n > 1 \\ O(1) & \textit{se } n \leq 1 \end{cases}$$

ammette come soluzione $T(n) = O(\log n)$.

Dimostrazione. Assumiamo per semplicità che n sia una potenza di 2, e riscriviamo la relazione di ricorrenza in modo leggermente diverso:

$$T(n) = \begin{cases} a + T(n/2) & \text{se } n > 1 \\ b & \text{se } n \leq 1 \end{cases}$$

per due opportune costanti $a \geq 0$, $b \geq 0$. Notiamo che

$$T(n/2) \leq a + T(n/4),$$

e quindi sostituendo nella precedente si ottiene:

$$T(n) \leq a + T(n/2) \leq a + a + T(n/4)$$

Continuando con questo procedimento, otteniamo

$$\begin{array}{rcl} T(n) & \leq & a + T(n/2) \quad \leq \\ & \leq & 2 \cdot a + T(n/4) \quad \leq \\ & \leq & 3 \cdot a + T(n/8) \quad \leq \\ & \cdots & \cdots \quad \cdots \\ & \leq & k \cdot a + T(n/2^k) \end{array}$$

per ogni k, $1 \leq k \leq \log_2 n$. Per $k = \log_2 n$ otteniamo:

$$T(n) \leq k \cdot a + T(n/2^k) \leq a \log_2 n + T(1) \leq a \log_2 n + b = O(\log n)$$

□

Vedremo nel Capitolo 2 metodi generali e più formali per risolvere relazioni di ricorrenza. In questo libro, useremo i logaritmi in base 2, arrotondandoli ad interi, e quindi $\log_2 n$ è in pratica il numero di bit necessari per scrivere n in binario (oppure, il più piccolo valore di i tale che $n < 2^i$). Se n vale un miliardo, $\log_2 n$ sarà 30, e l'algoritmo `fibonacci6` sarà più veloce degli algoritmi `fibonacci3` e `fibonacci4` nello stesso modo in cui gli algoritmi `fibonacci3` e `fibonacci4` sono più veloci dell'algoritmo `fibonacci2`.

Prima di concludere questo capitolo, vorremmo fare alcune osservazioni. Per prima cosa, osserviamo che all'interno della notazione asintotica O abbiamo omesso la base dei logaritmi. Questo può essere spiegato se si considera che, date due costanti a e b, con a e b maggiori di zero ($a > 0$, $b > 0$), si effettua un cambiamento di base dei logaritmi secondo la nota formula:

$$\log_b n = \log_b a \cdot \log_a n$$

Tale uguaglianza implica che

$$O(\log_b n) = O(\log_b a \cdot \log_a n) = O(\log_a n)$$

All'interno della notazione asintotica O, quindi, la base di un logaritmo può essere trascurata, in quanto si può cambiare base a meno di moltiplicare per una opportuna costante.

	Tempo di esecuzione	Memoria
`fibonacci2`	$O(2^n)$	$O(n)$
`fibonacci3`	$O(n)$	$O(n)$
`fibonacci4`	$O(n)$	$O(1)$
`fibonacci5`	$O(n)$	$O(1)$
`fibonacci6`	$O(\log n)$	$O(\log n)$

Tabella 1.1 Tempi di esecuzione e memoria richiesta dagli algoritmi per il calcolo di F_n, n-esimo numero di Fibonacci, in funzione del parametro n.

La seconda osservazione, di natura più metodologica, è relativa al parametro che abbiamo utilizzato per analizzare i tempi di esecuzione degli algoritmi presentati finora. In particolare, in tutto il capitolo abbiamo espresso i tempi di esecuzione degli algoritmi per il calcolo di F_n, n-esimo numero di Fibonacci, in funzione del parametro n, come illustrato nella Tabella 1.1. Ma siamo proprio sicuri di aver considerato il parametro giusto? Dato un generico problema $\mathcal{P}$ (come ad esempio il calcolo di F_n), ed un algoritmo $\mathcal{A}$ per risolvere il problema $\mathcal{P}$, l'algoritmo $\mathcal{A}$ riceverà in ingresso un'istanza I del problema $\mathcal{P}$, che ad esempio nel nostro caso è l'intero n, e produrrà una soluzione al problema $\mathcal{P}$, che nel nostro caso è F_n. In questo scenario, sembra naturale esprimere il tempo di esecuzione dell'algoritmo $\mathcal{A}$ in funzione della dimensione dell'istanza di ingresso, ovvero come una funzione $f(|I|)$. Se quindi il problema $\mathcal{P}$ è il calcolo di F_n, così come considerato in questo capitolo, allora l'istanza di ingresso è n, e la dimensione dell'istanza di ingresso è $|I| = O(\log n)$, dato che sono sufficienti $\lceil \log_2 n \rceil$ bit per descrivere l'intero n. In questo scenario, il tempo di esecuzione degli algoritmi presentati in questo capitolo deve essere correttamente espresso in funzione della dimensione dell'istanza di ingresso $|I|$, come illustrato nella Tabella 1.2. Scopriamo quindi che l'algoritmo `fibonacci2` è in realtà doppiamente esponenziale nella dimensione dell'istanza di ingresso!

	Tempo di esecuzione	Memoria
`fibonacci2`	$O\left(2^{2^{\lvert I\rvert}}\right)$	$O\left(2^{\lvert I\rvert}\right)$
`fibonacci3`	$O\left(2^{\lvert I\rvert}\right)$	$O\left(2^{\lvert I\rvert}\right)$
`fibonacci4`	$O\left(2^{\lvert I\rvert}\right)$	$O(1)$
`fibonacci5`	$O\left(2^{\lvert I\rvert}\right)$	$O(1)$
`fibonacci6`	$O(\lvert I\rvert)$	$O(\lvert I\rvert)$

Tabella 1.2 Tempo di esecuzione e memoria richiesta dagli algoritmi per il calcolo di F_n, n-esimo numero di Fibonacci in funzione della dimensione dell'istanza di ingresso $|I|$, $|I| = O(\log n)$.

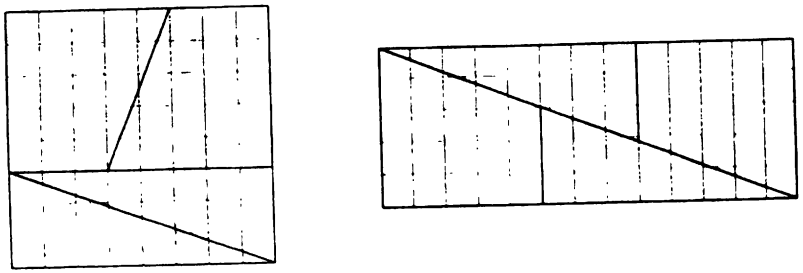


Figura 1.12 La successione di Fibonacci e un'illusione geometrica.

Anche l'ultima osservazione è di natura metodologica, e riguarda tutti gli algoritmi presentati nel corso di questo capitolo. Come sappiamo, il valore F_n cresce esponenzialmente all'aumentare di n e quindi, anche per valori di n non troppo grandi, F_n potrebbe non essere più rappresentabile in una sola parola di memoria del calcolatore, che ha una lunghezza limitata: ad esempio, già F_{48} non è rappresentabile usando parole di memoria con 32 bit. Operare con numeri così grandi implica quindi che, anche per poter eseguire una semplice operazione di addizione o moltiplicazione, la si debba scomporre in una sequenza di lunghezza non costante di passi su numeri più piccoli. L'analisi dovrebbe quindi tenere in considerazione questo aspetto, che abbiamo in prima approssimazione ignorato assumendo che tutte le operazioni aritmetiche elementari abbiano un costo $O(1)$. Vedremo come affrontare questo problema nel Capitolo 2.

1.8 Problemi

Problema 1.1 In una successione di Fibonacci, in cui ogni numero è ottenuto dalla somma dei due precedenti, i primi due numeri non devono essere necessariamente 1. Scegliete due numeri qualunque e generate una successione di Fibonacci a partire dai numeri prescelti: la somma dei primi dieci numeri della successione è sempre pari a undici volte il settimo numero! Riuscite a dimostrarlo per ogni scelta dei numeri di partenza?

Problema 1.2 Prendete un quadrato di 8×8 quadretti. Tagliatelo in quattro pezzi e riassemblatelo in un rettangolo come mostrato in Figura 1.12. Il quadrato originale aveva 64 quadretti, mentre il rettangolo prodotto ne ha 65!

(1) Come è possibile?
(2) È possibile estendere il ragionamento ad ogni quadrato di $F_n \times F_n$ quadretti?

Problema 1.3 Il Professor Tribonacci, dell'Università di Pizza, ha definito la sequenza dei numeri di Tribonacci:

$$X_k = \begin{cases} 0 & \text{se } k = 0 \\ 1 & \text{se } k = 1, 2 \\ X_k = X_{k-1} + X_{k-2} + X_{k-3} & \text{se } k \geq 3 \end{cases}$$

Il Prof. Tribonacci sostiene che l'algoritmo più efficiente per calcolare l'n-mo numero di Tribonacci X_n è dato dal seguente pseudocodice:

```
    algoritmo tribonacci(intero n) → intero
1.    if ( n = 0 ) then return 0
2.    else if ( n ≤ 2 ) then return 1
3.    else return tribonacci(n − 1) + tribonacci(n − 2) +
                  tribonacci(n − 3)
```

Sia $T(n)$ il tempo di esecuzione dell'algoritmo `tribonacci` su input n.

(a) Scrivere una relazione di ricorrenza per $T(n)$.
(b) Stimare $T(n)$ per n grande.
(c) Descrivere un algoritmo più efficiente per lo stesso problema, e dimostrare il suo tempo di esecuzione.

1.9 Sommario

In questo capitolo abbiamo introdotto, tramite un semplice esempio relativo al calcolo dei numeri di Fibonacci, l'importanza del progetto di algoritmi efficienti per le prestazioni dei sistemi software. Questo esempio, nonostante la sua semplicità, sembra contenere già *in nuce* molte delle problematiche che saranno centrali in questo libro.

Innanzitutto, abbiamo scoperto che progettare algoritmi efficienti può avere un effetto drammatico sull'incremento delle prestazioni. Ma *come misuriamo l'efficienza di un algoritmo?* Come abbiamo osservato, sembra naturale tentare di misurare le risorse richieste da un algoritmo darante la sua esecuzione: la quantità di tempo di calcolo (tempo di CPU) e di spazio (spazio di memoria). Per quanto riguarda il tempo di calcolo, sembra cruciale definire una metrica che sia indipendente dalle tecnologie e dalle piattaforme utilizzate, ad esempio misarando opportunamente il namero di passi richiesto da un algoritmo piuttosto che il suo tempo fisico di esecuzione in un particolare sistema di elaborazione.

Ma in funzione di cosa esprimiamo le risorse richieste da un algoritmo? Per misurare il tempo di esecuzione o l'occupazione di memoria di un algoritmo senza dover guardare i dettagli della particolare istanza del problema, sembrerebbe una buona idea esprimere tali grandezze in funzione della dimensione dell'istanza di ingresso stessa. Per riuscire a confrontare algoritmi diversi, senza arrivare allo stadio finale di implementazione del relativo codice, non sembra necessario disporre del numero esatto dei passi eseguiti dall'algoritmo: una semplice stima dell'ordine di grandezza può già darci un'idea di massima su quale algoritmo sarà più

Figura 1.13 La sorprendente presenza dei numeri di Fibonacci e della spirale aurea in molte specie vegetali e in comuni organismi marini.

veloce. Per questo motivo abbiamo introdotto informalmente la notazione asintotica O, che ci consente di ignorare, in prima approssimazione, fattori e dettagli poco rilevanti.

Infine, come abbiamo visto in questo capitolo introduttivo, sia la fase di progetto che quella di analisi di un semplice algoritmo evidenziano la necessità di ricorrere a strumenti matematici, come ad esempio la risoluzione di relazioni di ricorrenza. Tutti questi concetti, qui introdotti in maniera del tutto informale, saranno definiti più formalmente e con rigore matematico nei capitoli seguenti.

1.10 Note bibliografiche

Gli algoritmi hanno radici storiche profonde, e sono stati utilizzati sin dall'antichità per varie attività, come predire il futuro, prescrivere cure mediche, o preparare cibi. La parola "algoritmo" deriva direttamente da al-Khwārizmī, autore di uno dei più antichi trattati di algebra. Il suo nome completo, Muḥammad ibn Mūsā al-Khwārizmī significa letteralmente Muḥammad figlio di Mūsā da Khwarezm, una regione dell'Asia centrale a sud del Lago di Aral. Muḥammad al-Khwārizmī era un matematico vissuto nel IX secolo, ed il suo libro, *"al-Mukhtaṣar fī al-Jabr wa l-Muqābala"*, ci ha regalato anche la parola "algebra" (da al-Jabr).

La semplicità della relazione di Fibonacci può forse spiegare perché i numeri di Fibonacci appaiono in vari fenomeni naturali. Essi sono infatti presenti in varia misura in botanica: non solo numerosi fiori hanno un numero di Fibonacci di petali (come il girasole in Figura 1.13), ma la successione di Fibonacci si ritrova anche nel rapporto fillotattico, da una parola greca che significa "disposizione delle foglie". Ad esempio, scegliete una foglia su uno stelo ed assegnatele il numero 0; contate poi quante foglie intercorrono fino ad incontrare una foglia perfettamente allineata. Se nessuna foglia è stata asportata, con ogni probabilità otterrete un numero di Fibonacci! Gli ordinamenti delle scaglie degli ananas e delle brattee sulle pigne esibiscono peculiarità simili.

I numeri di Fibonacci compaiono anche nel triangolo di Pascal, nella formula binomiale, nella sezione aurea usata in molte opere architettoniche ed artistiche: Seurat, Mondrian, Leonardo da Vinci, Dalì hanno tutti usato il rettangolo aureo in qualcuna delle loro opere! Come visto nei Problemi 1.1 e 1.2, la successione di Fibonacci è infine protagonista indiscussa di curiosi giochi matematici. In particolare, il Problema 1.2 è un paradosso geometrico basato sull'identità di Cassini [1]:

$$F_{n+1}F_{n-1} - (F_n)^2 = (-1)^n. \qquad \text{per } n > 0$$

e pare fosse uno dei puzzle favoriti di Lewis Carroll [2, 3, 4], l'autore di *"Alice nel Paese delle Meraviglie"*.

Riferimenti bibliografici

[1] Jean-Dominique Cassini, "Une nouvelle progression de nombres", *Histoire de l'Académie Royale des Sciences* (1733), Paris, vol. 1, p. 201.

[2] Stuart Dodgson Collingwood, *The Lewis Carrol Picture Book*. T. Fisher Unwin, 1899.

[3] O. Schlömilch, "Ein geometrisches paradoxon", *Zeitschrift für Mathematik und Physik* 13 (1868), p. 162.

[4] Warren Weaver, "Lewis Carrol and a geometrical paradox", *American Mathematical Monthly* 45 (1938), p. 234–236.

2

Modelli di calcolo e metodologie di analisi

To iterate is human, to recurse, divine.
(Robert Heller)

In questo capitolo introdurremo le principali notazioni e metodologie utilizzate nel corso del libro per l'analisi di algoritmi. Alcune utili nozioni matematiche, tra cui definizioni e proprietà dei logaritmi, serie e successioni, ed elementi di calcolo combinatorio e della probabilità, sono invece richiamate in modo sintetico nell'Appendice.

2.1 Modelli di calcolo

Per poter analizzare e studiare l'efficienza di algoritmi, dobbiamo innanzitutto definire un *modello di calcolo* appropriato. Storicamente, il primo modello fu proposto dal matematico e logico inglese Alan Turing ed è infatti noto come *macchina di Turing* [17]. Informalmente, la macchina di Turing consiste in un dispositivo di contmllo che accede ad un nastm infinito bidirezionale (la memoria) diviso in celle. Ciascuna cella può contenere un simbolo di un certo alfabeto finito. L'accesso al nastm avviene tramite una testina che può spostarsi a destra o a sinistra di una posizione per volta; un passo di calcolo permette di scrivere un simbolo in una cella, leggere un simbolo da una cella, spostare la testina, o cambiare lo stato di contmllo. Sebbene fondamentale nello studio della calcolabilità e della complessità computazionale, la macchina di Turing risulta però tmppo di basso livello per i nostri scopi e – soprattutto – non sufficientemente realistica per permettere un'analisi accurata di algoritmi utilizzati in pratica.

Un modello più adeguato, introdotto in [14], si ispira all'architettura di von Neumann, la struttura su cui si basano gli attuali calcolatori, ed è noto come *macchina a registri*. Una macchina a registri consiste in un programma finito, un nastro di ingresso, un nastro di uscita, ed una memoria strutturata in un array di n parole, ciascuna con un indirizzo tra 1 e n. Le parole di memoria possono contenere un qualunque valore intero o reale. Un registm, detto accumulatore, contiene istante per istante uno degli operandi su cui agisce l'istruzione corrente. Un altro registro, detto contatore delle istruzioni, contiene l'indirizzo della successiva

istruzione da eseguire. Un passo di calcolo consiste nell'eseguire un'istruzione di ingresso/uscita (lettura o stampa), un'operazione aritmetico/logica, un accesso o una modifica del contenuto della memoria. Intuitivamente, le istruzioni coincidono quindi con quelle di un elementare linguaggio macchina. Sottolineiamo che una caratteristica fondamentale della macchina a registri è l'accesso diretto alla memoria.

Un modello di calcolo meno potente è la cosiddetta *macchina a puntatori*, in cui la memoria è vista come una collezione estendibile di nodi, ciascuno diviso in un numero fissato di campi: ciascun campo può contenere un numero o un puntatore ad un altro nodo. In ogni istante è possibile accedere ad un nodo corrente tramite un puntatore contenuto in un apposito registro. Diversamente dalla macchina a registri, quindi, non è possibile fare operazioni aritmetiche sugli indirizzi (come ad esempio succede nel caso di array): gli algoritmi e le strutture dati che richiedono aritmetica degli indirizzi (quali le tavole hash descritte nel Capitolo 7) non possono pertanto essere implementati nella macchina a puntatori. La macchina a puntatori risulta comunque un modello realistico per analizzare algoritmi basati su strutture dati collegate (vedi Capitolo 3).

Entrambi questi modelli assumono che gli algoritmi siano sequenziali, poiché vengono eseguiti un passo alla volta, e deterministici, perché il futuro comportamento della macchina è determinato univocamente dal suo stato corrente. Vale la pena notare che, grazie agli enormi progressi nella tecnologia, modelli di calcolo paralleli sono sempre più diffusi, e questo ha portato a varie generalizzazioni della macchina a registri. In entrambi i modelli illustrati (macchina a registri e macchina a puntatori) ci sono inoltre due aspetti poco realistici. Innanzitutto, si assume che la macchina possa manipolare in tempo costante valori di dimensione arbitraria: ci siamo già imbattuti in questo aspetto nel Capitolo 1, relativamente al calcolo dei numeri di Fibonacci, e ne parleremo in maggior dettaglio nel Paragrafo 2.1.1. In secondo luogo, c'è una visione piatta della memoria, che non corrisponde alla struttura dei modemi calcolatori che hanno una intera gerarchia di memoria che consiste (almeno) in registri, cache di primo e secondo livello, memoria RAM e disco esterno: non tutti gli accessi a memoria richiedono quindi lo stesso tempo, come vedremo nel Paragrafo 2.8.

2.1.1 Criteri di costo uniforme e logaritmico

Nella descrizione precedente abbiamo contato ciascuna operazione come un singolo passo, indipendentemente dalla dimensione degli operandi coinvolti nella operazione stessa. Questo criterio di misurazione del tempo di esecuzione di un algoritmo è noto come *misura di costo uniforme*. Sebbene utile in prima approssimazione, si tratta però di un modello troppo idealizzato: esso assume infatti che registri e memoria siano costituiti da parole di langhezza arbitrariamente grande. In pratica, le celle di memoria dei modemi calcolatori contengono invece interi rappresentabili con un numero prefissato di bit (tipicamente 32 o 64). Ci siamo già imbattuti in questo problema nel Capitolo 1: come osservato nel Paragrafo 1.7, poiché i valori dei numeri di Fibonacci crescono rapidamente, tutti gli algoritmi

proposti nel Capitolo 1 soffrono del fatto che i dati da manipolare potrebbero non essere rappresentabili in una sola parola di memoria, e quindi anche semplici operazioni aritmetiche non richiederebbero in realtà tempo costante.

Per ovviare a questo inconveniente, è stato proposto un altro criterio di misurazione, noto come *misura di costo logaritmico*, che assume che il costo di esecuzione delle istruzioni dipenda dalla dimensione degli operandi coinvolti. Quindi, eseguire un'operazione su un numero n implica pagare un costo proporzionale a $\log n$. Vediamo su un semplice esempio la differenza tra queste due misure.

Esempio 2.1 Il seguente frammento di programma, dato n, calcola il valore 2^n:

$x \leftarrow 2$
for $i = 1$ **to** n **do** $x \leftarrow x \cdot 2$

Secondo il criterio uniforme, il tempo di esecuzione è banalmente proporzionale a n, poiché la moltiplicazione nel ciclo `for` ha costo costante. Consideriamo ora il criterio di costo logaritmico. Ad ogni passo, il valore della variabile x raddoppia e quindi all'i-esimo passo si ha $x = 2^i$. I tempi spesi per incrementare i e per operare su x sono pari a $\log i$ e $\log x = \log 2^i = i$, rispettivamente. Il tempo di esecuzione è pertanto proporzionale a

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

Usando la serie aritmetica (vedi Paragrafo 17.2 in Appendice), possiamo limitare inferiormente e superiormente questa sommatoria come segue:

$$\frac{n(n+1)}{2} = \sum_{i=1}^{n} i \leq \sum_{i=1}^{n} (i + \log i) \leq \sum_{i=1}^{n} 2i = n(n+1)$$

dimostrando che il tempo di esecuzione è all'incirca proporzionale a n^2. □

Poiché usare il criterio di costo logaritmico a volte potrebbe generare una complicazione non necessaria, un approccio alternativo limita a $c \cdot \log n$ bit la dimensione dei numeri interi rappresentabili, dove c è una costante ed n è la dimensione dell'istanza del problema, e restringe le operazioni permesse sui numeri in virgola mobile. Nel corso di questo libro adotteremo questo approccio: gli algoritmi che presenteremo d'ora in poi sono quindi implementabili su una macchina a registri o su una macchina a puntatori con interi di dimensione $\leq n^c$.

2.2 La notazione asintotica O, Ω, Θ

Nel Capitolo 1 abbiamo discusso come misurare il tempo di esecuzione di un algoritmo, contando prima in maniera esatta il numero di linee di codice mandate in esecuzione per una certa istanza, e poi introducendo un'astrazione, la *notazione*

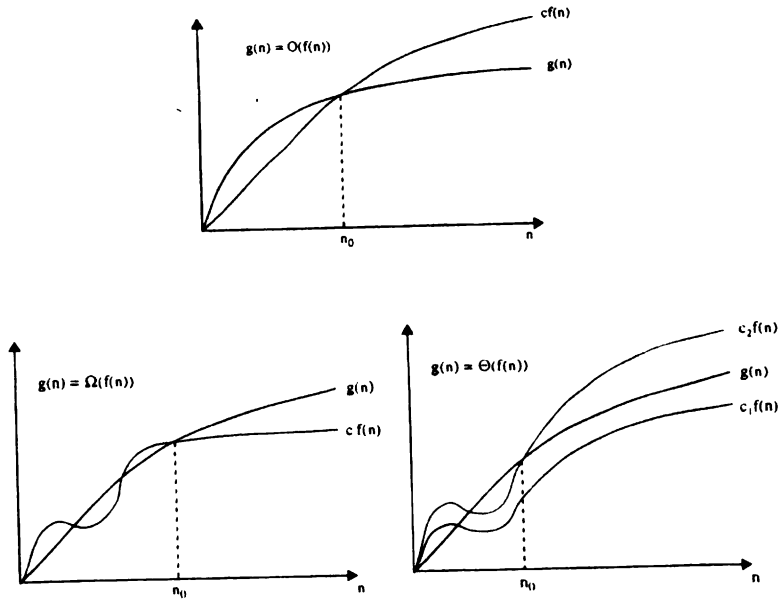


Figura 2.1 Illustrazione della notazione asintotica O, Ω e Θ.

asintotica O, per ignorare i dettagli non influenti come ad esempio le costanti moltiplicative ed i termini di ordine inferiore. La stessa notazione può essere utilizzata per descrivere anche l'uso di altre risorse di calcolo, come ad esempio la quantità di memoria. In questo paragrafo riprenderemo in maniera più formale e completeremo il discorso iniziato nel Paragrafo 1.6 del Capitolo 1. La notazione asintotica, introdotta formalmente nella Definizione 2.1, è mutuata dal calcolo infinitesimale.

Definizione 2.1 *Data una funzione $f(n)$, definiamo:*

- $O(f(n)) = \{g(n) : \exists\, c > 0$ *e* $n_0 \geq 0$ *tali che* $g(n) \leq cf(n)$ *per ogni* $n \geq n_0\}$
- $\Omega(f(n)) = \{g(n) : \exists\, c > 0$ *e* $n_0 \geq 0$ *tali che* $g(n) \geq cf(n)$ *per ogni* $n \geq n_0\}$
- $\Theta(f(n)) = \{g(n) : \exists\, c_1, c_2 > 0$ *e* $n_0 \geq 0$ *tali che, per ogni* $n \geq n_0$, $c_1 f(n) \leq g(n) \leq c_2 f(n)\}$

Intuitivamente, per n sufficientemente grande (ovvero per $n \geq n_0$) e per opportune costanti moltiplicative (c, c_1, c_2 nella Definizione 2.1), si ha:

- se una funzione $g(n) \in O(f(n))$, allora $g(n)$ cresce *al più* come $f(n)$;
- se una funzione $g(n) \in \Omega(f(n))$, allora $g(n)$ cresce *almeno* come $f(n)$;

- se una funzione $g(n) \in \Theta(f(n))$, allora $g(n)$ cresce *esattamente* come $f(n)$.

L'idea è illustrata nella Figura 2.1. Gli insiemi $O(f(n))$, $\Omega(f(n))$ e $\Theta(f(n))$ descrivono quindi delle classi di funzioni che sono limitate superiormente, inferiormente o da ambo i lati da $f(n)$, a meno di costanti moltiplicative. Con un leggero abuso di notazione, nel resto di questo libro scriveremo $g(n) = O(f(n))$ anziché $g(n) \in O(f(n))$: sebbene non precisa, questa notazione presenta infatti alcuni vantaggi che saranno evidenti in seguito. Analoghe considerazioni valgono per la notazione Ω e Θ. Dalla Definizione 2.1 segue immediatamente la seguente proprietà:

Proprietà 2.1 *Date due funzioni $f(n)$ e $g(n)$, risulta $g(n) = \Theta(f(n))$ se e solo se $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$.*

Abbiamo quindi $\Theta(f(n)) \subseteq O(f(n))$ e $\Theta(f(n)) \subseteq \Omega(f(n))$. Diamo ora un esempio d'uso della notazione asintotica. L'esempio illustra chiaramente come questa notazione nasconda sia le costanti moltiplicative sia i termini di ordine inferiore, che sono trascurabili per n sufficientemente grande.

Esempio 2.2 Consideriamo la funzione $g(n) = 3n^2 + 10n$. È facile vedere che $g(n) = O(n^2)$: infatti, scegliendo $c = 4$ e $n_0 = 10$, abbiamo che $3n^2 + 10n \leq 4n^2$ per ogni $n \geq 10$. Analogamente, risulta anche $g(n) = \Omega(n^2)$: infatti $3n^2 + 10n \geq n^2$ per ogni n, da cui $c = 1$ e $n_0 = 0$. Da quanto detto e dalla Proprietà 2.1 segue immediatamente che $3n^2 + 10n = \Theta(n^2)$. Osserviamo inoltre che, ad esempio, $3n^2 + 10n = O(n^3)$, mentre $3n^2 + 10n \notin \Theta(n^3)$. □

Osserviamo che la notazione Θ gode della proprietà di simmetria: $f(n) = \Theta(g(n))$ se e solo se $g(n) = \Theta(f(n))$. O ed Ω soddisfano invece la proprietà detta di simmetria trasposta: $f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$. Inoltre, è facile verificare che O, Ω e Θ sono tutte relazioni transitive: ad esempio, se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, allora $f(n) = O(h(n))$. È quindi facile convincersi del fatto che la notazione Θ definisca una relazione di equivalenza sulle funzioni, dato che è riflessiva, simmetrica e transitiva.

2.3 Delimitazioni inferiori e superiori

La notazione asintotica introdotta nel Paragrafo 2.2 può essere utilizzata in modo naturale per esprimere delimitazioni inferiori e superiori alla complessità di un problema rispetto ad una data risorsa di calcolo. Tipicamente, come abbiamo già osservato nel Capitolo 1, in questo libro ci occuperemo del tempo di esecuzione e dell'occupazione di memoria, ma esistono modelli di calcolo più complessi e più realistici rispetto a quelli descritti nel Paragrafo 2.1 in cui è significativo considerare anche altri tipi di risorse.

La notazione O è adatta ad esprimere *delimitazioni superiori* (in inglese, *upper bound*) sul costo di esecuzione di un algoritmo o sulla complessità di un problema, come precisato rispettivamente dalle Definizioni 2.2 e 2.3.

Definizione 2.2 *Un algoritmo $\mathcal{A}$ ha costo di esecuzione $O(f(n))$ su istanze di ingresso di dimensione n e rispetto ad una certa risorsa di calcolo, se la quantità r di risorsa* sufficiente *per eseguire $\mathcal{A}$ su una qualunque istanza di dimensione n verifica la relazione $r(n) = O(f(n))$.*

Definizione 2.3 *Un problema $\mathcal{P}$ ha una complessità $O(f(n))$ rispetto ad una data risorsa di calcolo se* esiste *un algoritmo che risolve $\mathcal{P}$ il cui costo di esecuzione rispetto a quella risorsa è $O(f(n))$.*

Per dare una delimitazione superiore alla complessità di un problema sembra quindi sufficiente progettare un algoritmo che risolva il problema e valutarne il costo di esecuzione. Ad esempio, mostreremo nel Paragrafo 4.2 del Capitolo 4 che il problema dell'ordinamento di n elementi ha complessità $O(n^2)$ rispetto al tempo di esecuzione e complessità $O(n)$ rispetto all'occupazione di memoria. In generale, per risolvere un dato problema, è però buona norma cercare di usare la minima quantità di risorsa possibile: come abbiamo già visto nel Capitolo 1, è quindi importante sforzarsi costantemente di progettare algoritmi sempre più efficienti, in modo da poter individuare la più "piccola" funzione $f(n)$ per cui si possa affermare che l'algoritmo ha costo di esecuzione $O(f(n))$ su ogni istanza di dimensione n. Ad esempio, nel Capitolo 4, mostreremo che il problema dell'ordinamento di n elementi ha complessità $O(n \log n)$ rispetto al tempo di esecuzione: poiché $O(n \log n) \subseteq O(n^2)$, questa delimitazione è certamente migliore rispetto alla delimitazione $O(n^2)$ citata precedentemente. A questo punto, sembra spontaneo chiedersi se esista un algoritmo di ordinamento con tempo di esecuzione $O(n)$ o addirittura inferiore: dopo un istante di riflessione possiamo intuire che per ordinare n elementi dobbiamo almeno esaminarli tutti, e quindi non è possibile ottenere tempi di esecuzione inferiori ad $O(n)$ per questo problema. In altre parole, abbiamo trovato una *delimitazione inferiore* alla complessità del problema dell'ordinamento. Come vedremo dalle Definizioni 2.4 e 2.5, la notazione Ω è molto adatta ad esprimere delimitazioni inferiori (in inglese, *lower bound*) sul costo di esecuzione di un algoritmo o sulla complessità di un problema.

Definizione 2.4 *Un algoritmo $\mathcal{A}$ ha costo di esecuzione $\Omega(f(n))$ su istanze di dimensione n e rispetto ad una certa risorsa di calcolo, se la massima quantità r di risorsa* necessaria *per eseguire $\mathcal{A}$ su istanze di dimensione n verifica la relazione $r(n) = \Omega(f(n))$.*

Definizione 2.5 *Un problema $\mathcal{P}$ ha una complessità $\Omega(f(n))$ rispetto ad una data risorsa di calcolo se* ogni *algoritmo che risolve $\mathcal{P}$ ha costo di esecuzione $\Omega(f(n))$ rispetto a quella risorsa.*

Intuitivamente, una delimitazione inferiore dà un'idea di quanto efficientemente possa essere risolto un problema e se ci sia la possibilità di migliorare gli algoritmi noti. I lower bound sono espressi in termini di argomentazioni matematiche che dimostrano che non si può risolvere il problema usando meno di una certa quantità di una certa risorsa di calcolo. Questo equivale a dimostrare che ogni algoritmo in un dato modello di calcolo deve richiedere una quantità minima di risorsa per la sua esecuzione. Dimostrare una delimitazione inferiore sembra quindi più difficile del trovare una delimitazione superiore: si deve infatti dimostrare che *ogni* algoritmo, e non solo gli algoritmi noti, ha costo di esecuzione superiore ad una certa quantità. Simmetricamente a quanto abbiamo visto per le delimitazioni superiori, nelle dimostrazioni di lower bound vogliamo individuare delimitazioni inferiori elevate, ovvero, la più "grande" funzione $f(n)$ per cui si possa affermare che tutti gli algoritmi hanno costo di esecuzione $\Omega(f(n))$ su istanze di dimensione n. Ad esempio, abbiamo facilmente derivato un lower bound $\Omega(n)$ per il problema dell'ordinamento, ma nel Capitolo 4 dimostreremo che il tempo di esecuzione di ogni algoritmo di ordinamento che opera esclusivamente tramite confronti tra elementi deve essere $\Omega(n \log n)$: poiché $\Omega(n \log n) \subseteq \Omega(n)$, questa delimitazione è certamente migliore rispetto alla delimitazione $\Omega(n)$ derivata precedentemente.

Sottolineiamo che la presenza di una delimitazione inferiore non indica necessariamente che algoritmi più efficienti siano impossibili, ma solo che è necessario cercarli al di fuori del modello considerato, magari esaminando più dettagliatamente il problema e le caratteristiche delle istanze. Ad esempio, vedremo nel Capitolo 4 algoritmi lineari per l'ordinamento di numeri interi: questo non è in contraddizione con il lower bound $\Omega(n \log n)$, poiché questi algoritmi si avvalgono di operazioni diverse dai confronti, sfruttando al meglio la particolare natura dei dati in ingresso.

Concludiamo questo paragrafo formalizzando il concetto di ottimalità di un algoritmo rispetto ad una data risorsa di calcolo.

Definizione 2.6 *Dato un problema $\mathcal{P}$ con complessità $\Omega(f(n))$ rispetto ad una data risorsa di calcolo, un algoritmo che risolve $\mathcal{P}$ è* ottimo *se ha costo di esecuzione $O(f(n))$ rispetto a quella risorsa.*

Osserviamo che la Definizione 2.6 tratta l'ottimalità di un algoritmo in senso asintotico, poiché nasconde nella notazione O ed Ω le costanti moltiplicative esatte dei costi di esecuzione.

2.4 Metodi di analisi

Nel Capitolo 1 abbiamo osservato che, per predire il tempo di esecuzione senza essere costretti a guardare i dettagli dell'istanza del problema, sembra una buona idea misurarlo come funzione della dimensione dell'istanza stessa. Eppure, potrebbe accadere che istanze diverse, sebbene della stessa dimensione, implichino tempi di esecuzione molto diversi: in questo paragrafo formalizzeremo questo concetto. Poiché tale aspetto non è visibile nel calcolo dei numeri di Fibonacci che abbiamo affrontato nel Capitolo 1, useremo un esempio diverso, più adeguato ai nostri scopi. In particolare, affronteremo un problema di ricerca: dati un elemento x ed un insieme S, x appartiene a S?

Esistono due varianti principali del problema di ricerca, a seconda che gli elementi in S siano ordinati o meno. Consideriamo, per esempio, l'elenco telefonico

```
algoritmo ricercaSequenziale(lista L, elem x) → booleano
1.   for each ( y ∈ L ) do
2.       if ( y = x ) then return trovato
3.   return non trovato
```

Figura 2.2 L'algoritmo per la ricerca sequenziale.

di una città e supponiamo di voler recuperare due diversi tipi di informazione: il numero telefonico associato ad un certo utente (ricerca per utente), oppure il nome dell'utente di un certo numero di telefono (ricerca per numero telefonico). La ricerca per utente corrisponde alla ricerca in un insieme ordinato, poichè gli utenti sono memorizzati nell'elenco telefonico in ordine lessicografico, e quindi ci aspettiamo di poter trovare il numero cercato molto velocemente. La ricerca per numero, invece, sembra più difficile: non potremo far altro che esaminare tutti i numeri telefonici nell'elenco, dal primo all'ultimo, fino a trovare quello desiderato e ricavare da esso il nome dell'utente. Quest'ultimo approccio è un tipico esempio di *ricerca sequenziale*. Ricerche di questo tipo sono molto frequenti: ad esempio, quando un sistema operativo deve trovare i nomi dei file in una directory, di solito si basa su una ricerca sequenziale.

Lo pseudocodice in Figura 2.2 illustra l'idea generica della ricerca sequenziale. Sono possibili molte varianti per l'algoritmo `ricercaSequenziale`, e molti dettagli sono omessi dalla Figura 2.2. Ad esempio, possiamo interrompere la ricerca quando troviamo un'occorrenza di x, oppure potremmo procedere alla ricerca di tutte le occorrenze. Potremmo rappresentare la lista tramite puntatori, oppure in un array o con altre strutture dati. Nonostante ciò, e nonostante non abbiamo informazioni sul tipo di elementi nell'insieme, possiamo comunque analizzare l'algoritmo `ricercaSequenziale`.

Essendo il confronto tra due elementi l'operazione che viene ripetuta più spesso, misureremo il tempo in termini di *confronti*. A titolo di esempio, assumiamo che la lista L contenga i primi dieci numeri di Fibonacci in maniera disordinata. Per rispondere alla domanda "$5 \in L$?" potremo fermarci non appena il valore 5 viene trovato, ma per rispondere alla domanda "$7 \in L$?" dobbiamo scandire tutta la lista, poiché 7 non è un numero di Fibonacci. Il tempo sembra quindi dipendere sia dall'elemento cercato x che dalla lista L. Poiché vorremmo predire il tempo senza dover eseguire l'algoritmo, affermare che il numero di confronti per trovare un elemento x in una lista L sia pari alla posizione di x in L non sembra molto informativo. Ci sono vari modi per ottenere una risposta più precisa, come vedremo nel prossimo paragrafo.

2.4.1 Caso peggiore, caso migliore e caso medio

L'idea consiste nell'analizzare il tempo di esecuzione su istanze pessime, istanze ottime ed istanze "tipiche", a parità di dimensione. Data un'istanza I di dimensione n, indichiamo con $tempo(I)$ il tempo di esecuzione dell'algoritmo su quella istanza.

- Analisi nel caso peggiore (*worst case*): quante operazioni eseguiamo nel caso peggiore, ovvero per le istanze di ingresso che comportano più lavoro per l'algoritmo?

$$T_{worst}(n) = \max_{\text{istanze } I \text{ di dimensione } n} \{tempo(I)\}$$

- Analisi nel caso migliore (*best case*): qual è il numero minimo di operazioni richieste dall'algoritmo se l'istanza di ingresso è particolarmente favorevole?

$$T_{best}(n) = \min_{\text{istanze } I \text{ di dimensione } n} \{tempo(I)\}$$

- Analisi nel caso medio (*average case*): qual è il tempo impiegato nel caso medio, ovvero, su istanze "tipiche" per quel particolare problema? Assumendo che ogni istanza I di dimensione n abbia probabilità $\mathcal{P}\{I\}$ di verificarsi, calcoliamo una media pesata sulle probabilità:

$$T_{avg}(n) = \sum_{\text{istanze } I \text{ di dimensione } n} \{\mathcal{P}\{I\} \cdot tempo(I)\}$$

Osserviamo innanzitutto che, rispetto all'analisi nel caso migliore che non sembra fornire molte informazioni utili, l'analisi nel caso medio è molto più significativa, ma è problematico definire quale sia esattamente la distribuzione di probabilità sulle istanze in ingresso. Dovendo fare delle ipotesi su tale distribuzione, l'accuratezza della nostra analisi dipenderà dalla validità di queste ipotesi. Inoltre, notiamo che è possibile avere un algoritmo per cui nessuna istanza richieda il tempo "medio". Ad esempio, possiamo pensare a un algoritmo che richiede 1 oppure 999 passi: la sua "media" è circa 500 se le istanze sono equiprobabili, ma nessuna istanza richiede esattamente 500 passi. In virtù delle considerazioni precedenti, l'analisi nel caso peggiore è quella che si usa di solito, poiché fornisce garanzie sulle prestazioni dell'algoritmo nel caso più sfavorevole. Inoltre, visto che il tempo nel caso medio è sicuramente non superiore a quello nel caso peggiore, possiamo usare l'analisi nel caso peggiore per avere informazioni sui limiti superiori del caso medio: questo è possibile senza dover conoscere la distribuzione dei dati di ingresso. Nel resto di questo libro useremo dunque prevalentemente l'analisi nel caso peggiore, e per brevità scriveremo $T(n)$ invece di $T_{worst}(n)$.

2.4.2 Analisi della ricerca sequenziale

Per chiarire meglio i concetti finora esposti, consideriamo l'algoritmo di ricerca sequenziale descritto in Figura 2.2. Indichiamo con n il numero di elementi nella lista L ed analizziamo il tempo di esecuzione in funzione di n rispettivamente nel caso migliore, peggiore, e medio.

Caso migliore. Nel caso migliore, la ricerca sequenziale esegue un solo confronto, trovando immediatamente x in prima posizione. Quindi $T_{best}(n) = 1$.

Caso peggiore. Il caso peggiore ha luogo quando x è all'ultimo posto, oppure quando $x \notin L$, e pertanto $T_{worst}(n) = n$.

Caso medio. L'analisi nel caso medio è in qualche modo più complessa. Sappiamo che il numero di confronti richiesti per trovare x è dato dalla posizione di x nella lista, che indicheremo per brevità con $pos(x)$. Il valore tipico di $pos(x)$ dipende dalla distribuzione dei dati in ingresso. Per acquisire familiarità con l'analisi nel caso medio, presenteremo due diverse analisi dell'algoritmo di ricerca sequenziale, corrispondenti a due ipotesi leggermente diverse sulla distribuzione delle istanze di ingresso.

Analisi 1. Un'ipotesi ragionevole sembrerebbe la seguente: se x è nella lista, può occupare qualsiasi posizione con la stessa probabilità, e quindi

$$\mathcal{P}\{pos(x) = i\} = \frac{1}{n}$$

per ogni $i \in [1, n]$. Con questa ipotesi, il numero medio di confronti $T_{avg}(n)$ è quindi dato da:

$$T_{avg}(n) = \sum_{i=1}^{n} \mathcal{P}\{pos(x) = i\} \cdot i = \sum_{i=1}^{n} \frac{1}{n} i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Se x non è nella lista, il numero di confronti richiesti è sempre n, ed è quindi il doppio del numero di confronti richiesti in media per trovare un elemento che è nella lista.

Analisi 2. Consideriamo ora un'altra distribuzione delle istanze in ingresso: assumiamo che ogni permutazione della lista sia equiprobabile e calcoliamo la media su tutte le possibili permutazioni. Poiché ci sono $n!$ permutazioni di n elementi e il numero di confronti eseguiti è pari alla posizione di x nella lista, dalla definizione di tempo medio di esecuzione si ha:

$$T_{avg}(n) = \sum_{\pi=1}^{n!} \frac{1}{n!} \cdot (\text{posizione di } x \text{ nella permutazione } \pi)$$

Come mostrato in Figura 2.3, possiamo partizionare le permutazioni in base alla posizione in cui appare l'elemento x, ottenendo n gruppi: le permutazioni in cui x è in prima posizione, oppure in seconda, o in terza, e così via. Tutti questi gruppi contengono lo stesso numero di permutazioni, essendo $(n-1)!$ i diversi modi di permutare i rimanenti $(n-1)$ elementi. Inoltre, se x occupa la posizione i, sono necessari esattamente i confronti per trovarlo. La sommatoria precedente può quindi essere riscritta come

$$T_{avg}(n) = \sum_{i=1}^{n} \frac{1}{n!} i \, (\text{numero di permutazioni con } x \text{ in posizione } i) =$$

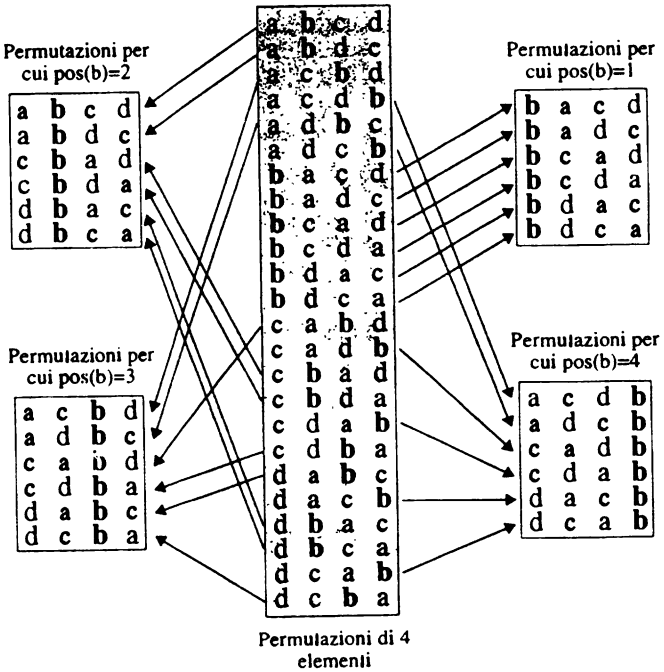


Figura 2.3 Raggruppamento delle 24 permutazioni di quattro elementi in base alla posizione dell'elemento b.

$$= \sum_{i=1}^{n} \frac{1}{n!} i \, (n-1)! = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$$

La nuova ipotesi sulla distribuzione delle istanze di ingresso restituisce pertanto esattamente lo stesso risultato di prima.

Un'ultima osservazione sull'analisi nel caso medio. Potremmo combinare i due casi in cui x appartiene o meno alla lista L nel modo seguente. Se x è presente in L con probabilità p, allora il numero di confronti in media è dato da:

$$p\left(\frac{n+1}{2}\right) + (1-p)\,n = n - \frac{p}{2}(n-1)$$

Se $p = 0$, ovvero se l'elemento cercato x non appartiene mai alla lista, ogni ricerca costa n confronti; se $p = 1$, ovvero se x appartiene sempre alla lista, ogni ricerca costa $(n+1)/2$ confronti in media. La formula precedente esprime inoltre tutte le possibili situazioni intermedie: ad esempio, se la probabilità che x appartenga alla lista è uguale alla probabilità che non vi appartenga (ovvero $p = 1/2$), in

```
algoritmo ricercaBinariaIter(array L, elem x) → booleano
    a ← 1
    b ← lunghezza di L
    while ( L[(a + b)/2] ≠ x ) do
        m ← (a + b)/2
        if ( L[m] > x ) then b ← m − 1
        else a ← m + 1
        if ( a > b ) then return non trovato
    return trovato
```

Figura 2.4 Implementazione iterativa dell'algoritmo per la ricerca binaria.

media eseguiamo circa $3n/4$ confronti, il che, come ci aspetteremmo, è un valore equidistante da n ed $(n+1)/2$.

2.4.3 Un algoritmo più veloce: la ricerca binaria

In questo paragrafo vedremo che è possibile progettare un algoritmo di ricerca più veloce dell'algoritmo `ricercaSequenziale` nel caso in cui l'insieme di elementi sia ordinato. Torniamo al nostro esempio iniziale: ricercare nell'elenco telefonico di una città il numero associato ad un certo utente x. Poiché l'elenco è in ordine alfabetico, potremmo scorrere l'elenco e fermarci quando troviamo l'utente cercato o un utente il cui nome y è successivo ad x nell'ordine lessicografico. Questo accelererebbe la ricerca dell'utente x nel caso in cui x non sia in elenco. In realtà, imitando quello che approssimativamente facciamo quando esaminiamo liste alfabetiche, possiamo fare ancora di meglio. Il procedimento è detto di *ricerca binaria* (o *dicotomica*).

Per usare la ricerca binaria, dobbiamo assumere di avere accesso diretto agli elementi in base alla loro posizione: non possiamo, ad esempio, memorizzare gli elementi in una lista rappresentata tramite puntatori, ma dobbiamo assumere che essi siano contenuti in un array. Prendiamo un elemento k in una posizione centrale dell'array e confrontiamolo con l'elemento x che stiamo cercando: se $k < x$, restringiamo la ricerca alle posizioni successive a quella di k, altrimenti procediamo nelle posizioni precedenti. Questo algoritmo si applica su elementi di qualunque tipo, purché tra essi sia definita una relazione d'ordine totale. Lo pseudocodice è mostrato in Figura 2.4. Le variabili a e b rappresentano l'estremo sinistro e destro dell'array su cui la ricerca deve proseguire ad ogni passo. Vediamo ora quanto costa effettuare la ricerca binaria su un array di n elementi rispettivamente nel caso migliore, peggiore, e medio.

Caso migliore. Il caso migliore si verifica quando $x = L[(n+1)/2]$, e pertanto $T_{best}(n) = 1$.

Caso peggiore. Nel caso peggiore, x viene trovato invece all'ultimo confronto, che avviene quando $a = b$. Per semplicità di analisi, assumiamo che n sia una

potenza di 2. Poiché m rappresenta la posizione centrale dell'array, è facile vedere che ad ogni passo la dimensione del sottoarray su cui la ricerca prosegue viene dimezzata: si parte quindi da un array di dimensione n, ci si riduce con un solo confronto ad un array di dimensione al più $n/2$, poi $n/4$ e così via. In generale, dopo i confronti, l'array delimitato da a e b ha dimensione $\leq n/2^i$: affinché risulti $a = b$, deve essere $n/2^i = 1$, ovvero $i = \lceil \log n \rceil$. Il caso peggiore della ricerca binaria richiede quindi tempo $T_{worst}(n) = O(\log n)$, e la ricerca binaria è esponenzialmente più veloce della ricerca sequenziale. L'insieme di dati in ingresso deve però essere ordinato: vedremo nel Capitolo 4 che fortunatamente esistono algoritmi molto efficienti per il problema dell'ordinamento.

Caso medio. Per semplicità di analisi, assumiamo che n sia una potenza di 2. Supponiamo che $x \in L$ (altrimenti eseguiremo sempre $\log n$ confronti) e che x possa occupare qualsiasi posizione di L con la stessa probabilità. Per definizione di tempo medio di esecuzione si ha:

$$T_{avg}(n) = \sum_{p=1}^{n} \frac{1}{n} \cdot (\text{numero confronti se } x \text{ sta in posizione } p)$$

Possiamo ora partizionare le n possibili posizioni in base al namero di confronti eseguiti dall'algoritmo `ricercaBinariaIter` per trovare un elemento x che si trovi in una specifica posizione. Esiste un solo caso in cui esegniamo un confronto: se x è proprio nella posizione centrale dell'array, ovvero $x = L[n/2]$. Esistono invece due casi in cui eseguiamo due confronti: se $x = L[n/4]$ oppure $x = L[3n/4]$. Analogamente, esistono quattro casi in cui esegniamo tre confronti: se $x = L[n/8]$, $x = L[3n/8]$, $x = L[5n/8]$, oppure $x = L[7n/8]$. Procedendo in questo modo, è facile convincersi della seguente proprietà:

Proprietà 2.2 *L'algoritmo* `ricercaBinariaIter` *esegue i confronti, con $1 \leq i \leq \log n$, se e solo se x si trova in una tra 2^{i-1} possibili posizioni dell'array L.*

In base alla Proprietà 2.2, le posizioni possono essere partizionate in $\log n$ gruppi e possiamo riscrivere il tempo medio come mostrato di seguito:

$$T_{avg}(n) = \sum_{i=1}^{\log n} \frac{1}{n}\, i\, (\text{numero di posizioni che richiedono } i \text{ confronti}) =$$

$$= \frac{1}{n} \sum_{i=1}^{\log n} i\, 2^{i-1}$$

Per calcolare il valore della sommatoria, rimandiamo all'uguaglianza (17.9) dimostrata in Appendice, ottenendo:

$$T_{avg}(n) = \frac{1}{n}\left(2^{\log n}(\log n - 1) + 1\right) = \log n - 1 + \frac{1}{n}$$

```
algoritmo ricercaBinariaRic(array L, elemento x) → booleano
1.   n ← lunghezza di L
2.   if ( n = 0 ) then return non trovato
3.   i ← ⌈n/2⌉
4.   if ( L[i] = x ) then return trovato
5.   else if ( L[i] > x ) then return ricercaBinariaRic(x,L[1; i − 1])
6.   else return ricercaBinariaRic(x,L[i + 1; n])
```

Figura 2.5 Implementazione ricorsiva dell'algoritmo per la ricerca binaria. $L[a;b]$ indica la porzione dell'array L compresa tra gli indici a e b (inclusi).

Il tempo medio si discosta quindi per meno di un confronto da quello nel caso peggiore! Questo risultato, d'altro canto, è intuitivo, se pensiamo che la metà degli elementi richiedono $\log n$ confronti, un quarto degli elementi ne richiedono $\log n - 1$, e così via.

Prima di concludere questo paragrafo, osserviamo che esiste una implementazione ricorsiva, anziché iterativa, dell'algoritmo di ricerca binaria, il cui pseudocodice è illustrato in Figura 2.5. Questa formulazione più semplice ed intuitiva, per poter essere analizzata, richiede però delle tecniche che vedremo in dettaglio nel prossimo paragrafo.

2.5 Analisi di algoritmi ricorsivi

Abbiamo visto nel Capitolo 1 che il tempo di esecuzione di algoritmi ricorsivi può essere espresso tramite *relazioni di ricorrenza*: infatti, il tempo richiesto da una procedura è uguale al tempo speso all'interno della procedura più il tempo speso per le chiamate ricorsive. Ad esempio, la relazione di ricorrenza relativa all'algoritmo di ricerca binaria è $T(n) = O(1) + T(n/2)$. Più precisamente, nel caso peggiore:

$$T(n) = 2 + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right)$$

Vedremo nel resto di questo paragrafo diversi metodi per trovare la soluzione di relazioni di ricorrenza simili a questa. I primi due metodi si applicano a relazioni di ricorrenza di qualunque tipo: a seconda della struttura della relazione stessa, potrebbero però essere di non facile applicazione. Il terzo metodo serve invece a risolvere una larga classe di relazioni di ricorrenza derivanti da algoritmi basati sulla tecnica *divide et impera*, come l'algoritmo `ricercaBinariaRic` in Figura 2.5. Questa tecnica consiste nel dividere un problema in sottoproblemi più piccoli, risolvere i sottoproblemi separatamente, e combinare le loro soluzioni per ottenere la soluzione al problema originario. Il nome *divide et impera* deriva da un motto degli antichi romani, la cui strategia di governo consisteva nel mantenere divisi i possibili nemici, evitandone la coalizione, per riuscire a dominarli più facilmente: questo celebre principio politico è stato poi adottato nel corso dei secoli da molti famosi condottieri e regnanti.

2.5.1 Metodo dell'Iterazione

Il modo più intuitivo per risolvere una relazione di ricorrenza consiste nello "srotolare" la ricorsione, riducendola ad una sommatoria dipendente solo dalla dimensione n del problema iniziale. Illustriamo questo metodo con un esempio semplice, relativo all'analisi della ricerca binaria.

Esempio 2.3 Sappiamo che la relazione di ricorrenza relativa alla ricerca binaria ha la forma

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

Assumendo per semplicità che k sia una potenza di 2, per $k = n, n/2, n/4$, rispettivamente, avremo:

$$T(n) = c + T\left(\frac{n}{2}\right) \qquad T\left(\frac{n}{2}\right) = c + T\left(\frac{n}{4}\right) \qquad T\left(\frac{n}{4}\right) = c + T\left(\frac{n}{8}\right)$$

da cui, attraverso sostituzioni successive, otteniamo:

$$T(n) = c + T\left(\frac{n}{2}\right) = 2c + T\left(\frac{n}{4}\right) = 3c + T\left(\frac{n}{8}\right) = \ldots = i \cdot c + T\left(\frac{n}{2^i}\right)$$

Quante volte dobbiamo iterare questo procedimento prima di raggiugere il passo base? Fintantoché non diventa $n/2^i = 1$, ovvero per $i = \log_2 n$ volte. Ponendo $i = \log_2 n$, possiamo quindi concludere che $T(n) = c \log n + T(1) = O(\log n)$. □

Non sempre applicare il metodo di iterazione è così semplice. Tipicamente esso richiede infatti l'uso di manipolazioni algebriche, quali regole per il cambiamento di base nei logaritmi e calcolo di limitazioni superiori a serie numeriche. Vediamo ora un esempio un po' più complesso.

Esempio 2.4 Sia data la relazione di ricorrenza

$$T(n) = \begin{cases} 9T(n/3) + n & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Per semplicità di analisi, assumeremo che n sia una potenza di 3. Srotolando la ricorsione otteniamo:

$$T(n) = 9T(n/3) + n = 9\,(9T(n/9) + n/3) + n =$$

$$= 9^2\,T(n/3^2) + 9n/3 + n = \ldots = 9^i\,T(n/3^i) + \sum_{j=0}^{i-1}(9/3)^j n$$

Osserviamo che $n/3^i = 1$ quando $i = \log_3 n$, da cui si ha:

$$T(n) = 9^{\log_3 n} + n \sum_{j=0}^{\log_3 n - 1} (9/3)^j = 9^{\log_3 n} + n \sum_{j=0}^{\log_3 n - 1} 3^j$$

Poiché $\log_3 n = \log_9 n \cdot \log_3 9$, risulta $9^{\log_3 n} = 9^{\log_9 n \cdot \log_3 9} = n^2$. Usando la serie geometrica (vedi Paragrafo 17.2 in Appendice) possiamo concludere che:

$$T(n) = n^2 + \frac{3^{\log_3 n} - 1}{3 - 1} = n^2 + \frac{n-1}{2}$$

e quindi $T(n) = \Theta(n^2)$. □

2.5.2 Metodo della sostituzione

Esiste uno stretto legame tra induzione e ricorsione, ed il metodo per risolvere relazioni di ricorrenza che presenteremo in questo paragrafo si avvale di questa proprietà. Esso consiste infatti nell'"intuire" la soluzione di una relazione di ricorrenza ed usare l'induzione matematica per dimostrare che la soluzione è effettivamente quella che si è intuita. Si tratta di un metodo molto potente quando si sia acquisita un po' di esperienza, ma, come vedremo, occorre prestare molta attenzione ad alcuni tipici trabocchetti e possibili problemi che esso presenta.

Esempio 2.5 Per illustrare il metodo di sostituzione, iniziamo con un esempio semplice:

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

Assumiamo che la soluzione sia $O(n)$: tramite l'induzione matematica cercheremo di dimostrare che $T(n) \leq c \cdot n$ per una opportuna costante $c > 0$. Poiché non conosciamo c, faremo i calcoli mantenendo c incognito. Il passo base è banalmente verificato, poiché $T(1) = 1 \leq c \cdot 1$ per ogni $c \geq 1$. Usando la relazione di ricorrenza e l'ipotesi induttiva su dimensioni $< n$, si ha:

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

Abbiamo quindi ottenuto che $T(n) \leq f(c) \cdot n$, dove $f(c) = c/2 + 1$. La dimostrazione per induzione funziona quando $f(c) \leq c$, ovvero per ogni $c \geq 2$. Ciò dimostra pertanto che $T(n) \leq 2n$. □

L'uso del metodo di sostituzione ci dà quindi anche un'informazione abbastanza precisa sulle costanti moltiplicative che sono invece nascoste nella notazione asintotica. Prima di procedere, sottolineiamo un tipico errore che si può commettere usando il metodo di sostituzione: le costanti nascoste nella notazione asintotica devono realmente essere costanti, e non qualcosa che può crescere ogni volta che si applica un passo di induzione.

Esempio 2.6 Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} n + T(n-1) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

È facile vedere per iterazione che $T(n) = \Theta(n^2)$. Domandiamoci ora: perché il metodo di sostituzione non permette di dimostrare che $T(n) = O(n)$? Assumendo per ipotesi induttiva che $T(n-1) = O(n-1)$, sembrerebbe infatti che $T(n) = n + O(n-1) = O(n)$, ma sappiamo che questo è sbagliato. Ciò sottolinea l'importanza di usare precise costanti moltiplicative: assumendo per ipotesi induttiva che $T(n-1) \leq c \cdot (n-1)$, non riusciamo infatti a dimostrare che $T(n) \leq c \cdot n$, e quindi a completare la dimostrazione. □

Esempio 2.7 Per illustrare altre sottigliezze relative all'uso del metodo di sostituzione, consideriamo infine la relazione di ricorrenza:

$$T(n) = \begin{cases} 9T(\lfloor n/3 \rfloor) + n & \text{se } n > 2 \\ 1 & \text{se } n = 1, 2 \end{cases}$$

Ipotizziamo (correttamente) che la soluzione sia $T(n) = O(n^2)$. Più precisamente, cerchiamo innanzitutto di dimostrare che $T(n) \leq c \cdot n^2$ per una opportuna costante $c > 0$ usando l'indazione matematica. Consideriamo i passi base: $T(1) = T(2) = 1 \leq c \cdot 1^2 \leq c \cdot 2^2$ per ogni $c \geq 1$, e quindi i passi base sono veri. Assamiamo ora, per ipotesi induttiva, che la disuguaglianza sia soddisfatta per ogni dimensione $k < n$, e dimostriamo che vale anche per $k = n$. Usando la relazione di ricorrenza, si ha:

$$T(n) = 9T(\lfloor n/3 \rfloor) + n \leq 9c\lfloor n/3 \rfloor^2 + n \leq 9c(n/3)^2 + n = c \cdot n^2 + n$$

Usando questa soluzione non riusciamo dunque a dimostrare la nostra affermazione, poiché $c \cdot n^2 + n > c \cdot n^2$. Poiché n ha un'ordine di grandezza inferiore rispetto a n^2, questo fatto sembra più un tecnicismo che non un reale problema. Possiamo infatti risolvere il problema avvalendoci di un'ipotesi induttiva più forte, in modo da far comparire un addendo negativo che, sommato ad n, faccia tornare i conti. Provando con l'ipotesi di soluzione $T(n) \leq c(n^2 - n)$, otteniamo infatti:

$$T(n) = 9T(\lfloor n/3 \rfloor) + n \leq 9c((n/3)^2 - n/3) + n = cn^2 - 3cn + n \leq c(n^2 - n)$$

se $-3cn + n \leq -cn$, ovvero per ogni $c \geq 1/2$. Sfortunatamente, sistemando un problema, ne abbiamo però introdotto un altro: il passo base ora non è più

vero! Infatti $T(1) = 1 > c(1^2 - 1) = 0$. Ciò può però essere facilmente risolto cambiando le condizioni agli estremi. Infatti la notazione asintotica richiede di dimostrare che $T(n) \leq c(n^2 - n)$ solo per $n \geq n_0$, il che ci permette di non considerare il caso $n = 1$. I passi base che useremo sono $n = 2, 3, 4, 5$, poiché applicando la relazione di ricorrenza per dimensioni > 5 ci si riconduce sempre a uno di questi casi. Sappiamo che $T(2) = 1$, ed è facile ottenere $T(3) = 12$, $T(4) = 13$, e $T(5) = 14$ dalla relazione di ricorrenza. Proviamo a vedere cosa accade, ad esempio, per $n = 3$: $T(3) = 12 \leq c(3^2 - 3) = 6c$ per ogni $c \geq 2$. Analogamente possiamo dimostrare che $c \geq 2$ soddisfa la disuguaglianza anche in tutti gli altri passi base, risolvendo il problema. □

2.5.3 Il teorema fondamentale delle ricorrenze

Una tecnica molto potente e generale per la progettazione di algoritmi è la tecnica *divide et impera*: ne abbiamo già vista un'applicazione nel Paragrafo 1.7 del Capitolo 1, e ne vedremo numerose altre nel corso di questo libro, ad esempio nei Capitoli 4, 5, e 10. L'idea consiste nel dividere i dati di ingresso in sottoinsiemi (*divide*), risolvere ricorsivamente il problema sui sottoinsiemi, e ricombinare infine la soluzione dei sottoproblemi per ottenere la soluzione globale (*impera*). Molte delle relazioni di ricorrenza derivanti da algoritmi basati sulla tecnica *divide et impera* possono essere facilmente risolte usando il teorema che presenteremo in questo paragrafo. Supponiamo che un problema di dimensione n venga diviso in a sottoproblemi di dimensione n/b ciascuno, e che dividere in sottoproblemi e combinare le soluzioni richieda tempo $f(n)$. La relazione di ricorrenza corrispondente a questo scenario è quindi:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{2.1}$$

Ad esempio, possiamo interpretare sia l'algoritmo ricorsivo per la ricerca binaria mostrato in Figura 2.5 che l'algoritmo `fibonacci6` descritto nel Paragrafo 1.7 del Capitolo 1 come applicazioni della tecnica *divide et impera*, con $a = 1$, $b = 2$, e $f(n) = O(1)$.

Abbiamo già visto nel Capitolo 1 che le chiamate ricorsive possono essere rappresentate tramite l'*albero della ricorsione*: la radice corrisponde alla prima chiamata, ed i figli di ogni nodo corrispondono alle chiamate ricorsive effettuate dal nodo stesso. Per analizzare la relazione di ricorrenza (2.1), consideriamo quindi l'albero della ricorsione, etichettando i nodi con la dimensione del sottoproblema corrispondente, come illustrato in Figura 2.6. Per semplicità, assumiamo che n sia una potenza di b, e che la ricorsione si fermi quando $n = 1$. Enunciamo ora alcune proprietà che saranno utili per riscrivere la relazione di ricorrenza come una formula chiusa. La dimensione del problema viene divisa per b ad ogni chiamata ricorsiva, e dipende pertanto esclusivamente dal livello nell'albero della ricorsione:

Proprietà 2.3 *I sottoproblemi al livello i dell'albero della ricorsione hanno dimensione n/b^i.*

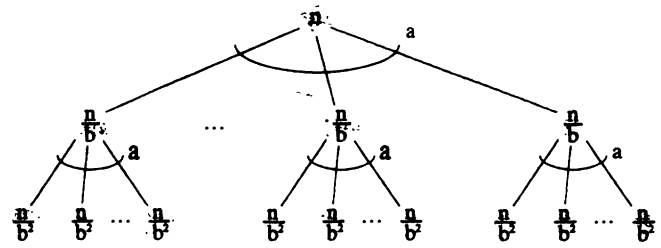


Figura 2.6 Albero della ricorsione corrispondente a relazioni di ricorrenza derivanti da algoritmi di tipo *divide et impera*.

Dalla relazione di ricorrenza (2.1) e dalla Proprietà 2.3 si ottiene quindi:

Proprietà 2.4 *Il contributo di un nodo a livello i al tempo di esecuzione (escludendo il tempo speso nelle chiamate ricorsive) è $f(n/b^i)$.*

Il fatto che i sottoproblemi nell'ultimo livello abbiano dimensione $n/b^i = 1$ implica $n = b^i$ e quindi $i = \log_b n$. Questo valore rappresenta una limitazione all'altezza dell'albero:

Proprietà 2.5 *Il numero di livelli nell'albero della ricorsione è $\log_b n$.*

Poiché ciascun nodo interno ha esattamente a figli, abbiamo anche una limitazione al numero di nodi su ciascun livello:

Proprietà 2.6 *Il numero di nodi al livello i dell'albero della ricorsione è a^i.*

Mettendo insieme le Proprietà 2.3 - 2.6, possiamo riscrivere la relazione di ricorrenza nella forma seguente:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right) \tag{2.2}$$

La soluzione dell'Equazione (2.2) è data dal seguente teorema, noto come *teorema fondamentale delle ricorrenze* o anche come *teorema master*, che si può dimostrare con alcune semplici manipolazioni algebriche.

Teorema 2.1 (Teorema fondamentale delle ricorrenze) *La relazione di ricorrenza*

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

ha soluzione:

1. $T(n) = \Theta(n^{\log_b a})$, *se* $f(n) = O(n^{\log_b a - \epsilon})$ *per* $\epsilon > 0$;

2. $T(n) = \Theta(n^{\log_b a} \log n)$, *se* $f(n) = \Theta(n^{\log_b a})$;
3. $T(n) = \Theta(f(n))$, *se* $f(n) = \Omega(n^{\log_b a+\epsilon})$ *per* $\epsilon > 0$ *e* $a \cdot f(n/b) \leq c \cdot f(n)$ *per* $c < 1$ *ed* n *sufficientemente grande.*

Dimostrazione. Assumiamo per semplicità che n sia una potenza esatta di b: la dimostrazione può essere estesa anche agli altri valori di n ragionando con le parti intere inferiori e superiori, ma ciò introdurrebbe molti dettagli tecnici non rilevanti. Ci limiteremo quindi a dimostrare separatamente i tre casi sotto l'ipotesi che n/b^i sia sempre un numero intero.

Caso 1: $f(n) = O(n^{\log_b a-\epsilon})$, per $\epsilon > 0$. Riscriviamo in modo più conveniente il generico termine della sommatoria (2.2):

$$a^i f\left(\frac{n}{b^i}\right) = O\left(a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}\right) =$$

$$= O\left(n^{\log_b a-\epsilon}\left(\frac{a\, b^\epsilon}{b^{\log_b a}}\right)^i\right) = O(n^{\log_b a-\epsilon}(b^\epsilon)^i)$$

Per limitare $T(n)$ possiamo quindi scrivere:

$$T(n) = \sum_{i=0}^{\log_b n} O(n^{\log_b a-\epsilon}(b^\epsilon)^i) = O\left(n^{\log_b a-\epsilon}\sum_{i=0}^{\log_b n}(b^\epsilon)^i\right) =$$

$$= O\left(n^{\log_b a-\epsilon}\left(\frac{b^{\epsilon(\log_b n+1)}-1}{b^\epsilon-1}\right)\right) = O\left(n^{\log_b a-\epsilon}\left(\frac{b^\epsilon n^\epsilon-1}{b^\epsilon-1}\right)\right) =$$

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

Analizzando l'Equazione (2.2) ed isolando i tempi di esecuzione relativi ai soli nodi sull'ultimo livello dell'albero, otteniamo:

$$T(n) \geq a^{\log_b n} = n^{\log_b a}$$

e quindi $T(n) = \Omega(n^{\log_b a})$. Le due limitazioni implicano che $T(n) = \Theta(n^{\log_b a})$.

Caso 2: $f(n) = \Theta(n^{\log_b a})$. Come nel caso 1, riscriviamo in modo più conveniente il generico termine della sommatoria (2.2):

$$a^i f\left(\frac{n}{b^i}\right) = \Theta\left(a^i\left(\frac{n}{b^i}\right)^{\log_b a}\right) = \Theta\left(n^{\log_b a}\left(\frac{a}{b^{\log_b a}}\right)^i\right) = \Theta(n^{\log_b a})$$

da cui:

$$T(n) = \sum_{i=0}^{\log_b n}\Theta(n^{\log_b a}) = \Theta(n^{\log_b a}\log_b n) = \Theta(n^{\log_b a}\log n)$$

Caso 3: è facile dimostrare che, sotto l'assunzione $a\,f(n/b) \leq c\,f(n)$, risulta $a^i f(n/b^i) \leq c^i f(n)$. Infatti si ha:

$$a^i f\left(\frac{n}{b^i}\right) = a^{i-1}\, a\, f\left(\frac{n/b^{i-1}}{b}\right) \leq a^{i-1}\, c\, f\left(\frac{n}{b^{i-1}}\right)$$

Iterando il ragionamento si ottiene la disuguaglianza desiderata. Per limitare $T(n)$, usando l'Equazione (2.2) e la serie geometrica con base $c < 1$, possiamo quindi scrivere:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right) \leq \sum_{i=0}^{\log_b n} c^i f(n) \leq f(n)\sum_{i=0}^{\infty} c^i = f(n)\frac{1}{1-c} = O(f(n))$$

Dalla relazione di ricorrenza, si ha inoltre che $T(n) = \Omega(f(n))$, da cui $T(n) = \Theta(f(n))$. Osserviamo che nella dimostrazione del caso 3 non abbiamo usato l'ipotesi $f(n) = \Omega(n^{\log_b a+\epsilon})$: questa condizione è infatti implicata dall'assunzione $a\,f(n/b) \leq c\,f(n)$, come si richiede di dimostrare nel Problema 2.6, ed è riportata nell'enunciato del teorema solo per simmetria con gli altri casi. □

Intuitivamente, la sommatoria nell'Equazione (2.2) risulta dominata dai termini ai due estremi: $O(f(n))$ oppure $O(n^{\log_b a})$. Quest'ultimo deriva dal fatto che almeno tutti i nodi dell'albero della ricorsione devono essere visitati, ed il loro numero è proporzionale ad $a^{\log_b n} = a^{\log_a n \log_b a} = n^{\log_b a}$. Per semplicità, potremmo quindi riformulare il teorema fondamentale dicendo che $T(n) = \Theta(n^{\log_b a}\log n)$ se $f(n) = \Theta(n^{\log_b a})$, e $T(n) = \Theta(f(n) + n^{\log_b a})$ altrimenti. Mostriamo ora alcune applicazioni del teorema fondamentale delle ricorrenze.

Esempio 2.8 Nel caso della relazione di ricorrenza della ricerca binaria, $T(n) = T(n/2) + O(1)$, si ha $f(n) = O(1) = \Theta(n^{\log_b a})$ poiché $a = 1$ e $b = 2$. Siamo quindi nel secondo caso del teorema fondamentale, da cui otteniamo immediatamente che $T(n) = \Theta(\log n)$. □

Esempio 2.9 Nel caso della relazione di ricorrenza $T(n) = 9T(n/3) + n$, si ha invece $a = 9$ e $b = 3$, da cui $n^{\log_b a} = n^2$ e quindi $f(n) = n = O(n^{2-\epsilon})$, ad esempio per $\epsilon = 1$. Dal primo caso del teorema fondamentale delle ricorrenze abbiamo quindi che $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$. □

Esempio 2.10 Consideriamo infine la relazione di ricorrenza $T(n) = 3T(n/9) + n$, da cui $a = 3$, $b = 9$, e quindi $n^{\log_b a} = \sqrt{n}$. Da ciò deriva $f(n) = n = \Omega(n^{1/2+\epsilon})$, ad esempio per $\epsilon = 1/2$. Per poter applicare il terzo caso del teorema; dobbiamo però verificare ancora che $a \cdot f(n/b) \leq c \cdot f(n)$ per $c < 1$ e n sufficientemente grande: ciò è banalmente verificato per $c = 1/3$, poiché $3(n/9) \leq c \cdot n$. Dal teorema fondamentale delle ricorrenze otteniamo quindi $T(n) = \Theta(n)$. □

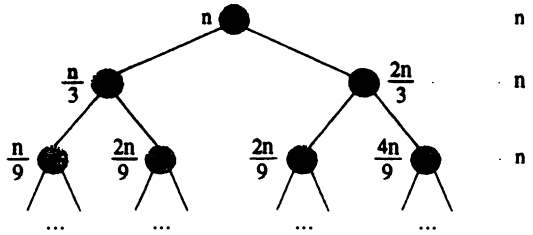


Figura 2.7 Albero della ricorsione corrispondente alla relazione di ricorrenza $T(n) = T(n/3) + T(2n/3) + n$.

Esempio 2.11 Ci sono casi in cui nessuna delle tre precedenti soluzioni può applicarsi, anche se la relazione di ricorrenza sembrerebbe avere la struttura appropriata: considerando, ad esempio, la relazione $T(n) = 3T(n/3) + n \log n$, è facile vedere che nessuno dei tre casi si applica. A prima vista, si potrebbe cercare di utilizzare il terzo caso, ma questo è impossibile perché $n \log n$ è solo logaritmicamente, e non polinomialmente, più grande di $n^{\log_b a} = n$. □

2.5.4 Altre tecniche utili

Illustreremo ora altre due tecniche che possono risultare utili nella soluzione di relazioni di ricorrenza. La prima si basa sull'analisi dell'albero della ricorsione, che abbiamo già usato nel Capitolo 1 e nell'analisi del teorema master, l'altra su cambiamenti di variabile.

Analisi dell'albero della ricorsione. Consideriamo la relazione di ricorrenza $T(n) = T(n/3)+T(2n/3)+n$. È facile vedere che essa non ha la forma per poter applicare il teorema fondamentale, ed anche risolverla per iterazione è abbastanza complesso: basta provare ad eseguire le prime iterazioni per vedere cosa accade! Ci sono comunque due metodi abbastanza convenienti per risolverla. Potremmo ad esempio usare il metodo di sostituzione: ma cosa fare se non abbiamo idea di quale sia la soluzione? Un altro metodo consiste nel disegnare l'albero delle chiamate ricorsive, indicando le dimensioni dei problemi di ogni chiamata ricorsiva, ed analizzando la dimensione totale dei problemi ad ogni livello dell'albero. L'albero della ricorsione relativo al nostro esempio è mostrato in Figura 2.7.

Ad ogni livello dell'albero spendiamo esattamente n. È inoltre facile vedere che il ramo destro dell'albero è quello più lungo: poiché risulta $n(2/3)^i = 1$ per $i = \log_{3/2} n$, l'albero ha altezza $\log_{3/2} n$ e la soluzione della ricorrenza è $O(n \log n)$. Vedremo nel Capitolo 5 una generalizzazione di questo approccio.

Cambiamenti di variabile. Consideriamo la seguente relazione di ricorrenza $T(n) = T(\sqrt{n}) + O(1)$. Anche se appare diversa da quelle viste finora, un semplice cambiamento di variabili può ricondurci ad una forma più familiare. Poniamo $n = 2^x$ (ovvero $x = \log n$), da cui $\sqrt{n} = 2^{x/2}$. Da ciò $T(2^x) = T(2^{x/2}) + O(1)$. Rinominando $T(2^x) = R(x)$, otteniamo la relazione di ricorrenza $R(x) = R(x/2) + O(1)$, che sappiamo risolvere con tecniche standard, ottenendo $R(x) = O(\log x)$. Combinando le uguaglianze $T(2^x) = O(\log x)$ e $n = 2^x$, si ha infine $T(n) = O(\log \log n)$.

2.6 Analisi di algoritmi randomizzati

A volte è utile pagare qualcosa per ridurre l'incertezza della vita: ad esempio, se assicuriamo la nostra automobile per incendio e furto, pagheremo annualmente una certa quota. Questo pagamento sarà inutile nel caso in cui non subiamo né un furto né un incendio (infatti avremmo potuto non pagare niente), ma ci permetterà di avere un rimborso dalla compagnia assicurativa nella sfortunata circostanza in cui la nostra macchina venga rubata o subisca un incendio. Lo stesso concetto si applica agli algoritmi: se il costo nel caso peggiore è molto più elevato di quello nel caso medio, potremmo preferire avere un algoritmo leggermente meno efficiente che riduca il costo del caso peggiore fintantoché non faccia crescere troppo quello del caso medio. Per esempio, se stiamo programmando un'applicazione per il controllo di un'automobile che decida quando attivare l'air bag, non vorremmo eseguire un algoritmo che tipicamente richiede un centesimo di secondo, ma a volte potrebbe richiedere ben cinque minuti: sarebbe preferibile avere un algoritmo che richieda sempre, ad esempio, un decimo di secondo, anche nel caso peggiore.

I numeri casuali (*random*, in inglese) sono molto utili a questo scopo. Sono utili in particolare a far valere l'analisi del caso medio anche quando l'istanza non è necessariamente casuale, o quando non conosciamo accuratamente la distribuzione dei dati in ingresso. L'idea è di manipolare l'istanza in modo tale che appaia come un'istanza casuale. Diciamo che un algoritmo è *randomizzato* se usa numeri casuali. Un algoritmo non randomizzato è invece detto *deterministico*. Il valore atteso del tempo di esecuzione di un algoritmo randomizzato, che chiameremo *tempo atteso*, è misurato come il tempo richiesto su una certa istanza e per una data sequenza di numeri casuali. Per una data istanza I, il tempo atteso dell'algoritmo è la media calcolata sulle diverse sequenze di numeri casuali:

$$T_{exp}(I) = \sum_{\text{sequenze random } R} \{\mathcal{P}\{R\} \cdot tempo(I, R)\}$$

Il tempo atteso dell'algoritmo nel caso peggiore su istanze di lunghezza n è quindi calcolato combinando questa formula con la formula per l'analisi nel caso peggiore introdotta nel Paragrafo 2.4.1:

$$T_{exp}(n) = \max_{\text{istanze } I \text{ di dimensione } n} \{T_{exp}(I)\} =$$

$$= \max_{\text{istanze } I \text{ di dimensione } n} \sum_{\text{sequenze random } R} \{\mathcal{P}\{R\} \cdot tempo(I, R)\}$$

```
algoritmo ricercaRandomizzata(lista L, elem x) → booleano
    permuta casualmente L
    for each ( y ∈ L ) do
        if ( y = x ) then return trovato
    return non trovato
```

Figura 2.8 Un algoritmo randomizzato per la ricerca sequenziale.

Sebbene la definizione sembri complicata, l'analisi nel caso atteso non è di solito più difficile dell'analisi nel caso medio. Come esempio, studieremo una versione randomizzata della ricerca sequenziale.

Esempio 2.12 L'algoritmo `ricercaRandomizzata` mostrato in Figura 2.8, prima di eseguire la ricerca sequenziale, calcola una permutazione casuale della lista L, in modo che l'elemento x appaia in una posizione casuale nella permutazione. Questo rallenta leggermente l'algoritmo, perché si spende del tempo per calcolare la permutazione, ma potrebbe rendere più veloce la ricerca. Se stiamo cercando un numero in una lista, probabilmente usare la ricerca randomizzata non sarebbe una buona idea, perché il tempo per calcolare la permutazione sarebbe probabilmente maggiore del tempo richiesto dall'algoritmo deterministico di ricerca sequenziale. Ma se i confronti sono molto lenti (ad esempio, se stiamo confrontando immagini e stringhe di testo molto lunghe), si potrebbero ottenere dei miglioramenti sostanziali, poiché il tempo per i confronti dominerebbe di gran lunga il tempo di esecuzione totale. Analizziamo ora il tempo atteso dell'algoritmo `ricercaRandomizzata`:

$$T_{exp}(n) = \max_{x,L} \left\{ \sum_{\text{perm } \pi} \mathcal{P}\{\pi\}\, T(x, \pi(L)) \right\}$$

dove la lista L contiene n elementi, $\pi(L)$ denota la lista L permutata secondo π, $\mathcal{P}\{\pi\}$ è la probabilità della permutazione π, e $T(x, \pi(L))$ è il numero di confronti per trovare x in $\pi(L)$. Partizionando le permutazioni di L in base alla posizione che vi occupa l'elemento x (come abbiamo fatto per l'analisi nel caso medio dell'algoritmo `ricercaSequenziale` nel Paragrafo 2.4.2), possiamo riscrivere la sommatoria come segue:

$$\sum_{\text{perm } \pi} \mathcal{P}\{\pi\}\, T(x, \pi(L)) = \sum_{p=1}^{n} \sum_{\pi:\, pos(x)=p \text{ in } \pi(L)} \mathcal{P}\{\pi\}\, T(x, \pi(L))$$

dove $pos(x)$ denota la posizione di x. Poiché ci sono $n!$ permutazioni, $\mathcal{P}\{\pi\} = 1/n!$. Inoltre, tra tutte le permutazioni esattamente $(n-1)!$ hanno x in una data posizione p, e per queste permutazioni $T(x, \pi(L)) = p$. Con ulteriori manipolazioni

algebriche otteniamo quindi:

$$\sum_{p=1}^{n} |\{\pi : pos(x) = p \text{ in } \pi(L)\}|\, \frac{1}{n!}\, p = \sum_{p=1}^{n} (n-1)!\, \frac{1}{n!}\, p =$$

$$= \sum_{p=1}^{n} \frac{1}{n}\, p = \frac{n+1}{2}$$

da cui $T_{exp}(n) = (n+1)/2$. Il numero di confronti è quindi esattamente lo stesso dell'analisi nel caso medio, indipendentemente dalla distribuzione delle istanze di ingresso. □

Vedremo nel corso di questo libro alcuni importanti usi dell'analisi randomizzata: ad esempio, nell'algoritmo di ordinamento detto `quickSort`, che presenteremo nel Paragrafo 4.5 del Capitolo 4, la randomizzazione fa decrescere il tempo di esecuzione da $O(n^2)$ a $O(n \log n)$. È anche possibile eseguire analisi più accurate di algoritmi randomizzati, ad esempio calcolando la varianza dei tempi di esecuzione o dimostrando affermazioni del tipo: "con probabilità molto alta, l'algoritmo richiede tempo vicino al suo tempo atteso". Ciò è importante se ci si vuole assicurare che le esecuzioni lente siano molto rare. Di solito queste forme di analisi sono più complesse, e non ne incontreremo in questo libro.

2.7 * Analisi ammortizzata

In molte situazioni pratiche, ci troviamo ad eseguire un algoritmo ripetutamente nel tempo. Ad esempio, i gestori di motori di ricerca per Internet vedono arrivare miliardi di interrogazioni al giorno, ognuna delle quali corrisponde all'esecuzione di un certo algoritmo di ricerca. In contesti di questo tipo, abbiamo tipicamente a che fare con collezioni di dati soggetti a sequenze di interrogazioni ed aggiornamenti. Useremo il termine *operazione di aggiornamento* per indicare una operazione che modifica il contenuto informativo di una collezione di dati. Ad esempio, il processamento di una richiesta di ricarica di un cellulare fatta tramite il sito Web della nostra banca coinvolge certamente un'operazione di aggiornamento dei nostri dati bancari. Parleremo invece di *operazione di interrogazione* per denotare una operazione che esplora i dati di una collezione senza modificarne il contenuto informativo. I capitoli 3, 6, 7, 8 e 9 saranno dedicati allo studio di problemi di questo tipo. L'*analisi ammortizzata* studia le prestazioni medie di una sequenza di operazioni su una collezione di dati, piuttosto che della singola esecuzione di un algoritmo come abbiamo visto finora in questo capitolo. La differenza principale rispetto all'analisi nel caso medio vista nel Paragrafo 2.4.1 è che nell'analisi ammortizzata le probabilità non entrano in gioco in nessun modo. Studiare il comportamento di un algoritmo su una sequenza di esecuzioni può darne una caratterizzazione più raffinata. Ad esempio, una singola esecuzione di

```
algoritmo incrementaContatore(array v, intero n)
1.   for i = 0 to n − 1 do
2.       v[i] ← not v[i]
3.       if ( v[i] = 1 ) then break
```

Figura 2.9 Algoritmo di incremento di un contatore binario a n bit.

un algoritmo potrebbe richiedere tempo $\Theta(n)$ nel caso peggiore, mentre n esecuzioni potrebbero richiedere tempo totale ancora $\Theta(n)$ nel caso peggiore, e non $\Theta(n^2)$ come potremmo pensare. Sembra paradossale? Il punto è che una singola esecuzione potrebbe richiedere molto tempo per via di una configurazione particolarmente svantaggiosa dei dati su cui opera. Tuttavia, i dati stessi potrebbero configurarsi in modo molto più vantaggioso durante le esecuzioni successive, per cui in media il tempo richiesto per ciascuna esecuzione potrebbe essere di fatto molto più basso di quello che si avrebbe nel singolo caso peggiore.

Per caratterizzare le prestazioni di un algoritmo in media su una sequenza di operazioni, parleremo di *costo ammortizzato*.

Definizione 2.7 (Costo ammortizzato) *Definiamo il* costo ammortizzato *di un algoritmo su una sequenza di k operazioni come:*

$$T_a(n) = \frac{T(n,k)}{k}$$

dove $T(n,k)$ è il tempo totale richiesto dall'algoritmo nel caso peggiore per tutte le k operazioni su istanze di dimensione n.

Esempio 2.13 Consideriamo l'operazione illustrata in Figura 2.9, che ad ogni esecuzione incrementa di uno il valore di un contatore binario a n bit rappresentato mediante un array v. Nell'array, la posizione 0 contiene la cifra meno significativa del contatore. L'algoritmo scorre le celle dell'array (riga 1) invertendone il contenuto (riga 2), e si ferma non appena è stato invertito il primo zero

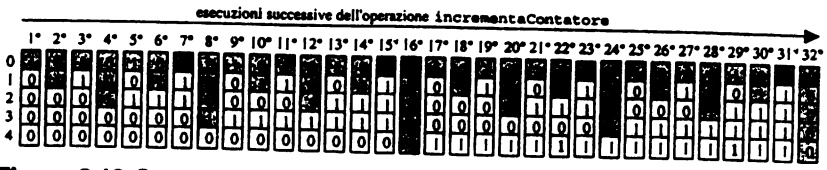


Figura 2.10 Successive configurazioni di un contatore binario a 5 bit ottenute a partire dal numero zero mediante esecuzioni successive dell'operazione `incrementaContatore` di Figura 2.9. Le celle dell'array scandite dal ciclo for ad ogni esecuzione sono evidenziate in grigio e mostrano sempre il contenuto dell'array dopo l'esecuzione dell'algoritmo.

(riga 3). Si noti che ad ogni esecuzione soltanto uno 0 al più viene trasformato in 1, mentre molti 1 potrebbero essere portati a 0. È facile vedere che, se l'array non contiene alcuno 0, l'algoritmo inverte il contenuto di tutte le celle. L'operazione `incrementaContatore` richiede quindi tempo $O(n)$ nel caso peggiore. Analizziamo ora il numero totale di passi effettuati dall'algoritmo su n esecuzioni successive. Questo numero coincide con il numero totale di bit invertiti. In Figura 2.10 è mostrato un esempio per $n = 5$ e $k = 2^5 = 32$ esecuzioni. Osserviamo che il primo bit si inverte ad ogni esecuzione, il secondo ogni 2, il terzo ogni 4, ecc. Il numero totale di inversioni è pertanto $32 + 16 + 8 + 4 + 2 = 62$. In media, il numero di inversioni per esecuzione è quindi $62/32$, cioè meno di 2. La cosa sorprendente è che questo rimane vero per qualsiasi n e per qualsiasi numero di esecuzioni, assumendo di partire da un contatore zero. Usando la serie geometrica (vedi l'Equazione (17.7) in Appendice), si ha:

$$T(n,k) = k + \left\lfloor \frac{k}{2} \right\rfloor + \left\lfloor \frac{k}{4} \right\rfloor + \cdots + 2 + 1 = \sum_{i=0}^{\lfloor \log_2 k \rfloor} \left\lfloor \frac{k}{2^i} \right\rfloor \leq k \sum_{i=0}^{\infty} \frac{1}{2^i} = 2k.$$

Questo significa che il costo ammortizzato per esecuzione è $T_a(n) = 2k/k = 2 = O(1)$, mentre il costo nel caso peggiore è $T(n) = O(n)$. Si noti che, diversamente dall'analisi nel caso medio, non stiamo facendo nessun ragionamento basato sulla probabilità che una cella contenga 1 o 0 all'inizio di una esecuzione. □

L'analisi ammortizzata permette di studiare scenari ancora più articolati in cui abbiamo a che fare con sequenze miste di operazioni di tipo diverso sugli stessi dati di interesse, come nel seguente esempio.

Esempio 2.14 Consideriamo le seguenti due operazioni su un insieme S di cardinalità n:

- `insert`(S, x): aggiunge l'elemento x all'insieme S;
- `delete`(S, q): rimuove $q \leq n$ elementi da S (il criterio per scegliere gli elementi da rimuovere non è rilevante per il nostro esempio).

Assumiamo che S sia rappresentato in modo tale che l'aggiunta o la rimozione di ciascun elemento richieda un singolo passo di esecuzione, ad esempio usando una pila o una coda. Con un'analisi nel caso peggiore, `insert` richiede tempo costante, mentre `delete` richiede tempo $O(q) = O(n)$. Un ragionamento più attento ci suggerisce però che non si può rimuovere un elemento se prima non lo si è inserito: siccome gli inserimenti avvengono uno alla volta, ad ogni `delete` che rimuove q elementi devono corrispondere necessariamente q inserimenti precedenti. Ogni elemento che finisce in S richiede un passo per l'inserimento e un passo per l'eventuale rimozione: se partiamo quindi da un insieme S vuoto, ogni sequenza mista di k_1 `insert` e k_2 `delete` richiederà al più $T(n, k_1 + k_2) \leq 2k_1$ passi in totale. Ciascuna operazione avrà pertanto costo ammortizzato:

$$T_a(n) = \frac{2k_1}{k_1 + k_2} = O(1)$$

Si noti che esso è indipendente dal numero di elementi n che si hanno in ogni momento nell'insieme S. □

Vi sono diversi metodi per calcolare il costo ammortizzato di un'operazione: fra quelli più noti, vedremo il *metodo dei crediti* e il *metodo del potenziale*. La scelta di un metodo piuttosto che un altro è una questione di convenienza rispetto al particolare algoritmo che si sta considerando.

2.7.1 Il metodo dei crediti

Come abbiamo visto nell'Esempio 2.14, in alcuni casi può accadere che il costo di un'operazione dipenda strettamente dall'esecuzione nel passato di altre operazioni: una `delete` può solo rimuovere elementi precedentemente inseriti da attrettante `insert`. Il *metodo dei crediti* permette di calcolare il costo ammortizzato di una operazione senza dover entrare nel dettaglio di queste dipendenze, che alle volte possono essere molto difficili da determinare con precisione.

Per eliminare lo studio delle dipendenze dirette tra operazioni, l'idea è quella di associare delle *monete immaginarie* (o *crediti*) agli oggetti di una collezione. Queste monete, depositate da alcune operazioni, saranno in grado di "pagare" il costo di altre operazioni. Più precisamente, assumeremo che:

1 moneta vale $O(1)$ passi di calcolo[1].

È importante notare che *l'uso delle monete serve solo per l'analisi*: gli algoritmi che analizziamo non ne hanno bisogno per funzionare!

Definizione 2.8 (Metodo dei crediti) *Il costo ammortizzato di una operazione secondo il metodo dei crediti è definito come:*

$$T_a(n) = T(n) + deposito(n) - prelievo(n),$$

dove $T(n)$ è il numero di passi effettivo che l'operazione compie nel caso peggiore, $deposito(n)$ è il numero di monete che deposita sugli oggetti, e $prelievo(n)$ è il numero di quelle che preleva dagli oggetti.

In generale, associeremo agli oggetti zero o più monete in base a qualche loro proprietà, e poi definiremo *prelievo* e *deposito* in modo che questa associazione venga rispettata tenendo conto di come gli oggetti stessi sono manipolati dalle operazioni. I seguenti esempi dovrebbero chiarire questi aspetti.

Esempio 2.15 Usiamo il metodo dei crediti per calcolare nuovamente il costo ammortizzato dell'algoritmo `incrementaContatore` discusso nell'Esempio 2.13. A questo scopo, associamo una moneta ad ogni cella dell'array v che contiene

[1] Ricordiamo qui il celebre detto di Paperon de' Paperoni: "Il tempo è denaro".

t, mentre non associamo alcuna moneta alle celle che contengono 0. Siccome ogni esecuzione dell'algoritmo trasforma al più uno 0 in 1, allora deve essere necessariamente $deposito(n) = 1$. Tutti i passi dell'algoritmo, tranne al più l'ultimo, pongono a 0 una qualche cella che era a 1: ma allora potremmo pagare questo costo interamente prelevando le monete dagli 1 che vengono portati a 0. Avremo quindi $T(n) - prelievo(n) \leq 1$, il che ci consente di scrivere: $T_a(n) = T(n) + deposito(n) - prelievo(n) \leq 2 = O(1)$. □

Esempio 2.16 Riprendiamo l'Esempio 2.14 e ricalcoliamo il costo ammortizzato di `insert` e `delete` usando il metodo dei crediti. A questo scopo, associamo ad ogni elemento dell'insieme S una moneta immaginaria. Per `insert`, abbiamo che il costo effettivo dell'operazione è $T(n) = 1$, $deposito(n) = 1$ per dotare l'elemento aggiunto della moneta immaginaria, e $prelievo(n) = 0$ perché non abbiamo bisogno di prelevare monete: abbiamo quindi $T_a(n) = T(n) + deposito(n) - prelievo(n) = 1 + 1 - 0 = 2 = O(1)$. Per `delete`, $deposito(n) = 0$ e possiamo pagare il costo della rimozione degli elementi con le monete che troviamo depositate sugli elementi stessi, per cui $T(n) = prelievo(n)$ e dunque $T_a(n) = T(n) + deposito(n) - prelievo(n) = 0 = O(1)$. □

Vedremo ulteriori esempi di uso del metodo dei crediti nei Capitoli 3, 8 e 9.

2.7.2 Il metodo del potenziale

Il *metodo del potenziale* che vedremo in questo paragrafo è concettualmente simile a quello dei crediti. La differenza principale è che, invece di associare crediti ai singoli oggetti, associamo piuttosto un *potenziale* ai dati nel loro insieme (un numero non negativo). L'associazione avviene tramite una *funzione potenziale* Φ che, applicata ad una istanza I della struttura dati, definisce il potenziale $\Phi(I)$ dell'istanza. Analogamente ai crediti visti nel Paragrafo 2.7.1, il potenziale di una collezione di oggetti può servire a pagare il costo di certe operazioni sulla collezione stessa. Il concetto è del tutto analogo a quello di un corpo in un campo elettrico o gravitazionale a cui è associata una energia potenziale che può essere trasformata in lavoro.

Definizione 2.9 (Metodo del potenziale) *Il costo ammortizzato di una operazione secondo il metodo del potenziale è definito come:*

$$T_a(n) = T(n) + \Phi(I') - \Phi(I),$$

dove $T(n)$ è il numero di passi effettivo che l'operazione compie nel caso peggiore, I è l'istanza di dimensione n della struttura dati in ingresso all'operazione ed I' è l'istanza della struttura dati dopo l'esecuzione dell'operazione.

Mostriamo ora che sotto opportune condizioni, la somma dei costi ammortizzati secondo il metodo del potenziale fornisce una delimitazione superiore al tempo totale richiesto nel caso peggiore dall'intera sequenza:

Teorema 2.2 *Siano $I_0, \ldots, I_k$ le istanze di dimensione n della struttura dati ottenute su una sequenza di k operazioni a partire da un'instanza iniziale I_0. Se $\Phi(I_0) = 0$ e $\Phi(I_k) \geq 0$, allora*

$$T(n,k) \leq \sum_{i=l}^{k} T_a(n)$$

dove $T(n,k)$ è il tempo totale richiesto dall'algoritmo nel caso peggiore per tutte le k operazioni su istanze di dimensione n.

Dimostrazione. Siano $I_0, \ldots, I_k$ le istanze di dimensione n ottenute su una sequenza di k operazioni a partire da un'istanza iniziale I_0. Assumendo che l'operazione i-esima richieda tempo $T_i(n)$, con $1 \leq i \leq k$, e sommando su tutte le operazioni possiamo scrivere:

$$\sum_{i=l}^{k} T_a(n) = \sum_{i=l}^{k} [T_i(n) + \Phi(I_i) - \Phi(I_{i-1})] = \sum_{i=l}^{k} [T_i(n)] + \Phi(I_k) - \Phi(I_0)$$

Infatti, $\Phi(I_l) - \Phi(I_0) + \Phi(I_2) - \Phi(I_1) + \cdots + \Phi(I_k) - \Phi(I_{k-1}) = \Phi(I_k) - \Phi(I_0)$ essendo una somma telescopica. Quindi, poiché $\Phi(I_0) = 0$ e $T(n,k) = \sum_{i=l}^{k} [T_i(n)]$, si ha:

$$T(n,k) = \sum_{i=l}^{k} T_a(n) - \Phi(I_k) \leq \sum_{i=l}^{k} T_a(n)$$

essendo per ipotesi $\Phi(I_k) \geq 0$. □

Se la funzione potenziale è scelta in modo opportuno, la Definizione 2.9 fornisce una misura di costo più raffinata dell'analisi nel caso peggiore, e il Teorema 2.2 ci garantisce che questa analisi è accurata.

Esempio 2.17 Usiamo il metodo del potenziale per calcolare ancora una volta il costo ammortizzato dell'algoritmo `incrementaContatore` discusso nell'Esempio 2.13. A questo scopo, definiamo:

$$\Phi(v) = \text{numero di bit a 1 nell'array } v.$$

Denotiamo poi con v_0 l'array iniziale e in generale con v_i l'array prodotto dall'operazione i-esima, con $1 \leq i \leq k$. È facile vedere che nel caso peggiore l'esecuzione i-esima determina una variazione di potenziale pari a $\Phi(v_i) - \Phi(v_{i-1}) = -z_i + 1$, dove z_i è il numero di bit portati a zero darante tale esecuzione. Inoltre, il tempo effettivo richiesto da tale esecuzione è $T_i(n) \leq z_i + 1$. Quindi, in base alla Definizione 2.9 otteniamo:

$$T_a(n) = T_i(n) + \Phi(v_i) - \Phi(v_{i-1}) \leq z_i + 1 - z_i + 1 = 2 = O(1).$$

Si noti che otteniamo con passaggi diversi lo stesso risultato che avevamo calcolato con il metodo dei crediti. □

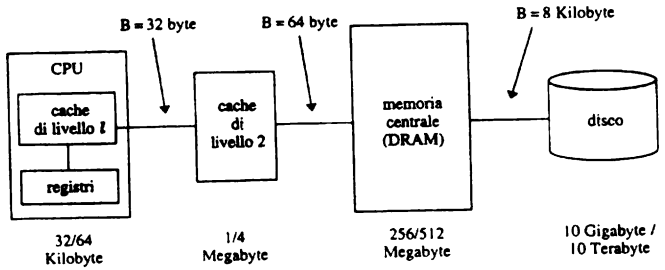


Figura 2.11 Gerarchia di memoria.

Lasciamo come esercizio per il lettore (si veda il Problema 2.10) il compito di analizzare con il metodo del potenziale il costo ammortizzato delle operazioni `insert` e `delete` descritte nell'Esempio 2.14. Vedremo altri esempi di uso del metodo del potenziale nel Capitolo 6.

2.8 * Modelli evoluti: la gerarchia di memoria

Per completezza, concludiamo questo capitolo descrivendo un modello di calcolo più sofisticato rispetto a quelli che abbiamo presentato nel Paragrafo 2.1, ma che meglio si presta ad astrarre una importante caratteristica delle moderne piattaforme di calcolo. Queste ultime contengono infatti una gerarchia di livelli di memoria, ciascuno con il proprio costo e le proprie caratteristiche prestazionali. Come schematizzato in Figura 2.11, la gerarchia spazia da memorie veloci, costose, e con una piccola capacità (come i registri e la cache), a memorie lente, poco costose, e grandi (come i dischi). Il livello più lento nell'intera gerarchia di Figura 2.11 è rappresentato dal disco: tipicamente, un accesso a disco (in breve, *I/O*) richiede un tempo di molti ordini di grandezza superiore rispetto al tempo di accesso alla memoria centrale: ad esempio, 10 millisecondi contro 100 nanosecondi su tipici calcolatori attuali.

D'altra parte, sempre più applicazioni relative a motori di ricerca per il Web, basi di dati, sistemi informativi geografici, devono gestire quantità di dati così grandi che la memoria centrale è insufficiente a contenerli tutti. Ad esempio, le moderne tecnologie satellitari permettono di ottenere mappe molto dettagliate, in cui è possibile associare un'informazione a 32 bit per metro quadrato: considerando la mappa di un intero continente, otterremmo circa 600 Terabyte (ovvero 600×1024 Gigabyte) di dati! In circostanze come queste, il sistema operativo tipicamente provvede a memorizzare parte dei dati su disco, per poi recuperarli quando necessario. Modelli classici come la macchina a registri falliscono nel predire con esattezza il tempo di esecuzione di un algoritmo su tali quantità di dati, poiché assumono che si possa accedere ad una qualunque locazione di memoria in tempo costante. Come abbiamo detto, recuperare un dato da disco è invece molto più lento che non accedere alla memoria centrale.

Per diminuire il costo d'accesso, i dati vengono letti da disco – e portati in memoria centrale – in *blocchi* di dimensione non costante: se la memoria centrale non ha spazio sufficiente, occorre sacrificare un blocco già presente, copiandone il contenuto su disco e creando nuovo spazio. È quindi importante che i dati importati tramite un singolo accesso a disco, fintantoché si trovano in memoria centrale, siano "sfruttati" il più possibile dall'algoritmo: in altre parole, se un algoritmo esibisce proprietà di località nell'accesso ai dati, il numero di I/O viene minimizzato, ottenendo prestazioni migliori. Per analizzare la caratteristica di località di accesso ai dati di un algoritmo, la classica macchina a registri è stata estesa in modo da supportare una visione a due livelli della memoria ed essere resa più realistica. Il *modello di memoria esterna* cattura quanto detto nel modo seguente:

- la memoria esterna ha una dimensione illimitata;
- la memoria interna ha una dimensione finita M;
- i dati sono trasferiti dalla memoria esterna a quella interna (e viceversa) in blocchi di dimensione B. Si assume $B < M/2$.

Tipici valori di B ed M su modemi calcolatori sono 8 Kilobyte e 512 Megabyte, rispettivamente. Come abbiamo già osservato, al fine di avere buone prestazioni in questo modello, è importante che gli algoritmi minimizzino il numero di letture e scritture in memoria esterna. Vediamo come ciò possa essere fatto tramite un semplice esempio, la lettura degli elementi di un array X contenente N interi di 4 byte ciascuno, con $N >> M$, mantenuto in memoria esterna.

Esempio 2.18 Consideriamo un algoritmo $\mathcal{A}_1$ che legge tutti gli elementi di X in modo caotico, ovvero senza un ordine preciso: se ad un certo passo vengono letti i 4 byte corrispondenti all'elemento $X[i]$, l'intero blocco che contiene $X[i]$ viene portato in memoria centrale con un I/O. Il successivo elemento letto, però, si troverà in quel blocco con bassissima probabilità se l'accesso agli elementi è casuale, e quindi anche la lettura successiva causerà un I/O. In totale, per leggere N elementi, nel caso peggiore avremo N I/O.

Ad esempio, se l'array ha dimensione totale $4N = 4$ Gigabyte, per leggere gli $N = 10^9$ elementi si avranno 10^9 I/O. Supponendo che un I/O richieda 10 millisecondi, il tempo totale richiesto sarà quindi di 10^{10} millisecondi, ovvero 10^7 secondi, che corrispondono a poco meno di quattro mesi. □

Esempio 2.19 Consideriamo ora un algoritmo $\mathcal{A}_2$ che legge sequenzialmente gli elementi di X: quando la lettura di un elemento $X[i]$ causa un I/O, i $\approx B/4$ elementi letti successivamente si troveranno già in memoria centrale e quindi non comporteranno I/O, perché si accede ad essi sequenzialmente ed immediatamente dopo $X[i]$. In totale, per leggere N elementi, avremo circa $4N/B$ I/O.

Assumendo, come nell'esempio precedente, che l'array abbia dimensione totale $4N = 4$ Gigabyte e che un blocco abbia dimensione $B = 8$ Kilobyte, per leggere tutti gli elementi in questo caso si avranno $(4 \cdot 10^9)/(8 \cdot 10^3) = 10^6/2$

I/O. Il tempo totale sarà pertanto $10^7/2$ millisecondi, ovvero $10^4/2$ secondi, che corrispondono approssimativamente ad un'ora e venti minuti. □

Nel Paragrafo 6.5 del Capitolo 6 vedremo un uso sofisticato di questo modello in una applicazione di notevole importanza pratica: la gestione di dizionari su memoria esterna.

2.9 Problemi

Problema 2.1 Calcolare il tempo di esecuzione dei seguenti frammenti di programma sia secondo il criterio di costo uniforme che secondo il criterio di costo logaritmico, assumendo che n sia un numero intero:

$x \leftarrow 3$
for $i = 1$ **to** n **do** $x \leftarrow x \cdot x$

$x \leftarrow 1$
for $i = 1$ **to** n **do** $x \leftarrow x \cdot x$

$x \leftarrow 1$
for $i = 1$ **to** n **do** $x \leftarrow x \cdot i$

Problema 2.2 Siano $f(n)$ e $g(n)$ due funzioni sempre positive. Dimostrare formalmente o confutare con un controesempio le seguenti affermazioni:

- se $f(n) \leq g(n)$ per ogni $n \geq 0$, allora $g(n) - f(n) = \Omega(g(n))$;
- se $f(n) = O(g(n))$, allora $2^{f(n)} = O(2^{g(n)})$;
- $f(n) = \Theta(f(n)/3)$;
- $f(n) = \Theta(f(n/3))$;
- $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

Problema 2.3 Sia $T(n)$ il tempo di esecuzione di un algoritmo su istanze di dimensione n e sia $f(n)$ una funzione sempre positiva. Dimostrare formalmente o confutare con un controesempio le seguenti affermazioni:

- se $T(n) = \Omega(f(n))$, allora il tempo di esecuzione dell'algoritmo è $\Omega(f(n))$ su ogni istanza di dimensione n;
- se il tempo di esecuzione è $O(f(n))$ nel caso migliore, allora $T(n) = O(f(n))$;
- $T(n) = \Theta(f(n))$ se e solo se il suo tempo di esecuzione nel caso peggiore è $O(f(n))$ ed il suo tempo di esecuzione nel caso migliore è $\Omega(f(n))$;

Problema 2.4 Per ogni coppia di funzioni $f(n)$ e $g(n)$, si dica se $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, e $f(n) = \Theta(g(n))$ completando la seguente tabella:

$f(n)$	$9^{\log_3 n}$	$n \log\log n$	$2^{n/2}$	1	n^2+8n	n^n	2^n	$4^{\log_2 n}$
$g(n)$	$n\sqrt{n}$	$n^{1+\epsilon}, \epsilon > 0$	$n^{\log\log n}$	2^9	$(n \log n)^2$	$n!$	$2^{n/4}$	$n^{\log_2 4}$
O								
Ω								
Θ								

Problema 2.5 Trovare la soluzione delle seguenti relazioni di ricorrenza, disegnando l'albero della ricorsione, per iterazione, per sostituzione o usando, se possibile, il teorema fondamentale delle ricorrenze:

- $T(n) = 4T(\frac{n}{2}) + n$
- $T(n) = 4T(\frac{n}{2}) + n^2$
- $T(n) = 4T(\frac{n}{2}) + n^3$
- $T(n) = 3T(\frac{n}{3}) + n$
- $T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + n$
- $T(n) = T(n-c) + T(c) + n^2$, dove $c \geq 1$ è una costante
- $T(n) = 2T(\sqrt{n}) + \log n$

Problema 2.6 (*) Dimostrare che, se $a \cdot f(n/b) \leq c \cdot f(n)$ per una costante $c < 1$, allora esiste $\epsilon > 0$ tale che $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Problema 2.7 L'algoritmo in Figura 2.12, dati due array A e B di n elementi ciascuno, restituisce `true` se A e B hanno un elemento in comune, `false` altrimenti. La chiamata iniziale è `elementoInComune`$(A,n,B,0,n-1)$. Qual è il tempo di esecuzione del passo base? Qual è il tempo di esecuzione totale? Si discutano il caso migliore e quello peggiore, assumendo che il passaggio di parametri costi tempo $O(1)$.

Problema 2.8 Definiamo il problema della ricerca di un indice speciale come segue:

> data una sequenza ordinata di n interi distinti, sia positivi che negativi, $a_1 < a_2 < \ldots < a_n$, determinare se esiste un indice i tale che $a_i = i$.

Progettare un algoritmo che risolve il problema della ricerca di un indice speciale in tempo $O(\log n)$.

Problema 2.9 Una sequenza $a_1, a_2, \ldots a_n$ di elementi distinti si dice *ciclicamente ordinata* se il più piccolo elemento è a_i per un qualche indice i non noto, e la sequenza $a_i, a_{i+1}, \ldots a_n, a_1, \ldots a_{i-1}$ è ordinata in modo crescente. Definiamo il problema della ricerca in una sequenza ciclicamente ordinata nel modo seguente:


```
algoritmo elementoInComune(array A e B, interi n, i e j) → booleano
  if ( i = j ) then
    for k = 0 to n do
      if ( A[k] = B[i] ) then return true
    return false
  else
    m ← (i + j)/2
    if ( elementoInComune(A, n, B, i, m) ) then return true
    else return elementoInComune(A, n, B, m + 1, j)
```

Figura 2.12 Un algoritmo ricorsivo per verificare se due array hanno un elemento in comune.

> data una sequenza ciclicamente ordinata di n elementi, trovare l'elemento minimo della sequenza.

Progettare un algoritmo che risolve il problema della ricerca in una sequenza ciclicamente ordinata in tempo $O(\log n)$.

Problema 2.10 Si considerino le operazioni `insert` e `delete` descritte nell'Esempio 2.14. Usando il metodo del potenziale, dimostrare che entrambe richiedono tempo ammortizzato $O(1)$.

2.10 Sommario

In questo capitolo abbiamo introdotto le principali metodologie per l'analisi di algoritmi. Abbiamo innanzitutto evidenziato l'importanza di definire un modello di calcolo appropriato, per poter analizzare con precisione la quantità di una certa risorsa usata da un algoritmo. Le tipiche risorse su cui ci concentreremo in questo libro sono il tempo di esecuzione (misurato in termini di operazioni elementari) e l'occupazione di memoria. Abbiamo però sottolineato che le moderne piattaforme di calcolo hanno architetture sempre più complesse, che richiedono lo sviluppo di modelli sofisticati che astraggano al meglio queste caratteristiche: ad esempio, il modello di memoria esterna che abbiamo descritto nel Paragrafo 2.8 è frutto della necessità di modellare i costi elevati di accesso al disco.

Abbiamo inoltre introdotto un utile formalismo, la notazione asintotica, per esprimere la quantità di una certa risorsa di calcolo usata da un algoritmo durante la sua esecuzione: la notazione asintotica permette di ignorare i dettagli non influenti, come ad esempio le costanti moltiplicative ed i termini di ordine inferiore, e può essere utilizzata in modo naturale per esprimere delimitazioni inferiori e superiori alla complessità di un problema rispetto ad una data risorsa.

Poiché la quantità di risorsa usata da un algoritmo potrebbe essere diversa a seconda dell'istanza in ingresso, per rendere l'analisi indipendente dalla particolare istanza abbiamo espresso questa quantità in funzione della dimensione del-

l'istanza di ingresso stessa. Abbiamo inoltre distinto vari casi di analisi: il caso peggiore, in particolare, permette di misurare la quantità di risorsa per le istanze di ingresso che risultano più sfavorevoli per l'algoritmo. Il caso medio, invece, esprime la quantità di risorsa usata in funzione della distribuzione dei dati in ingresso, che però non sempre è nota. Per poter applicare l'analisi del caso medio anche quando l'istanza non è necessariamente casuale, o quando non conosciamo accuratamente la distribuzione dei dati in ingresso, abbiamo introdotto la tecnica della randomizzazione, che consiste nel manipolare l'istanza usando generatori di numeri casuali in modo tale che essa appaia come un'istanza casuale.
Abbiamo infine mostrato che il tempo di esecuzione di algoritmi ricorsivi può essere espresso tramite relazioni di ricorrenza, discutendo vari metodi per risolvere queste relazioni. In particolare, abbiamo dimostrato il teorema fondamentale delle ricorrenze, che permette di trovare agevolmente la soluzione di molte relazioni di ricorrenza derivanti da algoritmi basati sulla tecnica *divide et impera*.

2.11 Note bibliografiche

Tra i testi che trattano le metodologie di progetto e di analisi di algoritmi ricordiamo [2, 3, 6, 8, 12], che sono una utile fonte di approfondimenti ed esercizi relativi agli argomenti affrontati in questo libro. Knuth ha fornito i primi contributi metodologici rigorosi allo studio degli algoritmi [7, 8, 9]: è in un suo celebre articolo che furono introdotte, ad esempio, le notazioni Ω e Θ [9]. Eccezion fatta per la macchina di Turing, nota già dal 1936 [17], gli anni '70 videro l'astrazione e la formalizzazione dei principali modelli di calcolo, che sono in uso ancora oggi: Shepherdson e Sturgis [14], e più tardi Cook e Reckhow [5], studiarono la macchina a registri, mentre Schönhage [15], basandosi su un precedente lavoro di Kolmogorov e Uspenskij [10], si concentrò sulla macchina a puntatori. Modelli più evoluti, come quello di memoria esterna [1], si sono invece affermati a partire dalla fine degli anni '80, e ancora oggi si assiste ad una intensa ricerca di modelli che allo stesso tempo siano realistici e non rendano troppo complessa l'analisi di algoritmi. L'analisi probabilistica è discussa in [13], ed un libro di riferimento per l'analisi e la progettazione di algoritmi randomizzati è [11]. Il termine "ammortizzato" è dovuto a Sleator e Tarjan: Tarjan illustra in [16] molte applicazioni dei metodi dei crediti e del potenziale. Il Teorema 2.1 per la soluzione delle relazioni di ricorrenza derivanti da algoritmi di tipo *divide et impera* è dovuto a Bentley, Haken, e Saxe [4]. Purdom e Brown trattano rigorosamente vari metodi per risolvere relazioni di ricorrenza [12].

Riferimenti bibliografici

[1] A. Aggarwal e J. S. Vitter "The input/output complexity of sorting and related problems", *Communications of the Association for Computing Machinery*, 31(9):1116–1127, 1988.

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, 1974.

[3] S. Baase, A. Van Gelder, *Computer algorithms: introduction to design and analysis*, 3rd edition, Addison-Wesley, Reading, 2000.

[4] J. L. Bentley, D. Haken, J. B. Saxe, "A general method for solving divide-and-conquer recurrences", *SIGACT News*, 12(3), 36–44, 1980.

[5] S. A. Cook e R. A. Reckhow, "Time Bounded Random Access Machines", *Journal of Computer and System Sciences*, 7(4): 354–375, 1973.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Introduction to algorithms*, McGraw-Hill, 2001.

[7] D. E. Knuth, "Mathematical analysis of algorithms", *IFIP Congress Ser.*, 71, North Holland, Amsterdam, 1972.

[8] D. E. Knuth, *The Art of Computer Programming*, volume 1 *Fundamental algorithms*, volume 2 *Seminumerical algorithms*, volume 3 *Sorting and searching*, Addison-Wesley, Reading, 1973.

[9] D. E. Knuth, "Big Omicron and big Omega and big Theta", *SIGACT News*, 8(2), 18–23, 1976.

[10] A. N. Kolmogorov e V. A. Uspenskij, "On the definition of an algorithm", *Russian Math. Surveys*, 30:217–244, 1963. English translation from *Uspekhi Mat. Nauk*, 13:3–28, 1958.

[11] R. Motwani e P. Raghavan, *Randomized algorithms*, Cumbridge University Press, Cumbridge, 1995.

[12] P. W. Purdom, Jr., C. A. Brown, *The analysis of algorithms*, Holt, Rinehart and Winston, New York, 1985.

[13] M. O. Rabin, "Probabilistic algorithms", in J. F. Traub (ed.) *Algorithms and complexity: new directions and recent results*, 21–39, Academic Press, New York, 1976.

[14] J. C. Shepherdson, E. E. Sturgis, "Computability of recursive functions", *Journal of the Association for Computing Machinery*, 10(2), 217–255, 1963.

[15] A. Schönhage, "Storage modifications machines", *SIAM Journal on Computing*, 9(3):490–508, 1980.

[16] R. E. Tarjan, "Amortized computational complexity", *SIAM Journal on Algebraic and Discrete Methods*, 6(2), 306–318, 1985.

[17] A. M. Turing, "On computable numbers, with an application to the Entscheindungsproblem", *Proceedings London Mathematical Society*, 2(42), 230–265, 1936.

3

Strutture dati elementari

Al vincitore darò da mangiare dell'albero della vita.
(Apocalisse 2.7)

In questo capitolo discuieremo alcune tecniche di base che consentono di gestire in modo efficiente collezioni di oggetti. Useremo la nozione di *tipo di dato* per descrivere le operazioni di interesse sulle nostre collezioni. Ad esempio, diremo che una istanza del tipo di dato `Dizionario` è una collezione di elementi a cui sono associate *chiavi* prese da un dominio totalmente ordinato. Operazioni tipiche su un dizionario sono le seguenti: inserimento (`insert`) di un elemento e di una chiave ad esso associata, cancellazione (`delete`) di un elemento data una chiave e ricerca (`search`) dell'elemento associato a una data chiave. Una possibile specifica del tipo di dato `Dizionario` è illustrata in Figura 3.1. Nel resto del libro useremo quesia semplice notazione per descrivere le operazioni di un tipo di dato, ignorando alle volie alcuni dettagli come possibili situazioni di errore. Ad esempio, assumiamo che in ogni operazione di cancellazione (`delete`) la chiave dell'elemento che si vuole rimuovere appartiene sempre al dizionario.

La nozione di tipo di dato specifica *cosa* un'operazione deve fare, ma non *come* l'operazione può essere realizzata, e soprattutto *come* gli oggetti della collezione possono essere organizzati in modo che le operazioni siano efficienti e la collezione stessa occupi poco spazio in memoria. Ad esempio, la ricerca di una chiave in un dizionario potrebbe essere un'operazione molto costosa in termini di tempo di esecuzione se esso non è strutturato opportunamente: come abbiamo visto nel Capitolo 2, una ricerca sequenziale potrebbe richiedere l'esame di tutte le chiavi della collezione. Se invece la collezione contiene oggetti su cui è definita una relazione di ordinamento totale e viene memorizzata in un array ordinato, l'algoritmo di ricerca binaria richiede tempo logaritmico. Tuttavia, in questo caso l'inserimento e la cancellazione di elementi potrebbe essere molto costosa se vogliamo mantenere la proprietà di ordinamento dell'array. Una possibile realizzazione del tipo di dato `Dizionario` basata su array ordinato è mostrata in Figura 3.2, dove usiamo una notazione orientata agli oggetti che può essere facilmente tradotta in definizioni di classi `C++` o `Java`.

Useremo il termine *struttura dati* riferendoci ad una particolare organizzazione delle informazioni che permette di supportare in modo efficiente le opera-

tipo `Dizionario`:
dati:
un insieme S di coppie $(elem, chiave)$.

operazioni:
`insert`$(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k).

`delete`$(chiave\ k)$
cancella da S la coppia con chiave k.

`search`$(chiave\ k) \rightarrow elem$
se la chiave k è presente in S restituisce l'elemento e ad essa associato, e `null` altrimenti.

Figura 3.1 Il tipo di dato `Dizionario`.

zioni di un tipo di dato. Riprendendo l'esempio precedente, un array ordinato di n elementi è una semplice struttura dati che supporta operazioni di ricerca in tempo $O(\log n)$ ed operazioni di inserimento/cancellazione in tempo $O(n)$. Nel Capitolo 6 vedremo che le operazioni del tipo di dato `Dizionario` possono essere tutte eseguite in modo molto efficiente utilizzando strutture dati collegate ad albero. In generale, vedremo come per uno stesso tipo di dato sono possibili molteplici realizzazioni alternative basate su strutture dati diverse che permettono di implementare le operazioni richieste in modo più o meno efficiente.

Nel resto di questo capitolo richiameremo alcune classiche tecniche per la rappresentazione di collezioni di oggetti ed alcuni semplici tipi di dato, come le pile, le code e gli alberi, discutendone possibili implementazioni efficienti. Come vedremo, questi tipi di dato sono ingredienti fondamentali per la risoluzione di molti problemi algoritmici classici. Assumeremo che il lettore abbia familiarità con l'uso di array e strutture concatenate, e ci concentreremo sullo studio delle prestazioni delle operazioni realizzate.

3.1 Tecniche per rappresentare collezioni di oggetti

In questo paragrafo discuteremo due tecniche fondamentali usate per rappresentare collezioni di elementi: quella basata su *strutture indicizzate* (array) e quella basata su *strutture collegate* (record e puntatori). Vedremo come la scelta di una tecnica piuttosto che di un'altra avrà un impatto cruciale sulle prestazioni di molte operazioni fondamentali come ricerca, inserimento e cancellazione.

3.1.1 Strutture indicizzate: array

Un array è una *struttura indicizzata* costituita da una collezione di celle numerate che possono contenere elementi di un tipo prestabilito. In questo testo assumeremo che in un array di dimensione h (contenente h celle) gli indici delle celle possano essere compresi indifferentemente tra 0 e $h-1$, oppure tra 1 e h. Useremo di volta in volta la scelta che maggiormente facilita l'esposizione, specificandola esplicitamente. In questo capitolo gli indici saranno compresi tra 0 e $h-1$. Inoltre, assumeremo che è possibile accedere in lettura e scrittura a una qualsiasi cella di un array in tempo costante. Le due proprietà basilari degli array sono le seguenti:

classe `ArrayOrdinato` **implementa** `Dizionario`:
dati: $S(n) = \Theta(n)$
un array S di dimensione n contenente coppie $(elem, chiave)$.

operazioni:
`insert`$(elem\ e, chiave\ k)$ $T(n) = O(n)$
rialloca l'array S aumentandone la dimensione n di uno; cerca il più piccolo indice i tale che $k \leq S[i].chiave$ e pone $S[j] \leftarrow S[j-1]$ per ogni j da $n-1$ a $i+1$; infine, pone $S[i] \leftarrow (e, k)$.

`delete`$(chiave\ k)$ $T(n) = O(n)$
trova l'indice i della coppia con chiave k in S e pone $S[j] \leftarrow S[j+1]$ per ogni j da i a $n-2$; infine, rialloca l'array S diminuendone la dimensione n di uno.

`search`$(chiave\ k) \rightarrow elem$ $T(n) = O(\log n)$
esegue l'algoritmo di ricerca binaria su S per verificare se S contiene la chiave k. Se la ricerca ha successo restituisce l'elemento e associato alla chiave, altrimenti restituisce `null`.

Figura 3.2 Dizionario realizzato mediante array ordinato.

Proprietà 3.1 (Forte) *Gli indici delle celle di un array sono numeri consecutivi.*

Proprietà 3.2 (Debole) *Non è possibile aggiungere nuove celle ad un array.*

La proprietà debole implica che il ridimensionamento di un array è possibile solo mediante *riallocazione*, cioè creando un nuovo array con la dimensione voluta, e copiando il contenuto delle celle di interesse dal vecchio array al nuovo array. Questo richiede tempo proporzionale al minimo fra la vecchia e la nuova dimensione. Le due proprietà viste hanno conseguenze molto importanti per l'efficienza delle operazioni realizzabili usando array.

Consideriamo nuovamente l'esempio del dizionario implementato con array descritto in Figura 3.2. Osserviamo che l'ordinamento dell'array e la sua proprietà forte ci permettono di usare la tecnica di ricerca binaria per realizzare l'operazione `search` in tempo logaritmico. Tuttavia, questo è pagato in termini di un tempo lineare di inserimento e di cancellazione di coppie nel dizionario. A prima vista, potrebbe sembrare che questo è dovuto principalmente alla proprietà debole degli

array che impone la riallocazione allorché servano nuove celle. Vedremo invece che il problema principale consiste nel mantenere l'array ordinato: rinunciando alla proprietà di ordinamento, nonostante la proprietà debole, possiamo supportare le operazioni di inserimento/cancellazione in un array in tempo costante in media usando ancora spazio lineare.

*** La tecnica del raddoppiamento-dimezzamento.** Vediamo ora una tecnica molto semplice che consente di mantenere efficientemente in un array una collezione non ordinata di n elementi soggetta a operazioni di inserimento e cancellazione. La tecnica è chiamata *raddoppiamento-dimezzamento* (*doubling-halving*) e consiste nel mantenere un array di dimensione h, dove per ogni $n > 0$, h soddisfa la seguente invariante:

$$n \leq h < 4n.$$

Le prime n celle dell'array contengono gli elementi della collezione, mentre il contenuto delle altre celle è indefinito. L'idea principale è quella di non effettuare riallocazioni ad ogni inserimento/cancellazione, ma solo ogni $\Omega(n)$ operazioni. L'invariante $n \leq h \leq 4n$ sulla dimensione dell'array viene mantenuta mediante riallocazioni, definite nel modo seguente:

- Inizialmente, quando $n = 0$, poniamo $h = 1$.
- Ogni qualvolta n supera h, l'array viene riallocato raddoppiandone la dimensione ($h \leftarrow 2h$).
- Ogni qualvolta n scende a $h/4$, l'array viene riallocato dimezzandone la dimensione ($h \leftarrow h/2$).

In questo modo, per ogni $n > 0$ si mantiene l'invariante: $h/4 < n \leq h$, ovvero $n \leq h < 4n$. Lo spazio utilizzato per memorizzare la collezione è $S(n) = \Theta(h) = \Theta(n)$. Si noti che h è sempre una potenza di 2, e quindi $h/2$ e $h/4$ sono sempre interi per ogni $h \geq 4$.

Le operazioni di inserimento e cancellazione possono essere realizzate come segue. Previo eventuale raddoppiamento dell'array, ogni nuovo elemento viene inserito nella cella di indice n, e poi si incrementa n di uno. Per cancellare un elemento in posizione i, con $0 \leq i < n$, basta sovrascriverlo con l'elemento in posizione $(n-1)$, decrementando poi n di uno ed eventualmente dimezzando l'array. Dimostriamo ora che queste operazioni richiedono tempo ammortizzato costante (vedi il Paragrafo 2.7 del Capitolo 2 per una definizione di costo ammortizzato).

Teorema 3.1 *Se v è un array di dimensione $h \geq n$ contenente una collezione non ordinata di n elementi, usando la tecnica del raddoppiamento-dimezzamento ogni operazione di inserimento o cancellazione di un elemento richiede tempo ammortizzato costante, cioè $T_{am}(n) = O(1)$.*

Dimostrazione. Osserviamo innanzitutto che, a parte le riallocazioni, ogni inserimento e cancellazione richiede tempo costante. Usando il metodo dei crediti

classe `ArrayDoubling` **implementa** `Dizionario`:
dati: $S(n) = \Theta(n)$
un array S di dimensione h, con $n \leq h < 4n$, contenente n coppie $(elem, chiave)$.

operazioni:
`insert`$(elem\ e, chiave\ k)$ $T_{am}(n) = O(1)$
se $n = h$ rialloca S raddoppiandone la dimensione h; pone $S[n] \leftarrow (e, k)$, ed incrementa n di uno.

`delete`$(chiave\ k)$ $T(n) = O(n)$
se $n = h/4$ rialloca S dimezzandone la dimensione h; cerca l'indice i dell'elemento con chiave k, pone $S[i] \leftarrow S[n-1]$ e decrementa n di uno.

`search`$(chiave\ k) \rightarrow elem$ $T(n) = O(n)$
esegue l'algoritmo di ricerca sequenziale su S per verificare se S contiene la chiave k. Se la ricerca ha successo restituisce l'elemento e associato alla chiave, altrimenti restituisce `null`.

Figura 3.3 Dizionario realizzato mediante array non ordinato mantenuto con la tecnica del raddoppiamento-dimezzamento.

introdotto nel Paragrafo 2.7.1 del Capitolo 2, mostriamo che possiamo pagare il costo di tutte le riallocazioni usando monete depositate dagli inserimenti sulle celle dell'array. In particolare, assumiamo che ogni inserimento depositi tre monete sulla cella scritta. Quando $n = h$ e si deve aggiungere un nuovo elemento, la riallocazione richiede di copiare i primi n elementi dal vecchio al nuovo array: questo costo temporale è pagato sottraendo 2 monete a ciascuna delle $n/2$ celle con indici compresi tra $(h/2 - 1)$ e $(h - 1)$. Quando $n = h/4 + 1$ e si deve rimuovere un elemento, la riallocazione richiede di copiare i primi n elementi dal vecchio al nuovo array: questo costo temporale è pagato sottraendo 1 moneta a ciascuna delle n celle con indici compresi tra $(h/4 - 1)$ e $(h/2 - 1)$. Usando induzione sul numero di operazioni, non è difficile dimostrare che, così facendo, in ogni istante risultano verificate le seguenti due proprietà invarianti:

- su ciascuna delle prime $h/2$ celle dell'array vi è almeno una moneta;
- se $n > h/2$, su ciascuna delle successive $(n - h/2)$ celle vi sono almeno tre monete.

Il costo ammortizzato di ogni inserimento/cancellazione è quindi costante. Osserviamo invece che il tempo per operazione nel caso peggiore è $O(n)$. □

Come esempio, in Figura 3.3 mostriamo una possibile realizzazione del tipo di dato `Dizionario` usando la tecnica del raddoppiamento-dimezzamento. Il fatto che l'array è non ordinato ci impedisce di usare ricerca binaria, e quindi il costo dell'operazione `search` è $O(n)$ usando ricerca sequenziale. Nonostante la

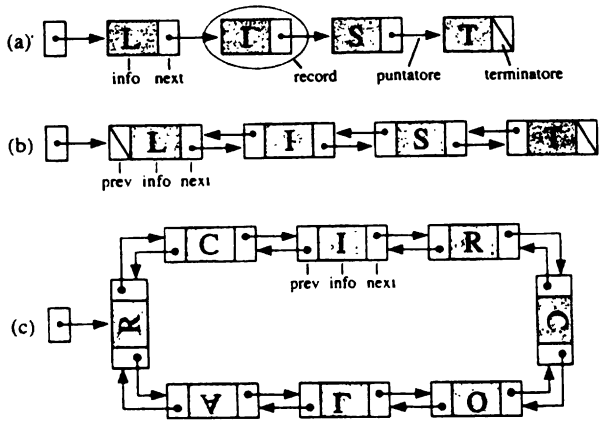


Figura 3.4 Esempi di strutture collegate. (a) lista semplice; (b) lista doppiamente collegata; (c) lista circolare doppiamente collegata.

cancellazione effettiva dell'elemento dall'array richieda tempo costante ammortizzato, l'operazione `delete` richiede tempo $O(n)$ poiché l'indice della chiave da cancellare non è noto a priori e va cercato nell'array.

3.1.2 Strutture collegate: record e puntatori

Un'attra tecnica fondamentale per rappresentare collezioni di elementi è quella basata su strutture collegate. I costituenti di base di una struttura collegata sono i *record*, che, come le celle degli array, sono numerati e contengono ciascuno un oggetto della collezione. Mentre la numerazione delle celle degli array è locale al singolo array, i numeri associati ai record sono tipicamente i loro *indirizzi* in memoria, e quindi sono globali nell'ambito di un programma. Diversamente dalle celle degli array, che vengono create contestualmente all'array che le contiene, i record sono creati e distrutti individualmente ed in maniera dinamica nella memoria del calcolatore. Pertanto, diversamente dagli indici degli array, gli indirizzi dei record non sono necessariamente consecutivi. Per mantenere i record di una collezione in relazione tra loro, ognuno di essi contiene uno o più indirizzi di altri record della collezione. Se un record A contiene l'indirizzo di un altro record B, diremo che esiste un *collegamento* tra A e B realizzato tramite un *puntatore*. I collegamenti tra record permettono ad un programma di esplorare una struttura collegata "saltando" di record in record: a questo scopo, è importante che ci sia sempre un record da cui si possano raggiungere tutti gli altri. In Figura 3.4 illustriamo alcuni tipici modi di organizzare record in strutture collegate. Questa organizzazione basata su collegamenti permette di inserire e cancellare record in modo molto flessibile, semplicemente aggiornando puntatori nella struttura. Diversamente dagli array, le strutture collegate sono quindi particolarmente

classe `StrutturaCollegata` **implementa** `Dizionario`:
dati: $S(n) = \Theta(n)$
una collezione di n record contenenti ciascuno una quadrupla $(elem, chiave, next, prev)$, dove $next$ e $prev$ sono puntatori al successivo e precedente record nella collezione, rispettivamente. Manteniamo inoltre un puntatore $list$ che contiene l'indirizzo di un record se la collezione non è vuota, e `null` altrimenti.

operazioni:
`insert`$(elem\ e, chiave\ k)$ $T(n) = O(1)$
viene creato un record p con elemento e e chiave k. Se $list$ = `null`, si effettua $p.next \leftarrow p$, $p.prev \leftarrow p$ e $list \leftarrow p$. Altrimenti, si collega il record p tra $list$ e $list.next$ effettuando $p.next \leftarrow list.next$, $list.next.prev \leftarrow p$, $p.prev \leftarrow list$ e $list.next \leftarrow p$.

`delete`$(chiave\ k)$ $T(n) = O(n)$
si trova il record p con chiave k come nella `search`; poi, si effettua $p.prev.next \leftarrow p.next$ e $p.next.prev \leftarrow p.prev$; infine, viene distrutto il record p.

`search`$(chiave\ k) \rightarrow elem$ $T(n) = O(n)$
se $list$ = `null` si restituisce `null`. Altrimenti, si scandisce la struttura saltando di record in record con $p \leftarrow p.next$ fino a quando non diventa $p = list$, verificando se qualche p ha chiave k. In caso positivo si restituisce l'elemento trovato, e `null` altrimenti.

Figura 3.5 Dizionario realizzato mediante struttura circolare doppiamente collegata.

adatte a realizzare operazioni di inserimento e cancellazione. Riassumiamo le due proprietà basilari delle strutture collegate:

Proprietà 3.3 (Forte) *È possibile aggiungere o togliere record a una struttura collegata.*

Proprietà 3.4 (Debole) *Gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi.*

Come nel caso degli array, queste proprietà hanno conseguenze importanti per l'efficienza delle operazioni realizzabili. Consideriamo ancora una volta l'esempio del dizionario: in Figura 3.5 è illustrata una sua realizzazione mediante una struttura circolare doppiamente collegata (vedi Figura 3.4 (c)). Osserviamo innanzitutto che la proprietà debole non consente l'uso della ricerca binaria tout-court; la `search` richiede quindi tempo lineare usando ricerca sequenziale. Nel Capitolo 6 vedremo che usando una struttura collegata ad albero è possibile supportare l'operazione `search` in tempo logaritmico usando un procedimento concettualmente simile a quello di ricerca binaria. L'inserimento e la cancellazione in una

tipo `Pila`:
dati:
una sequenza S di n elementi.

operazioni:
`isEmpty()` → *result*
restituisce `true` se S è vuota, e `false` altrimenti.

`push(`*elem e*`)`
aggiunge e come ultimo elemento di S.

`pop()` → *elem*
toglie da S l'ultimo elemento e lo restituisce.

`top()` → *elem*
restituisce l'ultimo elemento di S (senza toglierlo da S).

Figura 3.6 Il tipo di dato `Pila`.

struttura collegata richiedono tempo costante *nel caso peggiore*, diversamente dal caso degli array mantenuti con la tecnica del raddoppiamento-dimezzamento, dove queste operazioni richiedono tempo costante *ammortizzato*. Si noti comunque che, a meno che l'indirizzo del record contenente l'elemento da cancellare non sia noto, la `delete` richiede dapprima una ricerca, che come visto in questo caso costa tempo lineare.

3.2 Pile e code

Nella vita reale, ci troviamo spesso a trattare con collezioni di oggetti. Quando raccogliamo i piatti dopo una cena, li impiliamo formando una catasta; se li poggiamo nel lavandino, li riprendiamo nell'ordine opposto a quello in cui li abbiamo impilati: l'ultimo piatto depositato sulla pila sarà il primo ad essere lavato. Formalizziamo questa particolare disciplina di accesso, chiamata *LIFO* (*last-in first-out*), definendo il tipo di dato `Pila` mostrato in Figura 3.6. Come vedremo in seguito, questo tipo di dato sarà utile per risolvere diversi problemi algoritmici. Gli inserimenti in una pila (`push`) aggiungono elementi alla fine della sequenza, mentre le cancellazioni (`pop`) ne rimuovono sempre l'ultimo elemento. In una pila, gli accessi avvengono quindi ad una sola estremità della sequenza di elementi, e nessun elemento interno può essere estratto prima che tutti quelli che lo seguono siano stati estratti.

Un altro classico scenario reale è quello delle code di attesa: normalmente (anche se purtroppo non sempre accade) gli utenti di un ufficio postale si incolonnano davanti allo sportello in modo tale che il primo arrivato sia il primo ad essere servito. Formalizziamo questa disciplina di accesso, chiamata *FIFO* (*first-in first-out*), definendo il tipo di dato `Coda` mostrato in Figura 3.7. Come le pile,

tipo `Coda`:
dati:
una sequenza S di n elementi.

operazioni:
`isEmpty()` → *result*
restituisce `true` se S è vuota, e `false` altrimenti.

`enqueue(`*elem e*`)`
aggiunge e come ultimo elemento di S.

`dequeue()` → *elem*
toglie da S il primo elemento e lo restituisce.

`first()` → *elem*
restituisce il primo elemento di S (senza toglierlo da S).

Figura 3.7 Il tipo di dato `Coda`.

anche questo tipo di dato risulterà utile in seguito. In una coda, gli inserimenti (`enqueue`) aggiungono elementi alla fine della sequenza, mentre le cancellazioni (`dequeue`) ne rimuovono sempre il primo elemento. In una coda, gli accessi avvengono quindi ad entrambe le estremità della sequenza di elementi, e nessun elemento interno può essere estratto prima che tutti quelli che lo precedono siano stati estratti.

Pile e code possono essere realizzate usando sia strutture indicizzate che collegate, in modo che tutte le operazioni richiedano tempo costante. Osserviamo comunque che per questi tipi di dato l'uso di strutture collegate risulta più agevole a causa della loro naturale dinamicità. Assumiamo che il lettore abbia familiarità con le operazioni di base su array e su strutture collegate (ad esempio inserimento e rimozione di record in una lista collegata) e rimandiamo al Problema 3.1 e al Problema 3.2 per possibili realizzazioni di pile e code.

3.3 Alberi

In molte situazioni reali trattiamo con collezioni di oggetti su cui sono definite in modo naturale delle relazioni gerarchiche. Ad esempio, ogni persona ha una relazione di discendenza diretta con i suoi due genitori, i quali a loro volta sono in relazione con i propri genitori, e così via. Un albero genealogico è un tipico esempio di collezione di oggetti su cui sono definite delle relazioni gerarchiche (vedi Figura 3.8). Un altro esempio è l'albero delle chiamate ricorsive visto nel Capitolo 2, dove gli oggetti di interesse sono le funzioni eseguite in un programma e le relazioni considerate sono le chiamate che le attivano. In questo paragrafo formalizzeremo il concetto di *albero* e vedremo possibili modi di rappresentare alberi nella memoria di un calcolatore. Come vedremo nell'ambito delle strutture

Figura 3.8 Albero Genealogico della regina Claudia di Francia, figlia di Luigi XII e moglie di Francesco I. Parigi, Dipartimento dei Manoscritti, 1517.

dati, organizzare le informazioni di interesse sotto forma di alberi sarà utile anche quando sui dati stessi non sono definite in modo naturale delle relazioni.

Un albero (radicato) è una coppia $T = (N, A)$ costituita da un insieme N di *nodi* e da un insieme $A \subseteq N \times N$ di coppie di nodi, dette *archi*. Come vedremo nel Capitolo 11, un albero è un particolare tipo di *grafo*. In un albero, ogni nodo v (tranne la *radice*) ha un solo *genitore* (o *padre*) u tale che $(u, v) \in A$. Un nodo u può avere zero o più *figli* v tali che $(u, v) \in A$, e il loro numero viene chiamato *grado* del nodo. Un nodo senza figli è chiamato *foglia*, mentre nodi che non sono né foglie né la radice sono chiamati *nodi interni*. Parleremo inoltre di *antenati* e *discendenti* di un nodo intendendo i nodi raggiungibili da quel nodo salendo di padre in padre o scendendo di figlio in figlio nell'albero, rispettivamente. La *profondità* (o *livello*) di un nodo è il numero di archi che bisogna attraversare per raggiungerlo a partire dalla radice. Più formalmente, essa può essere definita ricorsivamente come segue:

1. La radice ha profondità zero;
2. Se un nodo ha profondità k, tutti i suoi figli hanno profondità $k + 1$;

Nodi con lo stesso genitore vengono detti *fratelli*, ed essi avranno quindi la stessa profondità. L'*altezza di un albero* è la massima profondità a cui si trova una foglia. Si veda la Figura 3.9 per un quadro riassuntivo della terminologia su alberi.

Come già osservato, molte strutture dati efficienti sono basate su alberi. In questo contesto, particolare rilievo hanno alcune classi di alberi con vincoli strutturali, che ne facilitano la rappresentazione o consentono di realizzare operazioni

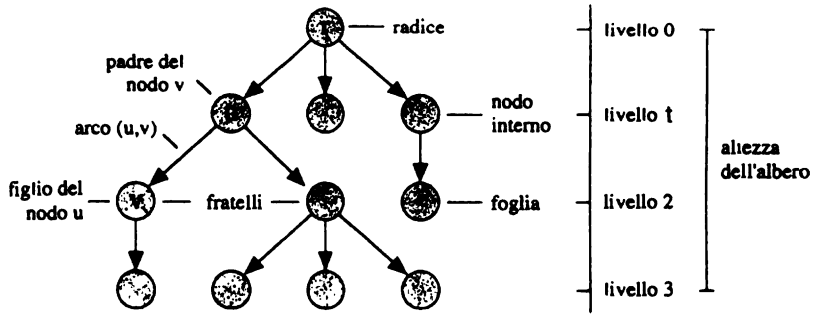


Figura 3.9 Esempio di albero radicato e terminologia.

su collezioni di elementi in modo efficiente. I principali vincoli strutturali riguardano il grado dei nodi e la profondità. In particolare, un albero d-ario è un albero in cui tutti i nodi tranne le foglie hanno grado d. Per $d = 2$ diremo che l'albero è *binario*. Infine, diremo che un albero d-ario in cui tutte le foglie sono sullo stesso livello è *completo*.

In Figura 3.10 mostriamo una descrizione schematica di un possibile tipo di dato `Albero`. Nel seguito di questo paragrafo discuteremo diverse rappresentazioni classiche per gli alberi, che permetteranno di realizzare le operazioni del tipo di dato `Albero` più o meno efficientemente. Per ogni rappresentazione, ci concentreremo sul tempo richiesto dalle operazioni `padre` e `figli`, lasciando come esercizio per il lettore lo studio delle altre operazioni (vedi Problema 3.4). Assumeremo che ogni nodo può contenere uno o più valori che ne rappresentano il contenuto informativo.

3.3.1 Rappresentazioni indicizzate

Vedremo ora due delle più comuni rappresentazioni di alberi basate su array: il *vettore padri* e il *vettore posizionale*. Entrambe richiedono spazio $O(n)$ per un albero con n nodi. L'idea di base è quella di rappresentare ogni nodo dell'albero con una cella di un array che contiene l'informazione associata al nodo, più eventualmente altri indici che consentono di raggiungere altri nodi dell'albero. Sebbene di facile realizzazione, le rappresentazioni basate su array rendono tipicamente difficoltoso l'inserimento e la cancellazione di nodi nell'albero.

Vettore padri. La più semplice rappresentazione possibile per un albero è forse quella basata sul vettore padri. Sia $T = (N, A)$ un albero con n nodi numerati da 0 a $(n-1)$. Un *vettore padri* è un array P di dimensione n le cui celle contengono coppie $(info, parent)$: per ogni indice $v \in [0, n-1]$, $P[v].info$ è il contenuto informativo del nodo v, mentre $P[v].parent = u$ se e solo se vi è un arco $(u, v) \in A$. Se invece v è la radice, allora $P[v].parent =$ `null`.

tipo `Albero`:
dati:
un insieme di nodi (di tipo *nodo*) e un insieme di archi.

operazioni:

`numNodi()` → *intero*
restituisce il numero di nodi presenti nell'albero.

`grado(`*nodo* v`)` → *intero*
restituisce il numero di figli del nodo v.

`padre(`*nodo* v`)` → *nodo*
restituisce il padre del nodo v nell'albero, o `null` se v è la radice.

`figli(`*nodo* v`)` → (*nodo*, *nodo*, ..., *nodo*)
restituisce, uno dopo l'altro, i figli del nodo v.

`aggiungiNodo(`*nodo* u`)` → *nodo*
inserisce un nuovo nodo v come figlio di u nell'albero e lo restituisce. Se v è il primo nodo ad essere inserito nell'albero, esso diventa la radice (e u viene ignorato).

`aggiungiSottoalbero(Albero` a, *nodo* u`)`
inserisce nell'albero il sottoalbero a in modo che la radice di a diventi figlia di u.

`rimuoviSottoalbero(`*nodo* v`)` → `Albero`
stacca e restituisce l'intero sottoalbero radicato in v. L'operazione cancella dall'albero il nodo v e tutti i suoi discendenti.

Figura 3.10 Possibili dati e operazioni su alberi.

Osserviamo che usando un vettore padri, da ogni nodo è possibile risalire in tempo $O(1)$ al proprio padre, mentre trovare un figlio richiede una scansione dell'array in tempo $O(n)$. Per ogni v, l'operazione `padre`(v) può essere quindi realizzata in tempo costante, mentre l'operazione `figli`(v) richiede tempo $O(n)$ per enumerare tutti i figli di v.

Vettore posizionale. Nel caso particolare degli alberi d-ari completi, con $d \geq 2$, è possibile usare una rappresentazione indicizzata dove ogni nodo ha una posizione prestabilita nella struttura. Sia $T = (N, A)$ un albero d-ario con n nodi numerati da 1 a n. Un *vettore posizionale* è un array P di dimensione n tale che $P[v]$ contiene l'informazione associata al nodo v, e tale che l'informazione associata all'i-esimo figlio di v è in posizione $P[d \cdot v + i]$, per i compreso tra 0 e $(d-1)$. Ad esempio, dato un nodo v in un albero binario completo, il figlio sinistro di v sarà in posizione $2v$ e quello destro in posizione $2v+1$. Per semplicità di indicizzazione, la posizione 0 dell'array non è utilizzata. Lo spazio richiesto per rappresentare un albero completo d-ario con n nodi è quindi $n+1$.

Diversamente dalle altre rappresentazioni considerate in questo capitolo, in

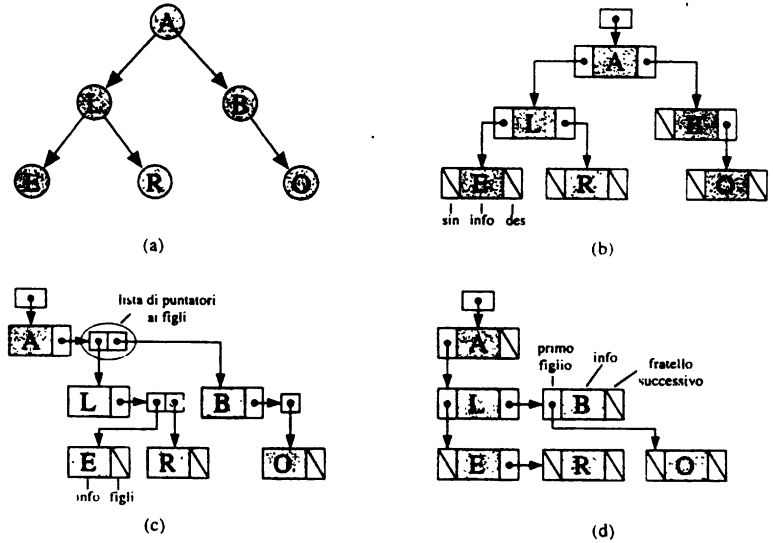


Figura 3.11 Rappresentazioni collegate di alberi. (a) Esempio di albero; (b) Albero di Figura (a) con rappresentazione di tipo *puntatori ai figli per* $d = 2$; (c) Albero di Figura (a) con rappresentazione di tipo *lista di figli*; (d) Albero di Figura (a) con rappresentazione di tipo *primo figlio-fratello successivo*.

un vettore posizionale le relazioni tra nodi non sono esplicite, ma definite implicitamente dalla posizione relativa dei nodi nella struttura. Si noti che utilizzando un vettore posizionale, da ogni nodo v è possibile risalire in tempo $O(1)$ sia al proprio padre (che ha indice $\lfloor v/d \rfloor$ se v non è la radice), che a uno qualunque dei figli: per ogni nodo v, l'operazione `padre`(v) può essere quindi realizzata in tempo costante, mentre l'operazione `figli`(v) richiede tempo $O($`grado`$(v))$.

3.3.2 Rappresentazioni collegate

Le rappresentazioni collegate di alberi sono più flessibili di quelle indicizzate, e permettono di supportare modifiche alla struttura di un albero molto più efficientemente. L'idea di base è quella di rappresentare ogni nodo dell'albero con un record che contiene l'informazione associata al nodo, più altri puntatori che consentono di raggiungere altri nodi dell'albero. Vedremo ora vari modi in cui mantenere collegamenti tra un nodo e i suoi figli, come illustrato in Figura 3.11. Per consentire una risalita in tempo costante da un nodo al suo genitore, ognuna delle rappresentazioni che vedremo può essere arricchita in modo che ciascun nodo contenga anche un puntatore al proprio padre: in questo modo, per ogni nodo v potremo supportare l'operazione `padre`(v) in tempo costante. Come vedre-

```
algoritmo visitaGenerica(nodo r)
1.   S ← {r}
2.   while (S ≠ ∅) do
3.       estrai un nodo u da S
4.       visita il nodo u
5.       S ← S ∪ { figli di u }
```

Figura 3.12 Algoritmo di visita generica di un albero a partire da un nodo r.

mo, usando rappresentazioni collegate l'operazione `figli`(v) può essere sempre realizzata in modo che richieda tempo proporzionale al grado di v.

Puntatori ai figli. Se ogni nodo dell'albero ha grado al più d, è possibile mantenere in ogni nodo un puntatore a ciascuno dei possibili d figli come illustrato in Figura 3.11 (b) nel caso $d = 2$. Se un figlio è assente, si pone il puntatore a `null`, che è una valore speciale usato tipicamente in linguaggi di programmazione come il C++ o Java per indicare un riferimento nullo. Lo spazio richiesto sarà $O(n \cdot d)$, che per d costante è $O(n)$. Ad esempio, se dobbiamo rappresentare un albero binario, manterremo per ogni nodo un puntatore al figlio destro ed uno al figlio sinistro.

Lista figli. Se il numero massimo di figli non è noto a priori, è possibile associare ad ogni nodo una lista di puntatori ai suoi figli come illustrato in Figura 3.11 (c). Questa lista può essere a sua volta rappresentata in modo indicizzato o collegato. In questo modo, lo spazio richiesto per rappresentare un albero con n nodi sarà $O(n)$ indipendentemente dal numero di figli di un nodo.

Primo figlio-fratello successivo. Come variante alla soluzione precedente senza dover usare una struttura dati addizionale (la lista di figli) per ogni nodo, è possibile mantenere per ogni nodo un puntatore al primo figlio (posto a `null` se non vi sono figli), e un puntatore al fratello successivo (posto a `null` se non vi sono fratelli successivi), come mostrato in Figura 3.11 (d). In questo modo, lo spazio richiesto sarà ovviamente ancora $O(n)$. Per scandire tutti i figli di un nodo, basta quindi scendere sul primo figlio, e poi scorrere i fratelli successivi saltando di fratello in fratello.

3.3.3 Visite di alberi

In molti casi, è necessario attraversare un albero visitandone tutti i nodi. Diversamente da collezioni di oggetti rappresentati in modo lineare, dove è possibile attraversare la struttura senza incontrare ramificazioni, nel caso degli alberi è necessario invece seguire tutti i rami possibili a partire dalla radice.


```
algoritmo visitaDFS(nodo r)
  Pila S
  S.push(r)
  while (not S.isEmpty()) do
    u ←S.pop()
    if (u ≠ null) then
      visita il nodo u
      S.push(figlio destro di u)
      S.push(figlio sinistro di u)
```

Figura 3.13 Visita in profondità iterativa di un albero binario. L'insieme dei nodi aperti S è rappresentato mediante una `Pila`.

Descriviamo ora un procedimento iterativo di visita generica, illustrato in Figura 3.12, che consente di visitare tutti i nodi di un albero raggiungibili a partire da un nodo prefissato (tipicamente la radice). L'algoritmo mantiene istante per istante in un insieme S contenente i nodi che rappresentano i punti di ramificazione rimasti in sospeso e da cui la visita deve proseguire. Diremo che questi nodi sono *aperti* e formano una *frangia* dell'albero, mentre diremo che un nodo diventa *chiuso* quando viene rimosso da S. Inizialmente, solo la radice sarà aperta (riga 1). Il passo di visita (righe 2–5) estrae un nodo aperto u chiudendolo (riga 3), lo visita (riga 4), e apre tutti i suoi figli aggiungendoli a S (riga 5). La visita può essere una qualunque elaborazione del nodo u.

Teorema 3.2 *L'algoritmo di visita generica applicato alla radice di un albero con n nodi termina in $O(n)$ passi. Lo spazio usato è $O(n)$.*

Dimostrazione. Assumiamo che le operazioni di inserimento e cancellazione di un elemento in S richiedano ciascuna tempo $O(1)$. Osserviamo che, poiché in un albero non vi è più modo di tornare ad un nodo a partire da uno dei suoi figli procedendo di figlio in figlio, ogni nodo verrà aperto e chiuso una sola volta. Quindi le iterazioni del ciclo `while` saranno al più $O(n)$. Poiché ogni nodo compare al più una volta in S, lo spazio richiesto è $O(n)$. □

Vedremo ora che, a seconda della struttura dati utilizzata per rappresentare l'insieme dei nodi aperti S, otteniamo visite con proprietà diverse.

Visita in profondità. Partendo dall'algoritmo di visita generica di Figura 3.12 e rappresentando l'insieme dei nodi aperti S mediante il tipo di dato `Pila` visto nel Paragrafo 3.2, otteniamo la *visita in profondità* (*depth-first search* o *DFS*). L'algoritmo di visita in profondità è illustrato in Figura 3.13. Si noti l'uso di `null` per indicare l'assenza di un figlio.

In una visita in profondità, si prosegue la visita dall'ultimo nodo lasciato in sospeso: poiché mettiamo in pila prima il figlio destro di ogni nodo e poi quello sinistro, tenderemo a seguire tutti i figli sinistri andando in profondità fino a che non

```
algoritmo visitaDFSRicorsiva(nodo r)
1.   if (r = null) then return
2.   visita il nodo r
3.   visitaDFSRicorsiva(figlio sinistro di r)
4.   visitaDFSRicorsiva(figlio destro di r)
```

Figura 3.14 Visita in profondità ricorsiva di un albero binario. I nodi aperti vengono mantenuti sfruttando la pila dei record di attivazione delle chiamate ricorsive.

si raggiunge la prima foglia sinistra. In generale, si passerà a visitare ogni sottoalbero destro di un nodo solo quando il sottoalbero sinistro è stato completamente visitato.

Esempio 3.1 L'ordine di visita dei nodi dell'albero binario in Figura 3.11 (a) mediante l'algoritmo di visita in profondità in Figura 3.13 è: A L E R B O. □

Naturalmente, impilando prima il figlio sinistro e poi quello destro si ottiene una variante dell'algoritmo con un effetto del tutto simmetrico.

Visita in profondità ricorsiva. Osserviamo che esiste una realizzazione ricorsiva molto elegante dell'algoritmo di visita in profondità, mostrata in Figura 3.14. Si noti che la pila S non appare esplicitamente nell'algoritmo: l'uso della ricorsione permette di usare la pila dei record di attivazione delle chiamate ricorsive per mantenere i nodi aperti. In particolare, si osservi che il nodo u passato come parametro rappresenta la radice del sottoalbero correntemente visitato, e i vari nodi u lasciati in sospeso saranno impilati come parametri delle chiamate ricorsive pendenti. Una chiamata termina se è applicata a un sottoalbero vuoto (u è `null`) oppure quando entrambi i sottoalberi di u sono stati interamente visitati.

Osserviamo che posizionando diversamente l'operazione di visita del nodo r rispetto alle chiamate ricorsive possiamo ottenere varianti dell'algoritmo in cui l'ordine di visita dei nodi dell'albero cambia. In particolare, tre varianti classiche della visita in profondità sono le seguenti:

- *Visita in preordine:* si visita prima la radice, poi si effettuano le chiamate ricorsive sul figlio sinistro e destro (come in Figura 3.14).
- *Visita simmetrica:* si effettua prima la chiamata ricorsiva sul figlio sinistro, poi si visita la radice, e infine si effettua la chiamata ricorsiva sul figlio destro.
- *Visita in postordine:* si effettuano prima le chiamate ricorsive sul figlio sinistro e destro, e infine si visita la radice.

Esempio 3.2 L'ordine di visita dei nodi dell'albero binario in Figura 3.11 (a) mediante le varianti della visita in profondità viste sopra è il seguente:


```
algoritmo visitaBFS(nodo r)
  Coda C
  C.enqueue(r)
  while (not C.isEmpty()) do
    u ←C.dequeue()
    if (u ≠ null) then
      visita il nodo u
      C.enqueue(figlio sinistro di u)
      C.enqueue(figlio destro di u)
```

Figura 3.15 Visita in ampiezza di un albero binario.

- *Visita in preordine:* A L E R B O.
- *Visita simmetrica:* E L R A B O.
- *Visita in postordine:* E R L O B A. □

Visita in ampiezza. Partendo dall'algoritmo di visita generica di Figura 3.12 e rappresentando l'insieme dei nodi aperti S mediante il tipo di dato `Coda` visto nel Paragrafo 3.2, otteniamo la *visita in ampiezza* (*breadth-first search* o *BFS*). L'algoritmo di visita in ampiezza è illustrato in Figura 3.15.

La caratteristica principale della visita in ampiezza è che i nodi vengono visitati *per livelli*: si visita dapprima la radice, poi i figli della radice, poi i figli dei figli, e così via.

Esempio 3.3 L'ordine di visita dei nodi dell'albero binario in Figura 3.11 (a) mediante l'algoritmo di visita in ampiezza in Figura 3.15 è: A L B E R O. □

3.4 Problemi

Problema 3.1 Si descriva una possibile realizzazione indicizzata ed una collegata del tipo di dato `Pila`. Si discuta il tempo richiesto da ognuna delle operazioni supportate e lo spazio richiesto.

Problema 3.2 Si descriva una possibile realizzazione collegata del tipo di dato `Coda`. Si discuta il tempo richiesto da ognuna delle operazioni supportate e lo spazio richiesto. Quali difficoltà aggiuntive sorgerebbero se si volesse invece utilizzare una rappresentazione basata su array?

Problema 3.3 Si consideri la rappresentazione di alberi basata su vettore posizionale descritta nel Paragrafo 3.3.1. In principio, è possibile rappresentare in questo modo anche alberi non completi, semplicemente marcando come inutilizzate le celle che non corrispondono a nodi dell'albero. Quanto spazio potrebbe essere

```
algoritmo profondita(nodo r) → intero
    if (r = null) return 0
    sin ← profondita(figlio sinistro di r)
    des ← profondita(figlio destro di r)
    return 1 + max{sin, des}
```

Figura 3.16 Calcolo della profondità dell'albero con radice r mediante visita in profondità ricorsiva.

necessario per rappresentare un albero non completo con n nodi? Si ponga $d = 2$ e si pensi ad esempio ad un albero a forma di catena dove sempre almeno uno dei due figli sinistro o destro sono assenti.

Problema 3.4 Si discuta una possibile realizzazione delle operazioni del tipo di dato `Albero` descritto in Figura 3.10 usando ciascuna delle rappresentazioni viste nei Paragrafi 3.3.1 e 3.3.2.

Problema 3.5 Si consideri la visita in profondità iterativa illustrata nel Paragrafo 3.3.3. Si noti che la pila potrebbe contenere degli elementi `null` che non corrispondono ad alcun nodo dell'albero binario. Lo spazio usato da questo algoritmo porebbe essere quindi strettamente maggiore di n. Dimostrare tuttavia che esso è ancora $O(n)$.

Problema 3.6 In Figura 3.16 è mostrata una applicazione del procedimento di visita in profondità per calcolare la profondità di un albero. Si scrivano varianti dell'algoritmo per:

1. calcolare il numero di foglie dell'albero;
2. calcolare il grado medio dei nodi dell'albero (numero medio di figli di un nodo);
3. verificare se esiste almeno un nodo dell'albero che abbia un dato contenuto informativo;

3.5 Sommario

In questo capitolo abbiamo discusso alcune semplici tecniche che consentono di rappresentare e manipolare efficientemente collezioni di oggetti nella memoria di un calcolatore. Abbiamo distinto la nozione di *tipo di dato*, che descrive le operazioni di interesse su una collezione di oggetti, da quella di *struttura dati*, che indica piuttosto una particolare organizzazione dei dati che permette di supportare in modo efficiente le operazioni desiderate. Le due principali tecniche di rappresentazione che abbiamo discusso sono quella *indicizzata*, basata su array, e quella *collegata*, basata su record e puntatori. Array, record e puntatori sono i "mattoni" con cui possiamo costruire ogni struttura dati. Abbiamo visto che essi hanno

Figura 3.17 Edward Hicks, *L'Arca di Noè*, 1846.

proprietà complementari, e che a seconda del tipo di operazione da realizzare una tecnica può risultare più vantaggiosa dell'altra. Ad esempio, la consecutività degli indici permette di trovare velocemente un oggetto usando ricerca binaria in un array ordinato, ma preservare l'ordinamento dell'array a fronte di inserimenti o cancellazioni di elementi diventa difficoltoso. Al contrario, cercare dei dati in una lista basata su record e puntatori può essere costoso, mentre aggiungere o togliere oggetti dinamicamente è invece molto semplice. Gli array sono quindi particolarmente adatti per mantenere un numero prefissato di oggetti a cui è necessario accedere in modo impredicibile, mentre le liste collegate funzionano bene se gli oggetti sono inseriti e cancellati frequentemente e gli accessi alla collezione sono sequenziali.

Nella seconda parte del capitolo, abbiamo considerato alcuni classici tipi di dato, come le *pile*, le *code* e gli *alberi*, discutendone possibili realizzazioni. Pile e code forniscono operazioni molto semplici che permettono di mantenere collezioni di oggetti secondo discipline di accesso del tipo: "l'ultimo inserito sarà il primo ad essere cancellato" (pila), o "il primo inserito sarà il primo ad essere cancellato (coda)". Gli alberi sono invece un tipo di dato particolarmente adatto a mantenere relazioni gerarchiche tra oggetti. Abbiamo visto vari modi di rappresentarli usando strutture indicizzate e collegate, e abbiamo discusso alcuni classici algoritmi di visita, che permettono di esplorarne il contenuto in modo sistematico.

3.6 Note bibliografiche

Secondo Knuth [3], che riporta molte informazioni interessanti sulla storia delle strutture dati elementari, l'uso dei puntatori risale ai tempi dei primi calcolatori elettronici, ed appare già in linguaggi come l'IPL-II, progettato nel 1956 da A. Newell, J. C. Shaw e H. A. Simon. La storia delle code ha radici profonde nelle attività umane. Una antica coda appare già nella Bibbia, come illustrato in

Figura 3.17. Sebbene la nozione di pila sia esistita in matematica ancora prima dell'avvento dell'informatica, l'uso delle pile come strutture dati elementari appare ad esempio già negli anni '50 [1]. Secondo Cormen et al. [2], il linguaggio A-1 progettato da G.M. Cooper nel 1951 usava alberi binari per rappresentare formule algebriche. Gonnet [4] riporta nel suo libro risultati sperimentali sulle prestazioni delle operazioni di molte strutture dati.

Riferimenti bibliografici

[1] A. W. Burks, D. W. Warren, and J. B. Wright. *An analysis of a logical machine using parenthesis-free notation*. Mathematical Tables and Other Aids to Computation, 8(46):53-57, Aprile 1954.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[3] D. E. Knuth. *Fundamental Algorithms*, Volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968. Second edition, 1973.

[4] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.

4

Ordinamento

In principio dunque fu Caos, poi Gaia dall'ampio seno, sicura ed eterna sede per tutti gli immortali che abitano la nevosa vetta dell'Olimpo, e Tartaro, denso di nebbia, negli anfratti della terra dalle ampie vie. Vi fu poi Eros, il più bello tra gli immortali, che a tutti gli uomini e gli dei dona saggi consigli nel petto e nel cuore. Da Caos nacquero Erebo e la nera Notte, che, unita a Erebo in amore, concepì Etere e Giorno.

(Esiodo, Teogonia)

Sappiamo, dallo studio della ricerca binaria e probabilmente anche dalla nostra esperienza quotidiana, che mantenere oggetti in modo ordinato ne rende poi più semplice e veloce la ricerca. In questo capitolo affronteremo quindi il problema dell'ordinamento, che si presenta molto spesso in informatica, e descriveremo algoritmi efficienti per ordinare, ad esempio, una lista di nomi alfabeticamente, oppure un insieme di numeri, o ancora un insieme di compiti d'esame in base al cognome dello studente.

Per generalità, assumiamo di avere un insieme di oggetti da ordinare rispetto ad una particolare relazione d'ordine. In prima approssimazione immaginiamo anche che gli oggetti siano distinti: vedremo poi che questa non è una restrizione grave e che molti degli algoritmi che studieremo possono facilmente gestire copie multiple dello stesso oggetto. Esistono numerosi algoritmi di ordinamento. Come introduzione, descriveremo tre algoritmi classici, ma non molto efficienti: `insertionSort`, `selectionSort`, `bubbleSort`. Analizzeremo poi in dettaglio altri cinque algoritmi molto più efficienti: `heapSort`, `mergeSort`, `quickSort`, `bucketSort` e `radixSort`. Ognuno degli algoritmi che descriveremo, oltre ad essere di per sé utile, ci aiuterà ad introdurre nuove idee e tecniche algoritmiche.

Che tempi di esecuzione dobbiamo aspettarci dagli algoritmi che risolvono problemi di ordinamento? E, innanzitutto, come misuriamo i tempi? Un modello astratto per il problema dell'ordinamento, che non fa assunzioni sul tipo di oggetti da ordinare, è il cosiddetto *modello basato su confronti*. Diremo che un algoritmo è basato su confronti se utilizza solo confronti tra oggetti, senza fare uso di altre primitive, quali operazioni aritmetiche (ad esempio somme e prodotti), logiche (ad esempio `and` e `or`) o altro (ad esempio operazioni di shift). Anche se può sembrare restrittivo, in realtà il modello dei confronti è abbastanza generale per catturare le proprietà degli algoritmi di ordinamento più noti. In un modello basato su confronti, è naturale prendere come unità di misura il numero di confronti

n	10	100	1000	10^6	10^9
$n \log_2 n$	≈ 33	≈ 665	$\approx 10^4$	$\approx 2 \cdot 10^7$	$\approx 3 \cdot 10^{10}$
n^2	100	10^4	10^6	10^{12}	10^{18}

Tabella 4.1 Confronto tra n, $n \log_2 n$, e n^2.

eseguito dall'algoritmo. Potremmo anche considerare altri tipi di operazioni, come il numero di spostamenti dei dati in memoria, ma non è difficile convincersi che il numero di confronti sia l'operazione dominante.

Sia n il numero di oggetti da ordinare. Supporremo che gli oggetti siano contenuti in un array A che gli algoritmi restituiscono ordinato al termine dell'esecuzione, ed assumeremo che A sia indicizzato con numeri da 1 a n. Gli algoritmi di ordinamento che descriveremo hanno tempi di esecuzione diversi in funzione di n, ma ci sono due tempi tipici: `quickSort` nel caso medio, `mergeSort`, e `heapSort` richiedono $O(n \log n)$ confronti, mentre `quickSort` nel caso peggiore, `insertionSort`, `selectionSort`, e `bubbleSort` richiedono $O(n^2)$ confronti. Vedremo che $O(n \log n)$ è quanto di meglio possiamo sperare di ottenere (in un modello basato su confronti), mentre $O(n^2)$ è l'andamento peggiore: corrisponde infatti a fare tutti i possibili confronti tra coppie di oggetti. La Tabella 4.1 mostra che $n \log n$ è di gran lunga preferibile a n^2 anche per valori piccoli di n. Quindi, anche se bisogna ordinare liste con pochi elementi, è conveniente usare algoritmi efficienti piuttosto che algoritmi lenti.

4.1 Una delimitazione inferiore al numero di confronti

Come abbiamo visto nel Paragrafo 2.3 del Capitolo 2, possiamo caratterizzare la complessità di un problema dimostrando, tramite argomentazioni matematiche, una delimitazione inferiore alla quantità di una certa risorsa di calcolo necessaria per risolvere il problema. In questo paragrafo dimostreremo una delimitazione inferiore $\Omega(n \log n)$ al numero di confronti richiesti nel caso peggiore per ordinare n elementi. Sottolineiamo che la nostra dimostrazione vale nel caso peggiore, ma con tecniche più sofisticate si può dimostrare che essa rimane vera anche nel caso medio. Osserviamo inoltre che nei Paragrafi 4.6 e 4.7 vedremo algoritmi in grado di ordinare in tempo $O(n)$: questo non è affatto in contraddizione con la delimitazione inferiore che deriveremo in questo paragrafo, poiché gli algoritmi lineari non sono basati su confronti, ma sfruttano particolari proprietà dei dati in ingresso.

Supponiamo ora di avere algoritmi di ordinamento che esaminano i dati effettuando unicamente confronti tra coppie di oggetti. Assumiamo di avere un qualunque algoritmo di ordinamento $\mathcal{A}$ sul cui funzionamento non sappiamo nulla, se non che opera esclusivamente tramite confronti. Sotto questa ipotesi, dimostreremo che il numero di confronti richiesto da $\mathcal{A}$ per ordinare n oggetti deve essere

nel caso peggiore almeno pari a $\Omega(n \log n)$. Poiché $\mathcal{A}$ è un algoritmo generico, la delimitazione sarà vera per *ogni* algoritmo basato su confronti.

4.1.1 Alberi di decisione

Dati un algoritmo generico $\mathcal{A}$ che ordina ricorrendo a confronti e un valore n che rappresenta il numero di elementi da ordinare, definiamo un albero, detto *albero di decisione*, che illustra le diverse sequenze di confronti che $\mathcal{A}$ potrebbe fare su istanze di dimensione n. Ogni nodo dell'albero rappresenta le posizioni degli elementi coinvolti nel confronto, e ogni cammino nell'albero descrive la sequenza di confronti ed il loro esito per una particolare esecuzione dell'algoritmo. Ogni nodo ha due figli, che rappresentano il comportamento dell'algoritmo in funzione dell'esito del confronto in quel nodo.

Un possibile albero di decisione per $n = 3$ è mostrato in Figura 4.1. L'albero nell'esempio descrive un algoritmo in cui il primo confronto è sempre tra l'oggetto in posizione uno e l'oggetto in posizione due nella sequenza di oggetti da ordinare (usiamo la notazione "1:2" per denotare questo). Se l'oggetto in posizione uno è minore oppure uguale all'oggetto in posizione due, il successivo confronto avverrà tra l'oggetto in posizione due e l'oggetto in posizione tre nella sequenza ("2:3"). Se l'oggetto in posizione due è minore oppure uguale all'oggetto in posizione tre, possiamo dedurre che la sequenza è già ordinata, e usiamo "1,2,3" per indicare la permutazione che, applicata alla sequenza in ingresso, la ordina in modo crescente. Ma se il secondo oggetto è più grande del terzo, allora sono possibili ancora due casi (ovvero due possibili permutazioni), e per distinguerli siamo costretti a fare un terzo confronto ("1:3").

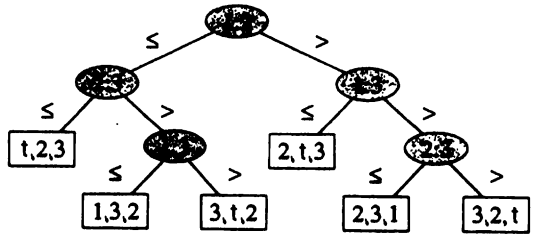


Figura 4.1 Un albero di decisione per l'ordinamento di tre elementi.

Ogni algoritmo che si basa su confronti può essere sempre descritto mediante un albero di decisione, dato che il confronto che sceglie di fare ad un certo punto può dipendere solo dagli esiti dei confronti precedenti. Viceversa, ogni albero di decisione può essere interpretato come un algoritmo di ordinamento: per ogni istanza, seguiamo un cammino a partire dalla radice dell'albero per determinare quali confronti fare e quale permutazione della sequenza produce l'ordinamento. Questo ci suggerisce che gli alberi di decisione non sono altro che un diverso formalismo per descrivere algoritmi di ordinamento basati su confronti. Questo

formalismo è molto utile per scoprire varie proprietà dell'algoritmo originale $\mathcal{A}$, tra cui le seguenti.

Proprietà 4.1 *Per una particolare istanza, la sequenza di confronti eseguiti dall'algoritmo $\mathcal{A}$ su quella istanza è rappresentata da un cammino dalla radice ad una foglia dell'albero di decisione corrispondente all'algoritmo.*

Proprietà 4.2 *Il numero di confronti fatti dall'algoritmo $\mathcal{A}$ nel caso peggiore è pari all'altezza dell'albero di decisione, ovvero, alla lunghezza del più lungo cammino dalla radice ad una foglia.*

Proprietà 4.3 *Il numero di confronti fatti dall'algoritmo $\mathcal{A}$ nel caso medio è pari all'altezza media dell'albero di decisione, ovvero, alla lunghezza media dei cammini dalla radice alle foglie.*

Lemma 4.1 *Un albero di decisione per l'ordinamento di n elementi contiene almeno $n!$ foglie.*

Dimostrazione. Ad ogni foglia dell'albero di decisione non ci sono più confronti da effettuare, e quindi ogni foglia corrisponde ad una soluzione del problema dell'ordinamento. Ogni possibile soluzione corrisponde a sua volta ad una permutazione degli n elementi da ordinare, e quindi l'albero di decisione deve contenere un numero di foglie pari almeno al numero di permutazioni distinte degli n elementi da ordinare, ovvero $n!$ foglie. □

4.1.2 La delimitazione inferiore nel caso peggiore

In base alla Proprietà 4.2, per determinare il numero di confronti fatti dall'algoritmo $\mathcal{A}$ nel caso peggiore dobbiamo determinare l'altezza dell'albero di decisione, ovvero la lunghezza del cammino più lungo.

Lemma 4.2 *Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli e sia k il numero delle sue foglie. L'altezza di T è almeno $\log_2 k$.*

Dimostrazione. Denotiamo con $h(k)$ l'altezza di un albero binario con k foglie e dimostriamo per induzione che $h(k) \geq \log_2 k$. La base dell'induzione è banale: l'albero binario con una sola foglia ha altezza $0 \geq \log_2 1$. Dimostriamo ora il passo induttivo. Assumiamo per ipotesi induttiva che $h(i) \geq \log_2 i$ per $i \leq k-1$. Poiché uno dei due sottoalberi ha almeno metà delle foglie, si ha

$$h(k) \geq 1 + h\left(\frac{k}{2}\right)$$

Per l'ipotesi induttiva

$$h\left(\frac{k}{2}\right) \geq \log_2\left(\frac{k}{2}\right)$$

e quindi

$$h(k) \geq 1 + h\left(\frac{k}{2}\right) \geq 1 + \log_2\left(\frac{k}{2}\right) = 1 + \log_2 k - \log_2 2 = \log_2 k$$

come volevamo dimostrare. □

Siamo ora pronti a dimostrare che il lower bound alla complessità del problema dell'ordinamento è $\Omega(n \log n)$ nel modello basato su confronti.

Teorema 4.1 *Il numero di confronti necessari per ordinare n elementi nel caso peggiore è $\Omega(n \log n)$.*

Dimostrazione. In base al Lemma 4.1, il numero di foglie in un albero di decisione è almeno $n!$. Per la Proprietà 4.2, il numero di confronti necessari per ordinare n elementi nel caso peggiore è pari all'altezza dell'albero, che è $\geq \log n!$ per il Lemma 4.2. Per avere una stima accurata del valore di $\log n!$, possiamo usare le disuguaglianze di De Moivre-Stirling (vedi Paragrafo 17.4 in Appendice):

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n-1}\right)$$

che valgono per n sufficientemente grande. Quindi:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

da cui

$$\log n! \approx n \log n - 1.4427n + \frac{1}{2}\log n + 0.826$$

Questo dimostra che il numero di confronti necessari per ordinare n elementi è $\Omega(n \log n)$, da cui segue la delimitazione inferiore nell'enunciato del teorema. □

4.2 Ordinare in tempo quadratico

In questo paragrafo presenteremo tre classici algoritmi di ordinamento che eseguono $O(n^2)$ confronti nel caso peggiore, e che quindi non sono ottimi in base alla delimitazione inferiore dimostrata nel Teorema 4.1. Due di essi, l'ordinamento per inserimento (`insertionSort`) e quello per selezione (`selectionSort`), lavorano incrementalmente, estendendo in maniera progressiva una sottosequenza ordinata finché essa non comprenda tutti gli elementi. L'algoritmo di ordinamento a bolle (`bubbleSort`), scambia invece ripetutamente elementi adiacenti finché l'intera sequenza non risulti ordinata. A parte pochi contatori ed indici, `insertionSort`, `selectionSort` e `bubbleSort` non usano memoria addizionale all'array A: per brevità, diremo che un algoritmo è in grado di ordinare *in loco* se usa solo la memoria necessaria per mantenere gli elementi in ingresso (l'array A) e uno spazio addizionale costante.

```
    algoritmo selectionSort(array A)
1.    for k = 0 to n − 2 do
2.        m ← k + 1
3.        for j = k + 2 to n do
4.            if ( A[j] < A[m] ) then m ← j
5.        scambia A[m] con A[k + 1]
```

Figura 4.2 Ordinamento per selezione.

4.2.1 Ordinamenti incrementali

Il `selectionSort` e l'`insertionSort` sono ottenuti con una tecnica che potremmo definire "incrementale" o "induttiva": supponiamo che i primi k elementi dell'array siano già ordinati, per $0 \leq k < n$, e vediamo come possiamo estendere l'ordinamento a $(k+1)$ elementi. Esistono almeno due possibili approcci:

- `selectionSort`: scegliamo il minimo degli $(n-k)$ elementi non ancora ordinati e lo assegnamo alla posizione $(k+1)$;
- `insertionSort`: prendiamo il $(k+1)$-esimo elemento nell'array e lo inseriamo nella sua posizione corretta rispetto ai primi k elementi già ordinati.

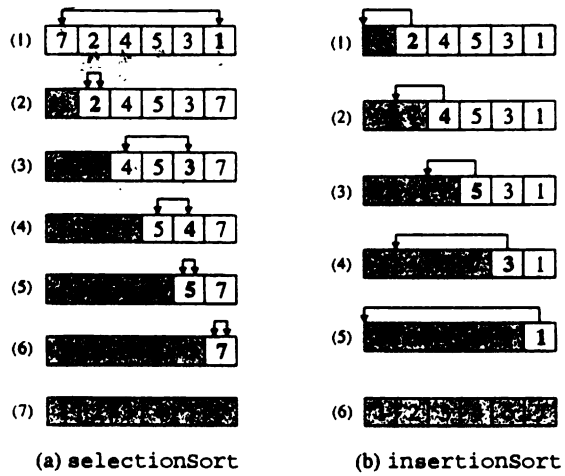
È facile convincersi che entrambi gli approcci estendono l'ordinamento da k a $(k+1)$ elementi. Partendo da $k = 0$, e iterando per n volte, si ottiene quindi la sequenza interamente ordinata. Diamo ora maggiori dettagli sui due algoritmi.

Ordinamento per selezione. Lo pseudocodice per il `selectionSort` è mostrato in Figura 4.2. Nel generico passo, gli elementi $A[k+1]$ ... $A[n]$ non sono ancora ordinati. La ricerca del minimo di questi elementi è effettuata nelle righe da 2 a 4, dopo la cui esecuzione la variabile m contiene l'indice della posizione in cui si trova il minimo cercato. L'inserimento di $A[m]$ nella posizione $(k+1)$ può quindi essere effettuato tramite un semplice scambio. Un esempio di esecuzione dell'algoritmo `selectionSort` è mostrato in Figura 4.3 (a): le frecce indicano gli elementi scambiati ad ogni passata, e l'elemento $A[m]$ è evidenziato in grassetto. Il seguente teorema analizza il tempo di esecuzione dell'algoritmo `selectionSort`.

Teorema 4.2 *L'algoritmo* `selectionSort` *ordina in loco n elementi eseguendo nel caso peggiore $\Theta(n^2)$ confronti.*

Dimostrazione. L'estrazione del minimo (righe 2–4) richiede $(n-k-1)$ confronti. Poiché il ciclo esterno viene eseguito $n-1$ volte, il numero totale di confronti è dato da

$$\sum_{k=0}^{n-2}(n-k-1) = \sum_{i=1}^{n-1} i = \Theta(n^2)$$



(a) `selectionSort` (b) `insertionSort`

Figura 4.3 Esecuzione degli algoritmi `selectionSort` e `insertionSort`.

come è facile verificare con il cambiamento di variabile $i = (n-k-1)$ ed utilizzando la serie aritmetica (vedi Paragrafo 17.2 in Appendice). □

Ordinamento per inserimento. La Figura 4.4 mostra lo pseudocodice dell'algoritmo `insertionSort`. Nel generico passo, gli elementi $A[1]$ ... $A[k]$ sono già ordinati e l'elemento $x = A[k+1]$ è inserito nella posizione che gli compete. In particolare, le righe 3 e 4 individuano la posizione j del più piccolo elemento maggiore di x. La riga 6 sposta a destra di una posizione gli elementi $A[j]$ ... $A[k]$, in modo da creare spazio per l'inserimento di x. Osserviamo che se x è maggiore di tutti gli elementi $A[1]$ ... $A[k]$, risulta $j = k+1$ e lo spostamento non deve essere eseguito. Un esempio di esecuzione dell'algoritmo `insertionSort` è illustrato in Figura 4.3 (b): la freccia indica la posizione in cui va inserito l'e-

```
    algoritmo insertionSort(array A)
1.    for k = 1 to n − 1 do
2.        x ← A[k + 1]
3.        for j = 1 to k + 1 do
4.            if ( A[j] > x ) then break
5.        if ( j < k + 1 ) then
6.            for t = k downto j do A[t + 1] ← A[t]
7.            A[j] ← x
```

Figura 4.4 Ordinamento per inserimento.

```
    algoritmo bubbleSort(array A)
1.    for i = 1 to (n − 1)
2.       for j = 2 to (n − i + 1)
3.          if ( A[j − 1] > A[j] ) then scambia A[j − 1] e A[j]
4.       if ( non ci sono stati scambi ) then break
```

Figura 4.5 Ordinamento a bolle.

lemento $A[k+1]$, che è evidenziato in grassetto. Il seguente teorema analizza il tempo di esecuzione dell'algoritmo `insertionSort`.

Teorema 4.3 *L'algoritmo* `insertionSort` *ordina in loco n elementi eseguendo nel caso peggiore* $\Theta(n^2)$ *confronti.*

Dimostrazione. Le righe 3 e 4 individuano la posizione j in cui $A[k+1]$ va inserito, eseguendo al più $(k+1)$ confronti. Le righe 6 e 7 non eseguono invece alcun confronto. Poiché il ciclo esterno viene eseguito $(n-1)$ volte, il numero totale di confronti è dato da

$$\sum_{k=1}^{n-1}(k+1) = \left(\sum_{k=1}^{n-1} k\right) + (n-1) = \Theta(n^2)$$

come è facile verificare utilizzando la serie aritmetica (vedi Paragrafo 17.2 in Appendice). □

Entrambi gli algoritmi incrementali ordinano *in loco*, ma sono molto lenti: $O(n^2)$ è infatti il peggior tempo di esecuzione asintotico che si possa ottenere, poiché corrisponde ad eseguire tutti i possibili confronti tra le

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

coppie di elementi. Vedremo però nel Paragrafo 4.3 che con l'uso di opportune strutture dati anche l'approccio incrementale può portare ad algoritmi efficienti.

4.2.2 Ordinamento a bolle

L'algoritmo di ordinamento a bolle (`bubbleSort`) opera eseguendo una serie di scansioni dell'array: in ogni scansione sono confrontate coppie di elementi adiacenti, e viene effettuato uno scambio se i due elementi non rispettano l'ordinamento. Se durante una scansione non viene effettuato nessuno scambio, l'array è ordinato e l'algoritmo termina. Lo pseudocodice è mostrato in Figura 4.5, ed un esempio di esecuzione dell'algoritmo `bubbleSort` è illustrato in Figura 4.6.

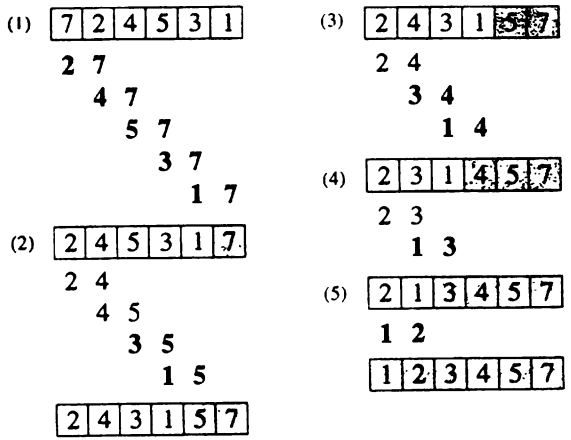


Figura 4.6 Esecuzione dell'algoritmo `bubbleSort`.

Ad ogni confronto nella riga 3, il massimo tra gli elementi $A[j-1]$ e $A[j]$ viene portato nella posizione j. Poiché nella i-esima scansione sono confrontate le prime $(n-i)$ coppie di elementi adiacenti, il massimo degli elementi $A[1]...A[n-i+1]$ viene portato in posizione $(n-i+1)$. È ora facile verificare che il `bubbleSort` mantiene la seguente invariante:

Lemma 4.3 *Dopo l'i-esima scansione, gli elementi* $A[n-i+1]$ ... $A[n]$ *sono correttamente ordinati e occupano la loro posizione definitiva, ovvero: per ogni* h, k, *con* $1 \le h \le k$ *e* $n-i+1 \le k \le n$, *risulta* $A[h] \le A[k]$.

Dimostrazione. Procediamo per induzione sul numero i dell'iterazione. Per $i = 1$, ovvero dopo la prima scansione, il massimo dell'intero array A avrà raggiunto la posizione n: il passo base è pertanto banalmente verificato.

Supponiamo ora che l'enunciato valga dopo le prime $(i-1)$ scansioni e verifichiamo che continua a valere anche dopo l'i-esima scansione. L'ipotesi induttiva garantisce che, per ogni h, k, con $1 \le h \le k$ e $n-i+2 \le k \le n$, risulta $A[h] \le A[k]$. Dopo la i-esima scansione, il massimo degli elementi $A[1]...A[n-i+1]$ avrà raggiunto la posizione $(n-i+1)$, e quindi, per ogni $h < n-i+1$, risulta anche $A[h] \le A[n-i+1]$. Combinando le due osservazioni, si può facilmente verificare la validità del passo induttivo. □

L'idea di fondo dell'algoritmo `bubbleSort` è dunque simile a quella del `selectionSort`, ma ad ogni passo la selezione del massimo avviene in modo più complicato, attraverso scambi tra elementi adiacenti. In pratica, si cerca di spingere gli oggetti più "grandi" verso la fine dell'array, facendoli risalire verso l'alto come bolle, da cui il nome `bubbleSort` (ovvero "ordinamento a bolle"). Siamo ora pronti ad analizzare le prestazioni del `bubbleSort`.

Teorema 4.4 *L'algoritmo* `bubbleSort` *ordina in loco n elementi eseguendo nel caso peggiore $\Theta(n^2)$ confronti.*

Dimostrazione. Per il Lemma 4.3, dopo al più $(n-1)$ cicli l'array è ordinato correttamente: l'i-esimo ciclo infatti trova l'i-esimo elemento più grande nell'array e lo fa salire in posizione $(n-i+1)$. Da ciò segue la correttezza. Dallo pseudocodice in Figura 4.5 si deduce che nel caso peggiore nella i-esima scansione vengono eseguiti $(n-i)$ confronti. Il numero totale di confronti è quindi dato da

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

come è facile verificare con il cambiamento di variabile $j = n - i$ ed utilizzando la serie aritmetica (vedi Paragrafo 17.2 in Appendice). □

4.3 Heapsort

L'algoritmo `heapSort` usa lo stesso approccio incrementale su cui si basa il `selectionSort`, ma esegue nel caso peggiore un numero inferiore di confronti grazie all'uso di strutture dati efficienti per selezionare il minimo ad ogni iterazione. Per convenienza, nella descrizione dell'algoritmo `heapSort` lavoreremo selezionando gli elementi dal più grande al più piccolo, anziché dal più piccolo al più grande come abbiamo fatto per l'algoritmo `selectionSort` in Figura 4.2. Il problema cruciale per rendere più veloce l'ordinamento per selezione sembra quello di progettare una struttura dati H su cui eseguire efficientemente le seguenti operazioni:

- dato un array A, generare velocemente un'istanza della struttura dati H;
- trovare il più grande oggetto in H;
- cancellare il più grande oggetto in H.

Ci sono molte strutture dati per questo problema: nei Capitoli 6 e 8 vedremo rispettivamente alberi di ricerca e code con priorità. Per ora ci limitiamo a descrivere una struttura dati *ad hoc* chiamata *heap*.

4.3.1 Struttura dati heap

Definiamo innazitutto un *heap* associato ad un insieme L di elementi. Una generalizzazione di questa definizione sarà data nel Paragrafo 8.1 del Capitolo 8, in cui mostreremo anche come implementare altre operazioni, quali l'inserimento di nuovi oggetti, che non considereremo invece in questa sede.

Definizione 4.1 (Heap) *Un* heap *associato a un insieme L di elementi è un albero binario radicato con le seguenti proprietà:*

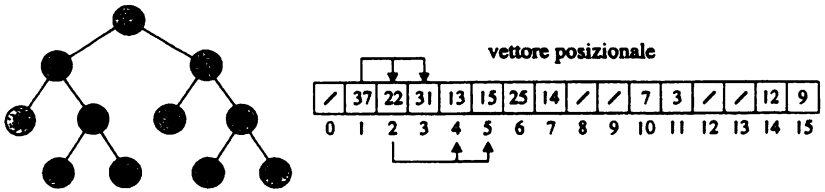


Figura 4.7 Un heap binario e la sua rappresentazione con un vettore posizionale (vedi il Paragrafo 3.3 del Capitolo 3). Le celle inutilizzate sono barrate.

1. Struttura: *l'albero è completo almeno fino al penultimo livello.*
2. Contenuto informativo: *gli elementi di L sono memorizzati nei nodi dell'albero; ogni nodo v memorizza uno ed un solo elemento, che denoteremo con $chiave(v)$, e ogni elemento è memorizzato in un solo nodo.*
3. Ordinamento a heap: *il valore dell'elemento in un nodo è sempre maggiore o uguale al valore degli elementi nei figli.*

Un esempio di heap è mostrato in Figura 4.7. La Proprietà 1 nella Definizione 4.1 implica una limitazione superiore logaritmica sull'altezza dell'albero, come dimostrato nel seguente lemma.

Lemma 4.4 *Un heap con n nodi ha altezza $\Theta(\log n)$.*

Dimostrazione. Sia h il numero di livelli dell'heap. I primi $(h-1)$ livelli sono tutti completi, e il numero di nodi in essi contenuti è pari a

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

trattandosi della serie geometrica di ragione 2 (vedi Paragrafo 17.2 in Appendice). Analogamente, il numero di nodi contenuti in h livelli è $\leq 2^{h+1} - 1$. Poiché deve risultare $2^h - 1 \leq n \leq 2^{h+1} - 1$, prendendo il logaritmo di tutti i membri si ottiene che $h = \Theta(\log n)$. □

Se denotiamo rispettivamente con $sin(v)$ e $des(v)$ il figlio sinistro e destro di un nodo v, possiamo sintetizzare la Proprietà 3 della Definizione 4.1 scrivendo

$$chiave(v) \geq chiave(sin(v))$$

$$chiave(v) \geq chiave(des(v))$$

La proprietà di ordinamento a hesp è quindi qualcosa di simile a quello che succede nella gerarchia di un'organizzazione aziendale: lo stipendio del presidente di un'azienda è tipicamente superiore a quello dei manager, i cui stipendi sono a loro volta superiori rispetto a quelli dei dipendenti. La radice dell'albero conterrà

```
algoritmo fixHeap(nodo v, heap H)
  if ( v è una foglia in H ) then return
  else
     sia u il figlio di v con chiave massima
     if ( chiave(v) < chiave(u) ) then
        scambia chiave(v) e chiave(u)
        fixHeap(u, H)
```

Figura 4.8 La procedura `fixHeap`.

il massimo dell'insieme L, in linea con il fatto il presidente ha di regola lo stipendio più alto. Sottolineiamo che la proprietà dell'ordinamento a heap può anche essere formulata in maniera simmetrica, richiedendo che il valore dell'elemento in un nodo sia sempre inferiore al valore degli elementi nei figli. Nel resto di questo paragrafo adotteremo comunque la prima formulazione, in accordo con la Definizione 4.1.

Cancellazione del massimo. Abbiamo detto che identificare il massimo in un heap è molto semplice: come conseguenza della proprietà di ordinamento a heap combinata con la transitività della relazione di ordinamento il massimo si trova sempre sulla radice dell'albero. Ma come cancellarlo? Supponiamo che il presidente dell'azienda vada in pensione: per sostituirlo, se non vogliamo nominare una persona esterna all'azienda, dobbiamo promuovere qualcuno nell'organizzazione interna dell'azienda. Per mantenere la proprietà heap, dobbiamo scegliere la persona col più alto stipendio, che deve essere per forza uno dei suoi dipendenti diretti (e, tra questi, quello con lo stipendio più elevato). La promozione di questa persona crea un vuoto, che possiamo riempire nello stesso modo. Sfortunatamente, questo approccio potrebbe inficiare la Proprietà 1 di struttura, poiché la foglia fisicamente rimossa al termine del processo potrebbe trovarsi sul penultimo livello: per convincervene, provate ad usare questo approccio per cancellare il massimo dall'heap di Figura 4.7. Possiamo comunque risolvere facilmente il problema come segue: scegliamo una qualunque foglia v sull'ultimo livello, copiamo l'elemento $chiave(v) = k$ nella radice, ed eliminiamo fisicamente il nodo v. La Proprietà 1 ora è certamente mantenuta, ma la Proprietà 3 di ordinamento a heap potrebbe non esserlo. Per ripristinarla usiamo una procedura `fixHeap`, che spinge verso in basso l'elemento k nella radice, tramite ripetuti scambi di nodi, come mostrato in Figura 4.8.

Si noti che nel caso in cui il nodo v abbia due figli, l'elemento massimo tra i due viene promosso al livello superiore, ripristinando localmente la proprietà di ordinamento a heap. Il numero di confronti richiesto da questa operazione eguaglia l'altezza dell'albero, che è $O(\log n)$ per il Lemma 4.4. Osserviamo che l'algoritmo è basato esclusivamente su confronti e scambi di chiavi: è quindi immediato implementare questa procedura nel caso in cui l'heap sia rappresentato

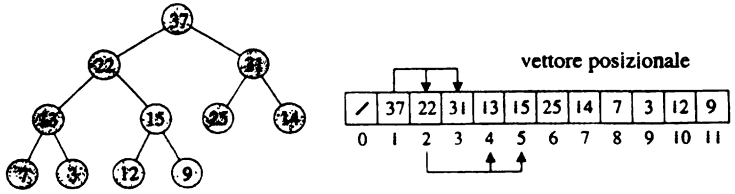


Figura 4.9 Un heap binario con struttura rafforzata ed il corrispondente vettore posizionale.

con un vettore posizionale, come illustrato in Figura 4.7. Ad esempio, scambiare la chiave di un nodo in posizione i nell'array con quella del suo figlio sinistro, equivale a scambiare le chiavi nelle posizioni i e $2i$.

Costruzione di un heap. In questo paragrafo mostreremo come costruire un heap in tempo $O(n)$. Innanzitutto, creiamo un albero binario di taglia e dimensione appropriata in cui mettiamo gli oggetti dell'array A in un ordine qualsiasi. Utilizzando la rappresentazione posizionale, potremmo interpretare lo stesso array A come heap. La struttura dell'albero in questo caso è ancora più particolare di quella descritta nella Definizione 4.1:

Struttura rafforzata: *l'albero è completo fino al penultimo livello e le foglie sull'ultimo livello sono tutte compattate a sinistra.*

La proprietà di struttura rafforzata, illustrata su un esempio in Figura 4.9, ci sarà poi utile per ordinare *in loco* mediante heap. La creazione dell'albero, utilizzando la rappresentazione posizionale, è un'operazione che non richiede alcun confronto. Dobbiamo poi spostare gli oggetti in modo che rispettino la proprietà di ordinamento a heap. A tale scopo, usiamo la tecnica *divide et impera*, che abbiamo già introdotto nel Paragrafo 2.5.3 del Capitolo 2: ricordiamo che questa tecnica si basa sul trovare dei sottoproblemi, risolverli ricorsivamente, e poi ricombinare le loro soluzioni per ottenere la soluzione al problema originario. Nel nostro caso specifico, i sottoproblemi sembrano coincidere in modo naturale con i sottoalberi sinistro e destro. Se trasformiamo in heap ricorsivamente i sottoalberi, otteniamo infatti un albero molto vicino ad un heap: l'unica eccezione potrebbe essere la chiave nella radice, che potrebbe non soddisfare la Proprietà 3. Come possiamo quindi ripristinare la proprietà di ordinamento a heap in tutto l'albero? Con un attimo di riflessione, ci accorgeremo che è sufficiente richiamare proprio sulla radice la procedura `fixHeap` che abbiamo descritto precedentemente.

Lo pseudocodice della procedura `heapify` per la costruzione di un heap è mostrato in Figura 4.10. Ancora una volta, osserviamo che nel caso in cui l'heap sia rappresentato con un vettore posizionale l'implementazione di questa procedura è molto semplice: se la radice di T occupa la posizione i nell'array A che rappresenta l'heap, i sottoalberi sinistro e destro possono infatti essere facilmente identificati specificando le posizioni $2i$ e $(2i+1)$. Riassumiamo le caratteristiche della struttura dati heap nel seguente teorema:

```
algoritmo heapify(albero T)
    if ( T è vuoto ) then return
    heapify(sin(T))
    heapify(des(T))
    fixHeap(radice(T), T)
```

Figura 4.10 La procedura `heapify` per la costruzione di un heap.

Teorema 4.5 *Un heap associato ad un insieme di n elementi può essere costruito in tempo $O(n)$ e supporta la cancellazione del massimo in tempo $O(\log n)$. L'heap può essere rappresentato in loco.*

Dimostrazione. L'unico punto ancora da dimostrare riguarda il tempo per la costruzione dell'heap. Il tempo di esecuzione $T(n)$ della procedura `heapify` è descritto dalla seguente relazione di ricorrenza:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(\log n)$$

dove $O(\log n)$ è il tempo richiesto da `fixHeap`. Possiamo risolvere la ricorrenza usando il Teorema 2.1: essendo $a = b = 2$ e $f(n) = \log n$, otteniamo $T(n) = O(n)$. Quindi `heapify` richiede $O(n)$ confronti. □

4.3.2 Ordinare *in loco* mediante heap

Sintetizziamo l'idea dell'`heapSort` nello pseudocodice di Figura 4.11, la cui analisi ci porta alle seguenti considerazioni. Dato l'array A, se possiamo costruire in tempo t_1 una struttura dati per cui trovare e cancellare il massimo richieda tempo t_2, allora possiamo ordinare in tempo $O(t_1 + n \cdot t_2)$. Usando la struttura dati heap, abbiamo $t_1 = O(n)$ e $t_2 = O(\log n)$, e quindi il numero totale di confronti richiesti da `heapSort` è $O(n + n \log n) = O(n \log n)$.

Nello pseudocodice di Figura 4.11, l'algoritmo non ordina però *in loco*, ma usa due strutture di appoggio H e X. Abbiamo già osservato che l'heap può essere direttamente implementato nell'array A usando la rappresentazione posizionale e sfruttando la proprietà di struttura rafforzata. Ma come possiamo evitare la lista di appoggio X? Consideriamo la prima estrazione del massimo e supponiamo di rimuovere fisicamente non una qualunque foglia sull'ultimo livello (come descritto nel Paragrafo 4.3.1), ma la foglia più a destra. Poiché questa foglia occupa la posizione n nell'array A, dopo la sua cancellazione tale posizione sarà libera (l'heap infatti conterrà solo $(n-1)$ elementi) e potrà essere usata per memorizzare il massimo appena estratto. Dopo la seconda estrazione di massimo, l'heap conterrà $(n-2)$ elementi e la posizione $(n-1)$ si renderà disponibile per contenere il secondo elemento più grande. Procedendo in questo modo, la sequenza


```
algoritmo heapSort(array A) → lista
    crea un heap H dall'array A
    sia X una lista vuota
    while ( H non è vuoto ) do
        rimuovi il massimo da H ed aggiungilo in coda a X
    return X
```

Figura 4.11 L'algoritmo `heapSort`.

di elementi ordinati si allungherà da destra verso sinistra nell'array A man mano che l'heap si contrae. Alla fine, quando l'heap è vuoto, l'array A conterrà tutti gli elementi ordinati in modo non decrescente. Un esempio di ordinamento mediante heap è mostrato in Figura 4.12; per semplicità, si assume che l'albero iniziale soddisfi la proprietà di ordinamento a heap e si illustrano solo alcune estrazioni di minimo. Il seguente teorema riassume le prestazioni dell'algoritmo `heapSort`.

Teorema 4.6 *L'algoritmo* `heapSort` *ordina in loco n elementi eseguendo nel caso peggiore $O(n \log n)$ confronti.*

4.4 Mergesort

Donald Knuth, nella sua enciclopedica trattazione di algoritmi di ordinamento e di ricerca [12], attribuisce la prima implementazione del `mergeSort` a John von

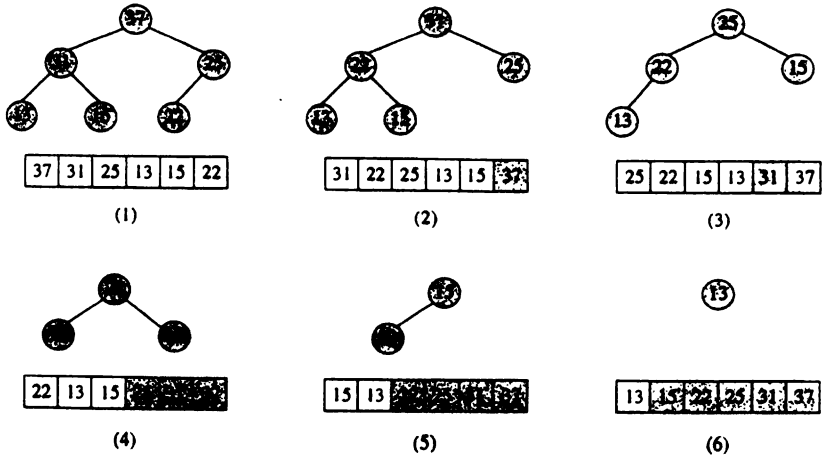


Figura 4.12 Esecuzione dell'algoritmo `heapSort`.

```
algoritmo merge(array A, interi i1, f1 e f2)
    sia X un array ausiliario di lunghezza f2 − i1 + 1
    i ← 1
    i2 ← f1 + 1
    while ( i1 ≤ f1 e i2 ≤ f2 ) do
        if ( A[i1] ≤ A[i2] )
        then X[i] ← A[i1]
            incrementa i ed i1
        else X[i] ← A[i2]
            incrementa i ed i2
    if ( i1 < f1 ) then copia A[i1; f1] alla fine di X
    else copia A[i2; f2] alla fine di X
    copia X in A[i1; f2]
```

Figura 4.13 Fusione di due sequenze ordinate in un'unica sequenza ordinata: le due sottosequenze sono individuate da $A[i_1; f_1]$ ed $A[f_1 + 1; f_2]$, rispettivamente. L'output è restituito in $A[i_1; f_2]$.

Neumann, uno dei pionieri dell'informatica, e la fa risalire al 1945. Si tratta infatti di un algoritmo molto intuitivo, e non è sorprendente che fosse utilizzato già in tempi così remoti (almeno per le discipline informatiche). Il `mergeSort` si basa sulla tecnica di *divide et impera*, che abbiamo già introdotto nel Paragrafo 2.5.3 del Capitolo 2. Vediamo prima il passo di *impera* e poi quello di *divide*.

Impera. Supponiamo di avere due sequenze già ordinate, Per "fonderle" velocemente, potremmo applicare l'algoritmo di Figura 4.13, che estrae ripetutamente il minimo delle due sequenze in ingresso e lo appende in una sequenza di uscita. Dati un array A e due indici $x \leq y$, denotiamo con $A[x; y]$ la porzione dell'array A costituita dagli elementi $A[x]$ ... $A[y]$. Nella nostra implementazione, le due sequenze sono memorizzate consecutivamente nell'array A e corrispondono rispettivamente ad $A[i_1; f_1]$ ed $A[f_1 + 1; f_2]$. La sequenza prodotta dall'esecuzione è contenuta in $A[i_1; f_2]$, ma è necessario usare un array ausiliario X durante la sua costruzione. La procedura `merge` è analizzata nel Lemma 4.5.

Lemma 4.5 *La procedura* `merge` *fonde due sequenze ordinate di lunghezza ℓ_1 ed ℓ_2 eseguendo al più $(\ell_1 + \ell_2 - 1)$ confronti.*

Dimostrazione. Esaminando lo pseudocodice di Figura 4.13, osserviamo che ogni confronto "consuma" un elemento da A. Poiché nel caso peggiore tutti gli elementi, tranne l'ultimo, sono aggiunti alla sequenza X tramite un confronto, avremo in totale $|X| - 1 = \ell_1 + \ell_2 - 1$ confronti. □

Divide. Visto che sappiamo come fondere due sequenze ordinate, potremmo inizialmente dividere l'array in due sottoarray di taglia bilanciata, ordinarli ricorsivamente e poi fonderli, come specificato dallo pseudocodice di Figura 4.14. L'albero delle chiamate ricorsive relativo ad una particolare esecuzione dell'algoritmo


```
algoritmo mergeSort(array A, indici i e f)
    if ( i ≥ f ) then return
    m ← (i + f)/2
    mergeSort(A, i, m)
    mergeSort(A, m + 1, f)
    merge(A, i, m, f)
```

Figura 4.14 L'algoritmo `mergeSort`.

`mergeSort` è mostrato in Figura 4.15. Notiamo che l'algoritmo `mergeSort` sembra molto più semplice da programmare rispetto a `heapSort`. La seguente analisi dimostra che è altrettanto efficiente.

Teorema 4.7 *L'algoritmo* `mergeSort` *ordina un array di lunghezza n eseguendo nel caso peggiore $\Theta(n \log n)$ confronti.*

Dimostrazione. In base al Lemma 4.5, il numero di confronti per fondere due sequenze ordinate di lunghezza totale n nel caso peggiore è $(n - 1)$. Per semplicità di analisi, assumiamo che n sia una potenza di 2. Poiché l'algoritmo `mergeSort` divide il problema in due sottoproblemi di taglia bilanciata, risolvendo questi ultimi ricorsivamente, otteniamo la seguente relazione di ricorrenza per il numero di confronti $C(n)$:

$$C(n) \leq n - 1 + 2C\left(\frac{n}{2}\right)$$

Applicando il secondo caso del Teorema 2.1 del Capitolo 2, sappiamo che la soluzione è $\Theta(n \log n)$. Se desideriamo una limitazione più accurata, possiamo usare il metodo di iterazione introdotto nel Paragrafo 2.5.1 del Capitolo 2, ottenendo la seguente catena di disuguaglianze:

$$C(n) \leq \sum_{i=0}^{\log n} 2^i \left(\frac{n}{2^i} - 1\right) = \sum_{i=0}^{\log n} (n - 2^i) = n(\log n + 1) - (2n - 1) =$$

$$= n \log n - n + 1 = \Theta(n \log n)$$

Infatti, l'albero della ricorsione ha altezza $\log n$, in ogni chiamata al livello i, $0 \leq i \leq \log n$, eseguiamo $(n/2^i - 1)$ confronti per la fusione, ed abbiamo 2^i chiamate nell'i-esimo livello. □

Diversamente da `heapSort`, l'algoritmo `mergeSort` non ordina però *in loco*: infatti la procedura `merge` usa una memoria ausiliaria di dimensione pari alla lunghezza delle sequenze da fondere. La nostra implementazione del `mergeSort` ha pertanto un'occupazione di memoria pari a $2n$.

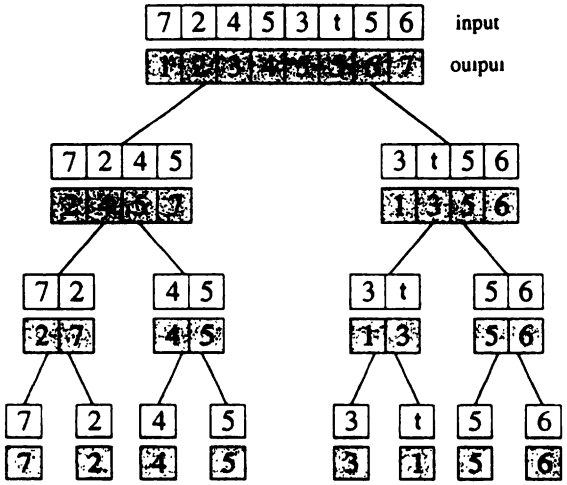


Figura 4.15 Albero delle chiamate ricorsive dell'algoritmo `mergeSort`: per ciascuna chiamata è mostrato il contenuto dell'array sia in ingresso che in uscita.

4.5 Quicksort

Anche l'algoritmo `quickSort` che descriveremo in questo paragrafo segue un approccio di tipo *divide et impera*, partizionando gli elementi in due sequenze che vengono ordinate ricorsivamente e poi ricombinate insieme. Differentemente dal `mergeSort`, però, in questo caso la parte più complessa è la fase di suddivisione (*divide*). Osserviamo che il `mergeSort` funziona indipendentemente da come si partiziona l'array, fintantoché le due sottosequenze abbiano dimensione $n/2$. Ma se potessimo fare questa divisione in modo tale che tutti gli elementi di una sequenza siano più piccoli di tutti gli elementi dell'altra, la parte di fusione (*impera*) sarebbe banale: basterebbe soltanto concatenare le due sequenze! Il `quickSort`, il cui pseudocodice è mostrato in Figura 4.16, usa esattamente quest'idea.

Divide: scegli un elemento x della sequenza (che chiameremo *perno*), partiziona la sequenza in due sottosequenze contenenti rispettivamente elementi $\leq x$ e $> x$, ed ordina ricorsivamente le due sottosequenze.

Impera: concatena le sottosequenze ordinate.

Sebbene l'algoritmo sia semplice, l'analisi è piuttosto sofisticata, come vedremo nel prossimo paragrafo. Prima di procedere, soffermiamoci sull'implementazione delle righe 3 e 4, ovvero sull'operazione di partizione. È infatti semplice partizionare usando memoria ausiliaria proporzionale al numero di elementi: basta eseguire una scansione di A e copiare gli elementi opportunamente in uno dei due array A_1 o A_2. Viene spontaneo chiedersi se sia possibile lavorare *in loco*, così da evitare l'uso di A_1 e A_2. La risposta è affermativa, sebbene partizionare

algoritmo `quickSort`(*array* A)
1. scegli un elemento x in A
2. partiziona A rispetto ad x calcolando:
3. $A_1 = \{ y \in A : y \leq x \}$
4. $A_2 = \{ y \in A : y > x \}$
5. **if** ($|A_1| > 1$) **then** `quickSort`(A_1)
6. **if** ($|A_2| > 1$) **then** `quickSort`(A_2)
7. copia la concatenazione di A_1 ed A_2 in A

Figura 4.16 L'algoritmo `quickSort`.

in loco introduca alcuni tecnicismi. Si scorre l'array avanzando in parallelo da sinistra verso destra e da destra verso sinistra, mettendo gli elementi minori del perno nella parte inferiore e quelli maggiori nella parte superiore dell'array. Per far ciò, è sufficiente mantenere due indici sup ed `inf`, come mostrano lo pseudocodice di Figura 4.17 e l'esecuzione dell'algoritmo in Figura 4.18. La procedura `partition` mantiene la seguente invariante:

Proprietà 4.4 *In ogni istante, gli elementi* $A[i] \ldots A[inf-1]$ *sono minori o uguali al perno, mentre gli elementi* $A[sup+1] \ldots A[f]$ *sono maggiori del perno.*

Grazie alla Proprietà 4.4, quando l'indice `inf` supera l'indice sup si ha la partizione desiderata. Poiché ogni elemento è confrontato con il perno una sola volta, calcolare la partizione degli elementi intorno al perno richiede $(n-1)$ confronti. L'albero delle chiamate ricorsive relativo ad una particolare esecuzione dell'algoritmo `quickSort` è mostrato in Figura 4.19: la figura illustra che, diversamente dal `mergeSort`, l'uso della procedura `partition` non garantisce che la partizione individuata ad ogni passo ricorsivo sia bilanciata. Vedremo chiaramente le implicazioni di questo fatto sul tempo di esecuzione nel corso dell'analisi.

4.5.1 Analisi di Quicksort

L'algoritmo `quickSort` ordina un array A di dimensione n suddividendolo in due sottoarray, A_1 ed A_2, di dimensioni a e b tali che $a+b=n-1$, ed ordinando ricorsivamente A_1 ed A_2. Poiché la partizione richiede $(n-1)$ confronti, la relazione di ricorrenza che descrive il numero di confronti $C(n)$ del `quickSort` è la seguente:

$$C(n) = n - 1 + C(a) + C(b)$$

Osserviamo che $a+b=n-1$, poiché il perno non è incluso nei sottoarray su cui si procede ricorsivamente, come mostrato nello pseudocodice del `quickSort` in Figura 4.17. Nel caso peggiore x potrebbe essere l'elemento più piccolo (o più grande) di A. In questo caso $a=0$, $b=n-1$, e la relazione di ricorrenza diventa $C(n) = n-1+C(n-1) = O(n^2)$, come abbiamo osservato nell'Esempio 2.6

```
procedura partition(array A, indici i e f) → indice
    x ← A[i]
    inf ← i
    sup ← f + 1
    while ( true ) do
        do inf ← inf + 1 while ( A[inf] ≤ x )
        do sup ← sup − 1 while ( A[sup] > x )
        if ( inf < sup ) then scambia A[inf] ed A[sup]
        else break
    scambia A[i] ed A[sup]
    return sup

algoritmo quickSort(array A, indici i e f)
    if ( i ≥ f ) then return
    m ← partition(A, i, f)
    quickSort(A, i, m − 1)
    quickSort(A, m + 1, f)
```

Figura 4.17 La procedura `partition` per partizionare *in loco*: `partition` restituisce la posizione del perno nella sottosequenza appena partizionata. Uso di `partition` nell'algoritmo `quickSort`.

del Capitolo 2. Nel caso peggiore il `quickSort` sembra quindi un algoritmo non efficiente. Come possiamo renderlo più efficiente? L'idea consiste nell'usare la randomizzazione, che abbiamo introdotto nel Paragrafo 2.6 del Capitolo 2.

Supponiamo di scegliere il perno x come il k-esimo elemento di A, dove k è scelto *a caso*: ogni valore di a è quindi ugualmente probabile. Come spiegato nel Paragrafo 2.6, per calcolare il numero atteso di confronti dobbiamo calcolare la somma dei tempi di esecuzione pesandoli in base alle probabilità di fare una certa scelta casuale. Nel nostro caso, la scelta casuale è il valore di k e le probabilità di scegliere il k-esimo elemento come perno, per ogni k, $1 \leq k \leq n$, è $1/n$. Otteniamo pertanto la seguente relazione di ricorrenza:

$$C(n) = \sum_{a=0}^{n-1} \frac{1}{n}(n - 1 + C(a) + C(n - a - 1))$$

Per semplificarla, notiamo che i termini $C(a)$ e $C(n - a - 1)$ danno luogo alla stessa sommatoria, e quindi:

$$C(n) = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a) \qquad (4.1)$$

Lemma 4.6 *La relazione di ricorrenza (4.1) ha soluzione* $C(n) \leq 2n \log n$.

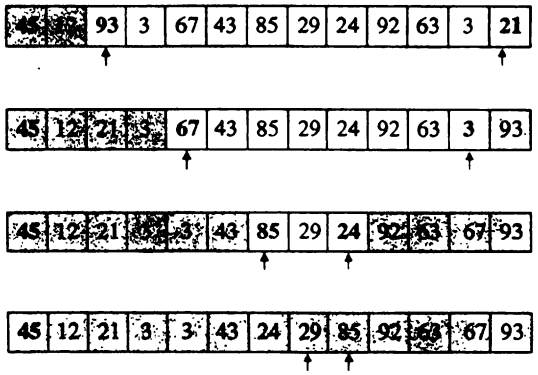


Figura 4.18 Esecuzione della procedura `partition`.

Dimostrazione. Usiamo il metodo di sostituzione introdotto nel Paragrafo 2.5.2 del Capitolo 2, dimostrando induttivamente che $C(n) \leq \alpha n \log n$ per una opportuna costante α. Dall'analisi emergerà che $\alpha = 2$. Il passo base è banale, poiché $C(1) = 0 = \alpha(1 \log 1)$ per ogni α. Assumiamo ora che la soluzione sia $C(i) \leq \alpha \cdot i \log i$ per qualche costante α e per $i < n$. Sostituendo nella relazione di ricorrenza (4.1) si ottiene:

$$C(n) = n - 1 + \sum_{i=0}^{n-1} \frac{2}{n} C(i) \leq n - 1 + \sum_{i=0}^{n-1} \frac{2}{n} \alpha \cdot i \log i =$$

$$= n - 1 + \frac{2\alpha}{n} \sum_{i=2}^{n-1} i \log i \leq n - 1 + \frac{2\alpha}{n} \int_2^n x \log x \, dx$$

per definizione di somme integrali. Usando il metodo di integrazione per parti ed alcune manipolazione algebriche otteniamo:

$$C(n) \leq n - 1 + \frac{2\alpha}{n} \left(n^2 \frac{\log n}{2} - \frac{n^2}{4} - 2 \ln 2 + 1 \right) =$$

$$= n - 1 + \alpha \cdot n \log n - \alpha \frac{n}{2} - O(1) \leq \alpha \cdot n \log n$$

quando $n - 1 < \alpha \cdot (n/2)$. In particolare, l'ultima disuguaglianza vale per $\alpha \geq 2$, da cui possiamo concludere che $C(n) \leq 2n \log n$. □

Riassumiamo le prestazioni dell'algoritmo `quickSort` nel seguente teorema.

Teorema 4.8 *L'algoritmo* `quickSort` *randomizzato ordina in loco un array di lunghezza n eseguendo $O(n^2)$ confronti nel caso peggiore. Il numero atteso di confronti è invece $O(n \log n)$.*

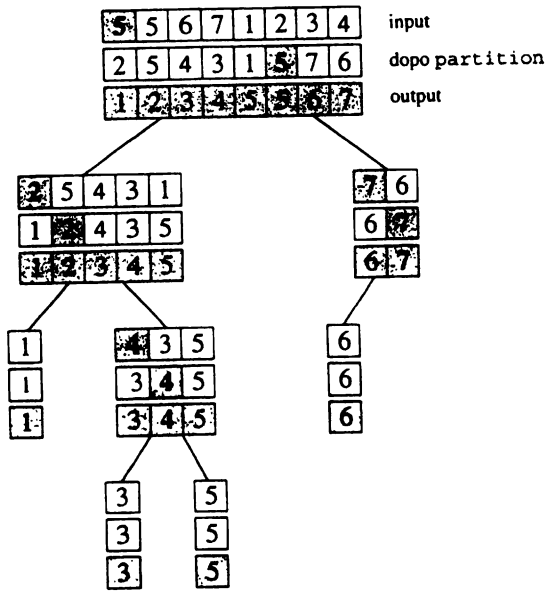


Figura 4.19 Albero delle chiamate ricorsive dell'algoritmo `quickSort`: per ciascuna chiamata è mostrato il contenuto dell'array in ingresso, dopo l'esecuzione di `partition`, e in uscita.

Ribadiamo che il `quickSort` è meno efficiente nel caso peggiore di algoritmi ottimi come `mergeSort` o `heapSort`, e che, proprio per evitare il caso peggiore, abbiamo bisogno di renderlo randomizzato. Ma il `quickSort` è forse tra gli algoritmi di ordinamento più veloci e più usati in pratica: esistono infatti vari modi per migliorarne sostanzialmente le prestazioni, rendendolo competitivo anche con algoritmi ottimi nel caso peggiore. Ad esempio, è stato osservato che selezionare il perno x come il mediano di tre valori scelti a caso porta in pratica a sostanziali benefici sul tempo di esecuzione. Molti dettagli sull'implementazione del `quickSort` sono descritti in [13].

4.6 Bucketsort

Finora abbiamo descritto vari algoritmi per ordinare in tempo $O(n \log n)$, ed abbiamo anche dimostrato una delimitazione inferiore che implica che questo tempo di esecuzione è ottimo nel modello dei confronti. Ma cosa possiamo dire al di fuori di questo modello, ad esempio se sono consentite altre operazioni a parte i confronti? I risultati dipenderanno dal tipo di dato specifico che dobbiamo ordinare: ad esempio numeri interi, numeri in virgola mobile, o stringhe di caratteri.

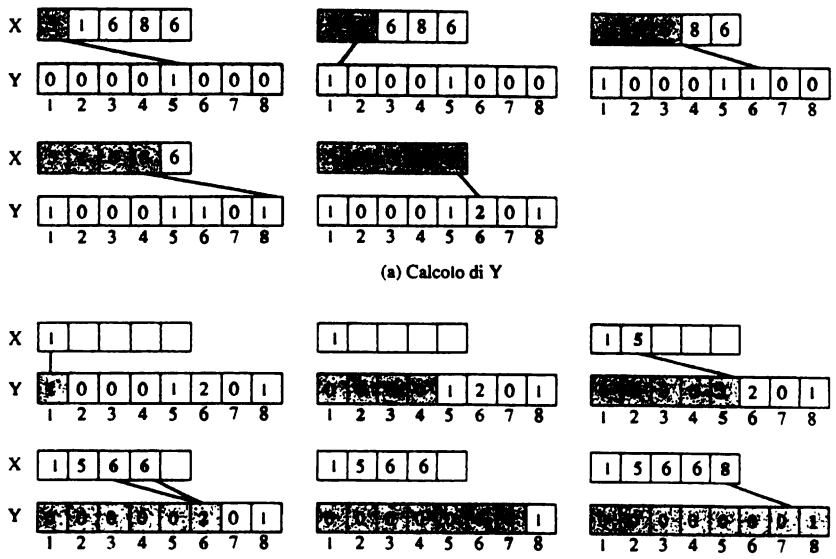

```
    algoritmo integerSort(intero n, array X)
1.    sia Y un array di dimensione n
2.    for i = 1 to n do Y[i] ← 0
3.    for i = 1 to n do incrementa Y[X[i]]
4.    j ← 1
5.    for i = 1 to n do
6.        while ( Y[i] > 0 ) do
7.            X[j] ← i
8.            incrementa j
9.            decrementa Y[i]
```

Figura 4.20 L'algoritmo `integerSort` per ordinare n numeri interi in $[1, n]$ in tempo lineare: dopo l'esecuzione della riga 3, $Y[k]$ è il numero di volte che k compare in X.

Ordinare n interi nell'intervallo $[1, n]$. Consideriamo innanzitutto un esempio molto semplice. Supponiamo di dover ordinare n interi *distinti* i cui valori sono tutti compresi tra 1 e n. Qual è il tempo richiesto per determinare la sequenza ordinata? La risposta è immediata: il tempo è $O(1)$, poiché la sequenza deve essere necessariamente $1, 2, 3, ..., n-1, n$. Consideriamo ora un esempio meno banale. Supponiamo che tutti i numeri siano ancora nell'intervallo $[1, n]$, ma che alcuni possano essere duplicati. L'algoritmo `integerSort` specificato in Figura 4.20 usa un array ausiliario Y per contare quante copie ci sono per ogni numero, come mostrato in Figura 4.21. Osserviamo che `integerSort` non effettua alcun confronto tra elementi, e quindi per analizzarlo dobbiamo tornare a considerare il tempo di esecuzione nell'accezione più generale.

Lemma 4.7 *L'algoritmo* `integerSort` *ordina n numeri interi in $[1, n]$ in tempo $O(n)$.*

Dimostrazione. Dopo l'esecuzione del ciclo `for` alla riga 3, $Y[i]$ contiene il numero di volte che il valore i compare in X, per ogni $i \in [1, n]$. I valori sono poi considerati in ordine crescente, e ciascuno viene riscritto su X tante volte quanto specificato nella corrispondente posizione dell'array Y. Da ciò segue la correttezza. Rispetto al tempo di esecuzione, l'unico punto non ovvio nell'analisi deriva dal fatto che il terzo ciclo `for` nello pseudocodice di Figura 4.20 contiene un ciclo `while` innestato. Normalmente, quando siamo in presenza di cicli innestati, la limitazione sul tempo si ottiene effettuando il prodotto del numero di operazioni di ogni ciclo; dobbiamo invece dimostrare che questo ciclo richiede comunque tempo $O(n)$, e non $O(n^2)$. Ciò segue dalla seguente osservazione. Il ciclo interno può essere eseguito anche n volte, ma non su tutte le n iterazioni del ciclo esterno! Per convincersi di questo, è sufficiente associare al tempo necessario per incrementare gli elementi dell'array Y il tempo necessario per decrementarli: dato che ci sono soltanto n incrementi, ci saranno solo n decrementi. □



(a) Calcolo di Y

(b) Ricostruzione di X

Figura 4.21 Esecuzione dell'algoritmo `integerSort` per ordinare n interi in $[1, k]$: nell'esempio $n = 5$ e $k = 8$.

Ordinare n interi nell'intervallo $[1, k]$. L'algoritmo `integerSort` può essere facilmente adattato per trattare il caso in cui il massimo valore da ordinare, k, sia diverso da n. È infatti sufficiente usare un array ausiliario Y di dimensione k, anziché n. In tal modo potremo indicizzare k valori diversi, ma il tempo di esecuzione diventerà $O(n + k)$: ogni elemento di Y infatti deve essere inizializzato e poi letto almeno una volta durante la ricostruzione di X. Nello pseudocodice di Figura 4.20, basta sostituire k a n nelle linee 1, 2 e 5. La Fignra 4.21 mostra l'algoritmo `integerSort` in azione.

Osserviamo che il tempo $O(n + k)$ è lineare nella dimensione dell'istanza se k è $O(n)$. Ma cosa accade per valori di k più grandi? Esiste un algoritmo per ordinare n numeri interi in tempo $O(n)$ se il massimo valore k è, ad esempio, $O(n^c)$, per una costante $c > 1$? Risponderemo a questa domanda, affermativamente, nel Paragrafo 4.7.

Ordinare n record con chiavi intere nell'intervallo $[1, k]$. Generalizziamo ulteriormente il problema, assumendo che gli elementi da ordinare non siano necessariamente numeri interi. Supponiamo quindi di avere una lista di n record, ognuno con una chiave che è un numero intero in $[1, k]$: ad esempio, un record potrebbe contenere nome, cognome e data di nascita di una persona, e la chiave


```
algoritmo bucketSort(array X di dimensione n, intero k)
1.   sia Y un array di dimensione k
2.   for i = 1 to k do Y[i] ← lista vuota
3.   for i = 1 to n do
4.       if ( chiave(X[i]) ∉ [1, k] ) then errore
5.       else appendi il record X[i] alla lista Y[chiave(X[i])]
6.   for i = 1 to k do
7.       copia ordinatamente in X gli elementi della lista Y[i]
```

Figura 4.22 L'algoritmo `bucketSort` per ordinare n record con chiavi intere in $[1, k]$ in tempo $O(n + k)$.

potrebbe essere il mese di nascita (quindi $k = 12$). Possiamo ancora ordinare questi elementi in tempo $O(n + k)$?

Il problema fondamentale sembra derivare dal fatto che non possiamo più usare contatori come nell'algoritmo `integerSort`, poiché potrebbero esistere elementi diversi ma con la stessa chiave (ad esempio, due o più persone nate nello stesso mese). È comunque semplice ovviare a questo inconveniente: usiamo un array ausiliario Y, di dimensione k, i cui elementi sono liste, anziché contatori. L'algoritmo, durante la lettura della sequenza, copia ogni record nella lista appropriata, in base al valore della corrispondente chiave. Alla fine è sufficiente concatenare tutte le liste ordinatamente, ovvero, per valori di k crescenti. Lo pseudocodice è mostrato in Figura 4.22. Dalla sua analisi, è facile verificare il seguente teorema:

Teorema 4.9 *L'algoritmo* `bucketSort` *ordina n record con chiavi intere in $[1, k]$ in tempo $O(n + k)$.*

Ad esempio, il `bucketSort` è molto efficiente se vogliamo ordinare 10000 persone in base al mese del loro compleanno: le chiavi possono assumere solo valori costanti in $[1, 12]$, e la distribuzione delle persone sulle 12 liste è un'operazione molto rapida. Concludiamo osservando che nessuno degli algoritmi presentati in questo paragrafo è in grado di ordinare *in loco*.

Stabilità. Definiamo un algoritmo di ordinamento *stabile* se esso preserva l'ordine iniziale tra due elementi dello stesso valore. Questa proprietà è importante per estendere il `bucketSort` ad un algoritmo che funziona bene anche quando k è grande, come vedremo nel Paragrafo 4.7. Quale degli algoritmi che abbiamo visto finora è stabile?

- Il `bucketSort` è stabile, poiché aggiungiamo gli elementi alle liste $Y[i]$ in ordine, e preserviamo questo ordine concatenando le liste.
- L'`heapSort` non è stabile, poiché la fase di inizializzazione (la procedura `heapify`) distrugge l'ordine iniziale tra gli elementi.

- La stabilità del `mergeSort` dipende da come implementiamo il passo di `merge`. Ad esempio, dividendo l'array nel suo punto di mezzo e privilegiando, nella fase di `merge`, gli elementi con indice inferiore in caso di valori uguali, allora l'algoritmo è stabile.
- Anche la stabilità del `quickSort` dipende dal modo in cui viene effettuata la partizione: la procedura `partition` presentata nel paragrafo 4.5, ad esempio, non è stabile, a causa dello scambio del pemo con il primo elemento del sottoarray da ordinare (riga 9 in Figura 4.17).

In realtà molti degli algoritmi che ordinano mediante confronti possono essere resi stabili: basta associare ad ogni elemento la sua posizione nella sequenza in ingresso, ed ordinare poi le coppie elemento – posizione. Solo alcuni algoritmi, invece, sono spontaneamente stabili, come abbiamo evidenziato sopra.

4.7 Radixsort

Abbiamo visto nel Paragrafo 4.6 come ordinare n numeri interi in $[1, k]$ in tempo lineare quando $k = O(n)$. Ma cosa fare se k è grande? Consideriamo, ad esempio, la rappresentazione decimale di un numero:

$$x = x_0 + 10x_1 + 100x_2 + 1000x_3 + \ldots$$

con $x_i \in [0, 9]$ per ogni i. Queste cifre decimali sono sufficientemente piccole per applicare ripetutamente il `bucketSort`. In particolare, partiamo dalla cifra meno significativa verso quella più significativa: ovvero eseguiamo prima un `bucketSort` in base a x_0, poi un `bucketSort` in base a x_1, e via dicendo. Una domanda sorge spontanea: perché eseguiamo varie passate di `bucketSort`, e soprattutto, perché ordiniamo per prime le cifre meno significative? Poiché l'ultimo passo di `bucketSort` è quello che ha l'effetto principale nell'ordinamento finale, se cercassimo di ordinare a mano faremmo certamente qualcosa di diverso, usando un approccio *top-down*: ordineremmo prima con un `bucketSort` in base alla cifra più significativa, e poi ordineremmo ricorsivamente gli elementi che hanno la stessa cifra iniziale. Questo procedimento naturalmente è corretto, ma è più difficile da implementare, perché divide il problema iniziale in tanti sottoproblemi di dimensioni anche molto diverse l'uno dall'altro. Al contrario, il `radixSort` segue un approccio *bottom-up* che consente di non dividere mai l'array: il `bucketSort` è applicato in ogni passata all'intero array, usando i `bucketSort` precedenti per separare elementi aventi la stessa i-esima cifra decimale. Possiamo esprimere l'algoritmo `radixSort` in maniera concisa come mostrato in Figura 4.23. Da questa formulazione è facile dimostrare la correttezza dell'algoritmo.

Lemma 4.8 *L'algoritmo* `radixSort` *mostrato in Figura 4.23 è corretto.*

Dimostrazione. Dimostreremo per induzione che, dopo i chiamate alla procedura `bucketSort`, i numeri sono ordinati in base alle prime i cifre meno significative.


```
    procedura bucketSort(array A di n interi, interi b e t)
1.      sia Y un array di dimensione b
2.      for i = 1 to b do Y[i] ← lista vuota
3.      for i = 1 to n do
4.          c ← t-esima cifra di A[i] nella rappresentazione in base b
5.          appendi A[i] alla lista Y[c + 1]
6.      for i = 1 to b do
7.          copia ordinatamente in A gli elementi della lista Y[i]

    algoritmo radixSort(array A di n interi)
8.      t ← 0
9.      while (esiste un numero la cui t-esima cifra è ≠ 0)
10.         bucketSort(A, 10, t)
11.         t ← t + 1
```

Figura 4.23 L'algoritmo `radixSort`, utilizzando come base per il `bucketSort` il valore 10. La t-esima chiamata al `bucketSort` considera come chiave la t-esima cifra meno significativa della rappresentazione decimale dei numeri.

Il passo base (per $i = 0$) è banalmente vero. Consideriamo ora il passo induttivo, e in particolare l'$(i + 1)$-esimo `bucketSort`. Siano x e y due qualunque numeri in ingresso. Considerando le cifre di x e y a partire dalla meno significativa, distinguiamo due casi:

- x e y hanno la stessa $(i+1)$-esima cifra: la proprietà di stabilità del `bucketSort` in tal caso preserva l'ordine precedente di x e y, che per ipotesi induttiva sono già ordinati in base alle prime i cifre.
- x e y hanno una diversa $(i + 1)$-esima cifra: il `bucketSort` al passo $(i + 1)$ li mette nell'ordine giusto rispetto alla $(i + 1)$-esima cifra.

Il passo induttivo è dunque vero in entrambi i casi. Poiché il namero di cifre è finito, per i sufficientemente grande la lista sarà correttamente ordinata. □

Consideriamo ora il tempo di esecuzione. L'algoritmo richiede tempo $O(n)$ per ogni `bucketSort`. Ci sono $\log_{10} k$ passate di `bucketSort`, e quindi il tempo totale è $O(n \log k)$. È questo il miglior algoritmo da usare? La risposta è ovviamente no. Se $k < n$, il `radixSort` richiede tempo $O(n \log k)$ mentre `bucketSort` ne richiede solo $O(n)$. Se $k > n$, $O(n \log k)$ può risultare perfino peggiore del tempo $O(n \log n)$ richiesto dagli algoritmi ottimali che ordinano per confronti. Come possiamo quindi migliorarlo? Innanzitutto, moltiplicazioni e divisioni sono costose, ed è meglio usare una base che è una potenza di 2: questa è un'ottimizzazione facile, ma fa risparmiare solo un fattore costante. Inoltre, usare la notazione decimale non sembra una scelta molto oculata, poiché il valore 10 è troppo piccolo per la base: grazie al Teorema 4.9, possiamo scegliere una base grande come $O(n)$ senza far aumentare significativamente i tempi richiesti dal `bucketSort`. Abbiamo quindi:

Teorema 4.10 *Usando come base per il* `bucketSort` *un valore* $b = \Theta(n)$, *l'algoritmo* `radixSort` *ordina* n *numeri interi in* $[1, k]$ *in tempo*

$$O\left(n\left(1+\frac{\log k}{\log n}\right)\right)$$

Dimostrazione. Ci sono $\log_b k = O(\log_n k)$ passate di `bucketSort`, e ciascuna richiede tempo $O(n)$. Usando le regole per il cambiamento di base dei logaritmi, il tempo totale è quindi $O(n \log_n k) = O(n \log k / \log n)$. Per contemplare nell'analisi anche il caso in cui $k < n$, aggiungiamo $O(n)$ a questa quantità: tempo $O(n)$ è infatti richiesto almeno per la lettura della sequenza. □

Ad esempio, supponiamo di voler ordinare un milione di nameri a 32-bit. Poiché un milione è compreso tra 2^{19} e 2^{20}, usando come base il valore 2^{16}, due passate di `bucketSort` sono sufficienti per ordinare e ciascuna di esse richiede solo tempo lineare.

Vale la pena sottolineare che esistono metodi più complicati per ordinare numeri interi con tempi di esecuzione del tipo $O(n \log \log k)$ o $O(n\sqrt{\log \log n})$ [9, 10]. Questi metodi purtroppo sono interessanti solo dal punto di vista teorico, e sono preferibili solo quando k è enormemente più grande di n.

4.8 Problemi

Problema 4.1 Siano date n monete d'oro tutte dello stesso peso, tranne una che è più leggera delle altre, ed una bilancia con due piatti, su ciascuno dei quali è possibile mettere un numero *qualunque* di monete e sapere se i piatti hanno lo stesso peso, o quale dei due è più leggero. Progettare un algoritmo per trovare la moneta "leggera" che richieda $O(\log n)$ pesate nel caso peggiore. Disegnare poi l'albero di decisione corrispondente all'algoritmo proposto nel caso $n = 8$, assumendo che ogni nodo dell'albero corrisponda alla pesatura di due sottoinsiemi di monete.

Problema 4.2 Implementare gli algoritmi `selectionSort` ed `insertionSort` in modo ricorsivo. Impostare poi una relazione di ricorrenza che descriva il tempo di esecuzione dell'implementazione proposta e risolverla sia per sostituzione che per iterazione.

Problema 4.3 (*) Discutere il numero medio di confronti eseguito dall'algoritmo `insertionSort` assumendo che le permutazioni della sequenza abbiano tutte la stessa probabilità. Qual è la probabilità che, inserendo l'i-esimo elemento, eseguiamo j confronti, per $j \in [1, i-1]$?

Problema 4.4 Considerare la seguente sequenza di numeri:

1 3 20 32 80 15 22 40 60 93 81 42

Assumendo che siano memorizzati in un array secondo la rappresentazione posizionale, discutere, motivando la risposta, se la sequenza rappresenta un heap.

Problema 4.5 Implementare una nuova operazione su heap, `insert`(H, k), che inserisce nell'heap H il nuovo valore k. Quanto costa, nel caso peggiore, costruire un heap inserendo ripetutamente elementi? Usare questa costruzione modifica il tempo di esecuzione dell'algoritmo `heapSort` nel caso peggiore?

Problema 4.6 Implementare le due nuove operazioni su heap descritte di seguito:

- `aumentaChiave`(H, i, k), che imposta $H[i]$ al massimo tra il valore precedente e il nuovo valore k, aggiornando la struttura heap appropriatamente.
- `decrementaChiave`(H, i, k), che imposta $H[i]$ al minimo tra il valore precedente e il nuovo valore k, aggiornando la struttura heap appropriatamente.

Problema 4.7 (**) Negli heap binari con invecchiamento è possibile far "invecchiare" tutti gli elementi dell'heap, decrementando (o incrementando) tutte le chiavi della stessa quantità. Potremmo farlo banalmente scorrendo l'array e modificando ciascuna chiave. Questa implementazione richiederebbe tempo $O(n)$. Dimostrare come effettuare l'invecchiamento in tempo $O(1)$.

Suggerimento: invece di memorizzare in ciascun nodo il valore assoluto della chiave associata al nodo, mantenere valori relativi, in particolare, la differenza tra la chiave del nodo e quella del padre. Solo la radice mantiene un valore assoluto. L'invecchiamento consiste quindi nell'alterare il valore della radice. Modificare lo pseudocodice di tutte le operazioni fondamentali viste per gli heap in modo da poter lavorare su heap con invecchiamento, mostrando che si mantengono sia la correttezza sia il tempo di esecuzione logaritmico.

Problema 4.8 (*) Dare una delimitazione inferiore al numero di confronti richiesti nel caso peggiore per fondere due sequenze ordinate di lunghezza n ciascuna.

Problema 4.9 Il problema della bandiera nazionale italiana è definito nel modo seguente. Sia A un array i cui elementi possono assumere solo uno di tre possibili valori: verde, bianco e rosso. Ordinare l'array in modo che tutti gli elementi verdi siano a sinistra, quelli bianchi al centro e quelli rossi a destra. L'algoritmo deve richiedere tempo lineare nel caso peggiore, può solo scambiare elementi, ed ha solo una variabile aggiuntiva per mantenere uno degli elementi. In particolare, non può usare contatori per mantenere il numero di elementi di un certo colore. Inoltre, una sola passata sull'array è sufficiente per completare l'ordinamento.

Problema 4.10 Descrivere un algoritmo che, dati n interi in $[1, k]$, preprocessa la sequenza in modo da poter poi rispondere a interrogazioni del tipo: "quanti interi cadono nell'intervallo $[a, b]$?", per ogni a e b in tempo $O(1)$. L'algoritmo deve richiedere tempo di preprocessamento $O(n + k)$.

Problema 4.11 Mostrare il comportamento dell'algoritmo `radixSort` per ordinare alfabeticamente la seguente lista di parole:

GRECO BRUNELLO ALBANA ZIBIBBO MERLOT BAROLO FREISA NURAGUS MOSCATO AMARONE TOCAI BARBERA ARNEIS PINOT CHARDONNAY RIESLING MALVASIA CABERNET

Prestare particolare attenzione al fatto che le parole hanno lunghezze diverse, proponendo una soluzione generale per questo problema.

Problema 4.12 (*) Dato un array di n record con chiavi binarie, progettare un algoritmo lineare per ordinare i record *in loco*: a record diversi può corrisponedere la stessa chiave. Assumere ora che le chiavi siano interi in $[1, k]$: progettare un algoritmo per ordinare i record *in loco* in tempo $O(n + k)$, usando solo spazio $O(k)$ in aggiunta all'array che contiene i record.

Problema 4.13 Siano $S_1, S_2, \ldots, S_k$ k insiemi di interi in $[1, n]$ tali che la somma delle loro cardinalità è n. Descrivere un algoritmo che richiede tempo $O(n)$ (non $O(n + k)$) per ordinare tutti gli S_i. Solo gli elementi all'interno dello stesso S_i devono essere ordinati, ma non siamo interessati ad ordinare gli S_i in relazione l'uno all'altro.

Problema 4.14 Il Professor Trombon cita il seguente algoritmo come uno dei suoi risultati di ricerca più entusiasmanti.

```
algoritmo TrombOn(array A, interi l e r)
    if ( l ≥ r ) then return
    if ( A[l] > A[r] ) then scambia A[l] ed A[r]
    TrombOn(A, l + 1, r − 1)
    if ( A[l] > A[l + 1] ) then scambia A[l] ed A[l + 1]
    TrombOn(A, l + 1, r)
```

- Scrivere una relazione di ricorrenza che descriva il numero di volte che due elementi dell'array A vengono confrontati, in funzione della lunghezza dell'array $n = r - l + 1$.
- Trovare il tempo di esecuzione dell'algoritmo `TrombOn` risolvendo la relazione di ricorrenza.
- Qual è il problema risolto dall'algoritmo `TrombOn`?
- Lo stesso problema potrebbe essere risolto più efficientemente?
- Raccomanderesti il Prof. Trombon per il Premio Nobel?

Problema 4.15 L'algoritmo `stupidSort` opera nel modo seguente. Partendo dall'inizio dell'array, esegue una scansione finché non trova due elementi consecutivi nell'ordine sbagliato: in tal caso, scambia gli elementi ed inizia una nuova scansione, sempre partendo dalla prima posizione dell'array.

- Dimostrare il tempo di esecuzione dell'algoritmo `stupidSort` nel caso migliore e nel caso peggiore.
- Mostrare una implementazione iterativa ed una implementazione ricorsiva dell'algoritmo `stupidSort`.
- Discutere l'uso di memoria nelle due implementazioni.

4.9 Sommario

In questo capitolo abbiamo analizzato il problema dell'ordinamento di un insieme di oggetti. Abbiamo dapprima introdotto un modello astratto per questo problema: nel modello basato su confronti non si fanno ipotesi sugli oggetti da ordinare, ma si consente all'algoritmo di operare solo tramite confronti tra elementi. In questo modello abbiamo poi derivato sia una delimitazione inferiore $\Omega(n \log n)$ alla complessità del problema, sia vari algoritmi ottimi nel caso peggiore (come il `mergeSort` e l'`heapSort`) o nel caso atteso (come il `quickSort`). I vari algoritmi di ordinamento che abbiamo descritto ci hanno inoltre permesso di introdurre nuove idee e tecniche di progettazione e di analisi di algoritmi che riassumiamo di seguito.

La tecnica del divide et impera. Il `mergeSort` ed il `quickSort` sono esempi diversi di applicazione della tecnica del *divide et impera*, una tecnica algoritmica molto potente e generale: l'idea consiste nel dividere i dati di ingresso in due o più sottoinsiemi, risolvere ricorsivamente il problema sui sottoinsiemi e poi ricombinare la soluzione dei sottoproblemi per ottenere la soluzione globale. Questi due algoritmi differiscono nel modo in cui partizionann il problema e ricombinano le soluzioni. Il `mergeSort` lavora con una partizione semplice, bilanciata sulla taglia: la soluzione ricorsiva delle due parti produce due sottoinsiemi ordinati che sono "fusi" in un modo non banale nella fase di ricombinazione. Il `quickSort` lavora invece con una partizione più complicata, basata sui valori degli elementi, ma il passo di ricombinazione è banale.

L'utilizzo di strutture dati efficienti. L'`heapSort` esemplifica l'idea che partendo da un algoritmo lento (il `selectionSort`) ed aggiungendo strutture dati opportunamente progettate, se ne possono migliorare sostanzialmente le prestazioni.

La tecnica della randomizzazione. Il `quickSort` ci ha dato anche l'opportunità di esplorare la potenza della randomizzazione per evitare i casi peggiori, rendendo valida l'analisi del caso medio anche quando l'istanza non è affatto casuale o quando non conosciamo la distribuzione delle istanze.

La dipendenza dal modello. Tutti gli algoritmi elencati sopra ordinano mediante confronti tra oggetti, e sono quindi indipendenti dal tipo degli oggetti da ordinare e dalla relazione d'ordine. Il `bucketSort` ed il `radixSort` mostrano invece che è possibile derivare algoritmi particolarmente efficienti per numeri e stringhe di caratteri, sfruttando con attenzione le proprietà degli oggetti da ordinare.

4.10 Note bibliografiche

Il problema dell'ordinamento è di grande importanza in innamerevoli contesti, e non è quindi sorprendente che esso sia stato studiato fin degli albori delle discipline informatiche: uno dei primi articoli scientifici è infatti apparso nel lontano

1956 [8]. Da quel momento, numerosi ricercatori hanno approfondito questo problema. Knuth [12] rappresenta certamente una delle fonti più autorevoli sul problema dell'ordinamento, e descrive in dettaglio non solo gli algoritmi che abbiamo presentato in questo capitolo, ma molti altri interessanti approcci. In particolare, Knuth cita un raccoglitore meccanico risalente al 1938 come primo esempio di ordinamento per fusione: il raccoglitore era infatti in grado di fondere in un unico passo due pacchi di schede perforate già ordinate. Una pionieristica implementazione del `mergeSort` sul calcolatore EDVAC, realizzata nel 1945, è attribuita a John von Neumann. L'algoritmo `heapSort` fu originariamente proposto da Williams [14], mentre Floyd [6] ha il merito di aver mostrato come costruire un heap in tempo lineare. Il `quickSort` fu inventato da Tony Hoare [11], ed esemplifica un approccio più generale che potremmo chiamare "ordinamento per distribuzione". Sedgewick fornisce molti dettagli implementativi su questo algoritmo [13]. Il problema 4.9 fu risolto per la prima volta in [4]. Varie versioni di ordinamenti lineari sono attribuite da Knuth a H. H. Seward, che nel 1954 introdusse l'idea di contare il numero di copie di ogni oggetto per poi posizionarlo opportunamente. Il `radixSort` a partire dalla cifra meno significativa sembra fosse popolare già nel 1929, nelle macchine perforatrici di schede usate da L. J. Comrie. Infine, gli alberi di decisione furono introdotti per la prima volta da Ford e Johnson [7]; generalizzazioni di questo modello per studiare delimitazioni inferiori alla complessità di vari problemi di ordinamento sono discusse in [2].

Nonostante questa estesa letteratura, il problema dell'ordinamento ha continuato a presentare interessanti sfide fino ai nostri giorni. In questo capitolo, ad esempio, abbiamo studiato il tempo di esecuzione degli algoritmi di ordinamento solo in funzione del numero n di elementi da ordinare, e non, ad esempio, in funzione del grado di disordine della sequenza in ingresso: riflettendo un istante, però, potremo realizzare che alcuni algoritmi, come il `bubbleSort`, sono in grado di trarre vantaggio dal fatto che la sequenza in ingresso sia già abbastanza (se non completamente) ordinata, mentre altri, come l'`heapSort`, non sono in grado di sfruttare questa caratteristica. La ricerca sugli algoritmi di ordinamento adattivi studia esattamente questo aspetto, prendendo in considerazione varie misure di disordine: una interessante rassegna di tecniche ed algoritmi adattivi è presentata in [5]. Molte applicazioni reali, inoltre, devono affrontare il problema di ordinare enormi quantità di dati, dell'ordine delle migliaia di TeraByte. In questo scenario, è necessario tener conto del fatto che i dati potrebbero non entrare interamente nella memoria centrale, e quindi progettare algoritmi di ordinamento in grado di accedere alla memoria esterna minimizzando il numero di operazioni di input/output. Ad esempio, è stato mostrato in [1] come adattare l'algoritmo `mergeSort` al modello di memoria esterna che abbiamo descritto nel Paragrafo 2.8 del Capitolo 2, ma molto rimane ancora da dire in questo ambito, soprattutto se si considera che i moderni calcolatori hanno una gerarchia di memoria sempre più complessa e con un numero di livelli sempre maggiore.

Riferimenti bibliografici

[1] A. Aggarwal e J. S. Vitter "The input/output complexity of sorting and related problems", *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] M. Ben-Or, "Lower bounds for algebraic computation trees", *Proceedings of the fifteenth Annual ACM Symposium on Theory of Computing*, 80–86, 1983.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Introduction to algorithms*, McGraw-Hill, 2001.

[4] E. W. Dijkstra, *A discipline of programming*, Prentice-Hall, Englewood-Cliffs, NJ, 1976.

[5] V. Estivill-Castro e D. Wood, "A survey of adaptive sorting algorithms", *ACM Computing Surveys*, 24(4):441–476, 1992.

[6] R. W. Floyd, "Algorithm 245 (treesort)", *Communications of the ACM*, 7:701, 1964.

[7] L. R. Ford, Jr., e S. M. Johnson, "A tournament problem", *The American Mathematical Monthly*, 66:387–389, 1959.

[8] E. M. Friend, "Sorting on electronic computer systems", *Journal of the Association for Computing Machinery*, 3:134–168, 1956.

[9] Y. Han, "Deterministic sorting in $O(n \log\log n)$ time and linear space", *Journal of Algorithms*, 50(1):96–105, 2004.

[10] Y. Han e M. Thorup, "Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space ", *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science* (FOCS'02), 135–144, 2002.

[11] C. A. R. Hoare, "Quicksort", *The Computer Journal*, 5(1):10–15, 1962.

[12] D. E. Knuth, *Sorting and searching*, volume 3 di *The Art of Computer Programming*, Addison-Wesley, 1973.

[13] R. Sedgewick, "Implementing quicksort programs", *Communications of the ACM*, 21(10):847–857, 1978.

[14] J. W. J. Williams, "Algorithm 232 (heapsort)", *Communications of the ACM*, 7:347–348, 1964.

5

Selezione e statistiche di ordine

In medio stat virtus.

(Proverbio latino)

Gli algoritmi per risolvere problemi di selezione e di statistiche d'ordine permettono di estrarre da grandi quantità di dati un piccolo insieme di numeri che ne rappresentino alcune caratteristiche statisticamente salienti. Esempi tipici includono il calcolo della media, della moda, o del mediano di n numeri. La *media* di n numeri è definita come la somma dei numeri divisa per n e può essere facilmente calcolata in tempo $O(n)$. La *moda* è invece il valore più frequente. Un approccio per calcolare la moda consiste nell'ordinare i valori e poi sfruttare l'ordinamento per contare quante copie ci sono di ogni valore: il tutto richiede tempo $O(n \log n)$. Esiste una delimitazione inferiore di $\Omega(n \log n)$ per il calcolo della moda (più difficile da dimostrare rispetto al lower bound per l'ordinamento): quindi, nel modello dei confronti, non possiamo sperare di fare di meglio [13]. Osserviamo che la moda potrebbe anche essere definita indipendentemente dall'esistenza di una relazione d'ordine tra i valori; ad esempio, in una collezione di oggetti rossi, blu e verdi, potremmo chiedere qual è il colore più ricorrente: possiamo comunque applicare lo stesso metodo definendo una relazione di ordine arbitraria tra i valori.

Il *mediano*, infine, è il valore che occuperebbe la posizione $\lceil n/2 \rceil$ se l'insieme dei valori fosse ordinato: al contrario della moda, per definire il mediano si richiede di avere ana relazione di ordine. Anche il mediano può essere calcolato facilmente in tempo $O(n \log n)$ ordinando la sequenza e restituendo l'elemento di posizione $\lceil n/2 \rceil$ nella sequenza ordinata. Vedremo però in questo capitolo che esistono algoritmi più efficienti. Osserviamo, innanzitutto, che una strategia a volte utile per risolvere un problema è cercare di risolverne uno più generale: talvolta risolvere il problema più generale può infatti risultare più semplice che non risolvere quello originario, perché problemi più generali danno più facilmente luogo a sottoproblemi ricorsivi. Il problema più generale che risolveremo è quello della *selezione*:

dati un insieme di n elementi ed un intero $k \in [1, n]$, trovare l'elemento che occuperebbe la k-esima posizione se l'insieme fosse ordinato.

```
algoritmo minimo(array A) → elem
1.   min ← A[1]
2.   for i = 2 to n do
3.      if ( A[i] < min ) then min ← A[i]
4.   return min
```

Figura 5.1 Ricerca del minimo.

Il mediano è pertanto un caso speciale di selezione per $k = \lceil n/2 \rceil$. Come nel caso dell'ordinamento, assumeremo che i dati in ingresso siano contenuti in un array disordinato A indicizzato con numeri da 1 a n. Vedremo due algoritmi di selezione: uno randomizzato basato sul `quickSort` (Paragrafo 5.2) ed uno deterministico (Paragrafo 5.3): l'algoritmo randomizzato è più veloce in pratica, mentre quello deterministico è estremamente elegante ed interessante dal punto di vista teorico. Prima di considerare questi algoritmi, studieremo nel Paragrafo 5.1 il problema della selezione quando k è un valore molto piccolo o molto grande rispetto al numero n di elementi, ma comunque lontano da $\lceil n/2 \rceil$: sebbene ciò non sia utile ai fini del calcolo del mediano, ci permetterà comunque di prendere familiarità con il problema e di introdurre alcune tecniche interessanti.

5.1 Selezione per piccoli valori di k

In questo paragrafo affronteremo il problema della selezione per piccoli valori di k: le stesse tecniche possono essere facilmente adattate per risolvere problemi di selezione quando k è molto grande, ovvero molto vicino al numero n di elementi. Consideriamo innanzitutto il problema della ricerca del minimo, ovvero il caso $k = 1$. Abbiamo già incontrato questo problema nel Paragrafo 4.2 del Capitolo 4: l'algoritmo `selectionSort`, infatti, estrae ripetutamente il minimo tra gli elementi non ancora ordinati e lo mette in una posizione opportuna. Il problema di estrazione del minimo è molto semplice, e può essere risolto con $(n-1)$ confronti come mostrato in Figura 5.1: basta esaminare l'array, mantenendo il valore minimo incontrato fino a quel momento, ed aggiornandolo opportunamente nel caso in cui si incontri un elemento con valore inferiore. Vedremo ora come generalizzare questa semplice idea per cercare, in modo non banale, il secondo minimo e per risolvere in tempo lineare il problema della selezione quando $k = O(n/\log n)$.

5.1.1 Ricerca del secondo minimo

Seguendo la stessa strategia dell'algoritmo `minimo` in Figara 5.1, potremmo mantenere traccia di due valori: il primo ed il secondo elemento più piccolo incontrato fino ad un certo punto. Mentre stiamo esaminando i dati in ingresso, basterà confrontare ogni elemento col valore attuale del secondo minimo per decidere se


```
algoritmo secondoMinimo(array A) → elem
1.   primoMin ← A[1]
2.   secondoMin ← A[2]
3.   if ( secondoMin < primoMin ) then scambia primoMin e secondoMin
4.   for i = 3 to n do
5.      if ( A[i] < secondoMin ) then
6.         secondoMin ← A[i]
7.         if ( secondoMin < primoMin ) then
8.            scambia primoMin e secondoMin
9.   return secondoMin
```

Figura 5.2 Ricerca del secondo minimo.

l'elemento incontrato rientra nei primi due. Se un valore è tra i primi due più piccoli, si esegue un ulteriore confronto per decidere se è il minimo o meno. Lo pseudocodice è mostrato in Figura 5.2. L'algoritmo `secondoMinimo`, anche se molto semplice, presenta delle interessanti proprietà in sede di analisi.

Lemma 5.1 *L'algoritmo* `secondoMinimo` *esegue* $(2n-3)$ *confronti nel caso peggiore ed* $n + O(\log n)$ *confronti nel caso medio.*

Dimostrazione. Il caso peggiore si presenta quando i valori sono in ordine decrescente e, di conseguenza, ognuna delle $(n-2)$ iterazioni effettua due confronti: uno a riga 5 e l'altro a riga 8. Il totale è quindi di $2(n-2)+1 = 2n-3$ confronti. Per condurre un'analisi nel caso medio, assumiamo che ogni permutazione di A sia equiprobabile. Sotto questa ipotesi, ogni iterazione esegue ancora il primo confronto (riga 5), ma esegue il secondo confronto (riga 8) solo se $A[i]$ è il minimo oppure il secondo minimo dei primi i valori di A. Ognuno dei primi i valori di A può essere uno dei due più piccoli con uguale probabilità, pari a $2/i$. In media, eseguiamo quindi il secondo confronto un numero di volte pari a

$$\sum_{i=3}^{n} \frac{2}{i} = 2\ln n + O(1)$$

dove il simbolo ln indica il logaritmo naturale. Osserviamo che per stabilire la precedente uguaglianza abbiamo usato la nota limitazione sulla serie armonica

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

Ulteriori dettagli sulla serie armonica sono riportati in Appendice. Il numero medio di confronti che si effettuano in totale è pertanto $n + O(\log n)$. □

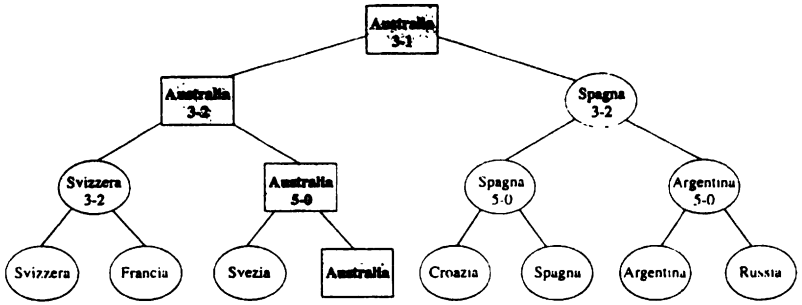


Figura 5.3 L'albero del torneo relativo alla Coppa Davis del 2003, a partire dai quarti di finale: i nodi rettangolari individuano gli incontri vinti dalla squadra migliore, i nodi grigi ovali sono i candidati ad essere la seconda squadra più forte.

Possiamo sperare di ottenere lo stesso tempo $n + O(\log n)$ anche nel caso peggiore? Pensiamo ad un torneo di tennis con n giocatori, assumendo per semplicità che n sia una potenza di 2: costruiamo un albero binario completo con n foglie, detto *albero del torneo*, i cui nodi interni rappresentano incontri ad eliminazione diretta. Un esempio è illustrato in Figura 5.3. Al primo livello ci sono $n/2$ incontri: gli $n/2$ vincitori passano al turno successivo, ovvero al livello successivo dell'albero. Assumendo che in ogni partita il migliore riesca sempre a vincere, il giocatore più forte vincerà tutti gli incontri: potrà quindi essere trovato come il vincitore dell'ultimo incontro (la finale) nella radice dell'albero. Dove e quanto velocemente possiamo trovare il secondo giocatore più forte nell'albero del torneo? La risposta è data dal seguente teorema:

Teorema 5.1 *L'albero del torneo permette di trovare il secondo minimo con $n + O(\log n)$ confronti nel caso peggiore.*

Dimostrazione. Chiamiamo M ed S il giocatore migliore ed il secondo giocatore migliore, rispettivamente. Sicuramente c'è stato un incontro giocato da M ed S, in cui M ha vinto. Se così non fosse (ovvero, se S non si fosse mai confrontato con M), dovrebbe esistere un incontro tra S ed un giocatore $X \neq M$ che S non è riuscito a superare: ma ciò contraddirrebbe l'ipotesi che S sia il secondo giocatore più forte. Poichè S ed M si sono incontrati, per trovare S basta cercarlo tra gli sconfitti dal giocatore più forte, che sono $\lceil \log n \rceil$: in particolare, sarà sufficiente organizzare ricorsivamente un torneo tra di loro per trovare il secondo giocatore più forte. Questo produce $n + \lceil \log n \rceil - 2$ confronti per trovare il minimo ed il secondo minimo anche nel caso peggiore. □

Il piccolo fattore logaritmico additivo nel Teorema 5.1 ci lascia sperare che, in generale, possiamo effettuare la selezione più efficientemente dell'ordinamento. Nel prossimo paragrafo mostreremo infatti, tramite una generalizzazione dell'idea dell'albero del torneo, come selezionare il k-esimo elemento in tempo lineare quando $k = O(n/\log n)$.


```
algoritmo heapSelect(array A, intero k) → elem
    heapify(A)
    for i = 1 to k − 1 do deleteMax(A)
    return max(A)
```

Figura 5.4 L'algoritmo `heapSelect`.

5.1.2 L'algoritmo Heapselect

L'albero del torneo che abbiamo usato nel precedente paragrafo è abbastanza simile all'heap introdotto nel Paragrafo 4.3.1 del Capitolo 4, e il procedimento per trovare il secondo giocatore più forte ricorda molto quello di cancellazione del massimo in un heap in cui ogni nodo abbia valore maggiore o uguale al valore del padre. Questa analogia suggerisce di usare la struttura dati heap per risolvere il problema della selezione, come mostrato in Figura 5.4. L'analisi dell'algoritmo `heapSelect` è immediata, tenendo conto dell'analisi degli heap condotta nel Teorema 4.5 del Capitolo 4.

Teorema 5.2 *L'algoritmo* `heapSelect` *risolve il problema della selezione del k-esimo elemento in tempo $O(n + k \log n)$.*

Dimostrazione. Basta osservare che le procedure `heapify`, `deleteMax` e `max` richiedono rispettivamente tempo $O(n)$, $O(\log n)$ ed $O(1)$, come dimostrato nel Paragrafo 4.3.1 del Capitolo 4. □

Per $k = O(n/\log n)$, il tempo di esecuzione dell'algoritmo `heapSelect` è pertanto $O(n)$. Questo algoritmo è dunque interessante per piccoli valori di k. Purtroppo, nel caso $k = \lceil n/2 \rceil$, questo approccio ha lo stesso costo di un ordinamento, e quindi non risolve ancora il problema del calcolo del mediano.

5.2 Calcolo randomizzato del mediano

Per risolvere il problema della selezione per qualunque valore di k, ritorniamo all'idea originale: ricorriamo ad un algoritmo di ordinamento e poi restituiamo il k-esimo elemento dell'array ordinato. In particolare, consideriamo l'algoritmo `quickSort` visto nel Paragrafo 4.5 del Capitolo 4, che possiamo facilmente adattare al problema della selezione come mostrato in Figura 5.5. Si noti che, per efficienza e per semplicità di analisi, l'algoritmo `selectl` partiziona l'array A in tre sottoinsiemi, anziché due come nel caso del `quickSort`. È comunque facile modificare la procedura `partition` vista nel Paragrafo 4.5 in modo da produrre i tre sottoinsiemi, lavorando comunque *in loco* ed in tempo $O(n)$.

Il tempo di esecuzione atteso dell'algoritmo `selectl` è $O(n \log n)$, esattamente come per il `quickSort`. Osserviamo però che, se k è minore della lunghezza di A_1, `selectl`(A, k) restituirà sempre un oggetto di A_1: in tal caso non

```
algoritmo select1(array A, intero k) → elem
    scegli un elemento x in A
    partiziona A rispetto ad x calcolando:
        A1 = { y ∈ A : y < x }
        A2 = { y ∈ A : y = x }
        A3 = { y ∈ A : y > x }
    quickSort(A1)
    quickSort(A3)
    return k-esimo elemento della concatenazione di A1, A2 ed A3
```

Figura 5.5 Adattamento dell'algoritmo `quickSort` al problema della selezione.

importa se A_3 venga ordinato o meno, e la chiamata `quickSort`(A_3) risulta del tutto superflua. Allo stesso modo, se k è maggiore della somma delle lunghezze di A_1 ed A_2, `select`(A, k) restituirà sempre un oggetto di A_3, ed in questo caso è la chiamata `quickSort`(A_1) a risultare superflua. In entrambi i casi, possiamo risparmiare tempo facendo solo una delle due chiamate ricorsive. Otteniamo quindi `select2`, una variante dell'algoritmo `select1` mostrata in Figura 5.6.

```
algoritmo select2(array A, intero k) → elem
    scegli un elemento x in A
    partiziona A rispetto ad x calcolando:
        A1 = { y ∈ A : y < x }
        A2 = { y ∈ A : y = x }
        A3 = { y ∈ A : y > x }
    if ( k ≤ |A1| ) then
        quickSort(A1)
        return k-esimo elemento di A1
    else if ( k > |A1| + |A2| ) then
            quickSort(A3)
            return (k − |A1| − |A2|)-esimo elemento di A3
    else return x
```

Figura 5.6 Una variante dell'algoritmo di selezione mostrato in Figura 5.5.

Pur facendo meno chiamate al `quickSort`, il tempo di esecuzione atteso dell'algoritmo `select2` rimane comunque $O(n \log n)$: la chiamata al `quickSort` (su A_1 o su A_3) può infatti richiedere comunque tempo $O(n \log n)$. Per migliorare ulteriormente l'algoritmo, possiamo osservare che le chiamate al `quickSort` ci servono solo per ordinare un sottoarray e poi restituire un elemento in posizione specificata: ma questo è esattamente il problema da cui siamo partiti! Applichiamo quindi lo stesso ragionamento (ovvero, fare una sola chiamata ricorsiva) anche alle chiamate `quickSort`(A_1) e `quickSort`(A_3): possiamo ottenere questo


```
algoritmo quickSelect(array A, intero k) → elem
    scegli un elemento x in A
    partiziona A rispetto ad x calcolando:
        A1 = { y ∈ A : y < x }
        A2 = { y ∈ A : y = x }
        A3 = { y ∈ A : y > x }
    if ( k ≤ |A1| )
    then return quickSelect(A1, k)
    else if ( k > |A1| + |A2| )
        then return quickSelect(A3, k − |A1| − |A2|)
        else return x
```

Figura 5.7 L'algoritmo randomizzato `quickSelect`.

effetto sostituendo le chiamate al `quickSort` con opportune chiamate ricorsive all'algoritmo di selezione stesso. L'algoritmo così ottenuto, che chiamiamo `quickSelect`, è mostrato in Figura 5.7.

Per calcolare il tempo di esecuzione dell'algoritmo `quickSelect`, osserviamo innazitutto che la chiamata ricorsiva è sempre fatta su un problema più piccolo. Se ogni chiamata ricorsiva dimezzasse la dimensione del problema otterremmo la relazione di ricorrenza

$$T(n) = O(n) + T(n/2)$$

che ha soluzione $T(n) = O(n)$, come si può facilmente vedere applicando il teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2). Ovviamente, non sempre siamo così fortunati. Nel caso peggiore potremmo fare una chiamata ricorsiva ad un sottoproblema di $(n-1)$ elementi: infatti A_3 potrebbe essere vuoto ed A_2 potrebbe contenere un solo elemento, il perno x. In questo caso l'equazione di ricorrenza diventerebbe

$$T(n) = O(n) + T(n-1)$$

e, come sappiamo già dall'analisi del `quickSort`, avrebbe soluzione $T(n) = O(n^2)$. Quindi, nel caso peggiore, l'algoritmo `quickSelect` potrebbe essere molto inefficiente. Fortunatamente, il numero atteso di confronti è di molto inferiore.

Analisi probabilistica. Assumiamo ora che il perno x sia scelto in maniera casuale e calcoliamo il numero atteso di confronti. Per iniziare, poniamoci nel caso $k = \lceil n/2 \rceil$ e consideriamo l'insieme S_1 di elementi minori o uguali a x, e l'insieme S_2 di elementi maggiori o uguali a x: il mediano si trova sicuramente nel più grande di questi insiemi, che avrà dimensione $\geq n/2$. Quindi, se non restituiamo x come mediano (nel qual caso l'algoritmo termina), sicuramente eliminiamo $|A2| + \min\{|A1|, |A3|\}$ elementi dalla chiamata ricorsiva. Vediamo

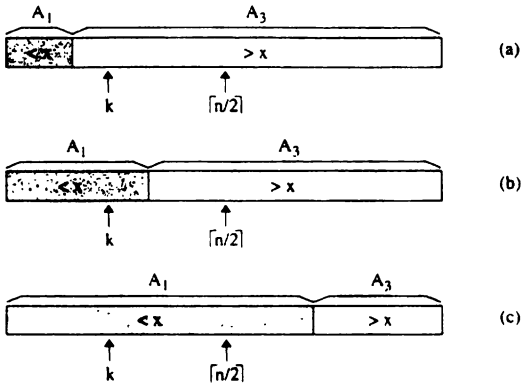


Figura 5.8 Il calcolo del mediano rappresenta il caso peggiore per il problema di selezione, poiché si ricorre sempre sull'array di dimensione maggiore.

ora una delimitazione inferiore al numero di elementi eliminati nel caso in cui $k \neq \lceil n/2 \rceil$.

Lemma 5.2 *Il caso peggiore nell'esecuzione dell'algoritmo* `quickSelect` *si ha per* $k = \lceil n/2 \rceil$.

Dimostrazione. Se $k \neq \lceil n/2 \rceil$, le possibilità di ricorrere sul più piccolo dei due insiemi S_1 ed S_2 aumentano, come mostrato per $k < \lceil n/2 \rceil$ in Figura 5.8: per calcolare il mediano, la ricorsione ha luogo sempre sull'array di dimensione maggiore, ma per calcolare il k-esimo elemento, nel caso (b) si ricorre sull'array di dimensione minore. Il caso $k > \lceil n/2 \rceil$ è simmetrico. □

Grazie al Lemma 5.2, per analizzare `quickSelect` possiamo evitare di scrivere una relazione di ricorrenza randomizzata con due parametri, k ed n, scrivendo invece una relazione di ricorrenza con la sola variabile n. Come ci aspettiamo da un'analisi di valore atteso, otterremo una ricorrenza con una somma, su tutte le possibili scelte casuali, della probabilità di fare quella scelta moltiplicata per il tempo di esecuzione che deriva dalla scelta stessa (vedi Paragrafo 2.6).

Abbiamo detto che ad ogni passo eliminiamo sempre $|A2| + \min\{|A1|, |A3|\}$ elementi dalla chiamata ricorsiva. Questo valore è un qualsiasi numero tra 1 ed $n/2$, con la stessa probabilità: ciò deriva dal fatto che ogni elemento ha la stessa probabilità di essere scelto come perno, e quindi ogni partizione dell'array A è equiprobabile. Di conseguenza, la chiamata ricorsiva a `quickSelect` avviene in maniera equiprobabile su array che hanno una qualsiasi dimensione tra $n/2$ e $(n-1)$: più precisamente, la probabilità di ricorrere su un array di dimensione i è $1/(n/2) = 2/n$, per ogni $i \in [n/2, n-1]$. Ricordando dal Paragrafo 4.5 del Capitolo 4 che la partizione intorno al perno può essere effettuata con $(n-1)$ confronti ed usando la formula per l'analisi del tempo di esecuzione atteso data

nel Paragrafo 2.6 del Capitolo 2, otteniamo la seguente relazione di ricorrenza per il numero atteso di confronti:

$$C(n) = n - 1 + \sum_{i=n/2}^{n-1} \frac{2}{n} C(i) \tag{5.1}$$

Lemma 5.3 *La relazione di ricorrenza (5.1) ha soluzione* $C(n) \leq 4n$.

Dimostrazione. Dimostreremo che $C(n) \leq 4 \cdot n$ usando il metodo di sostituzione. Supponiamo, per ipotesi induttiva, che $C(i) \leq c \cdot n$, per ogni $i < n$ e per una opportuna costante c. Risulta quindi:

$$C(n) = n - 1 + \sum_{i=n/2}^{n-1} \frac{2}{n} C(i) \leq n - 1 + \sum_{i=n/2}^{n-1} \frac{2}{n} c \cdot i = n - 1 + \frac{2c}{n} \sum_{i=n/2}^{n-1} i =$$

$$= n - 1 + \frac{2c}{n} \left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{n/2-1} i \right)$$

Usando la serie aritmetica $\sum_{k=1}^{n} k = n(n+1)/2$ otteniamo:

$$C(n) = n - 1 + \frac{2c}{n} \left(\frac{n^2}{2} - \frac{n^2}{8} - \frac{n}{4} \right) = \left(1 + \frac{3c}{4}\right) n - 1 - \frac{c}{2} \leq \left(1 + \frac{3c}{4}\right) n$$

L'induzione funziona quindi quando $(1 + 3c/4) \leq c$, ossia per $c \geq 4$. □

Riassumiamo quindi le prestazioni dell'algoritmo `quickSelect`:

Teorema 5.3 *Per ogni* k, $1 \leq k \leq n$, *l'algoritmo* `quickSelect` *randomizzato trova il* k*-esimo elemento di un array disordinato di lunghezza* n *eseguendo* $O(n^2)$ *confronti nel caso peggiore. Il numero atteso di confronti è invece* $O(n)$.

Selezione tramite campionamento. Floyd e Rivest [9] hanno osservato che una scelta più accorta del pemo x rende l'algoritmo più efficiente, proponendo una tecnica simile a quella usata degli istituti demoscopici per effettuare indagini statistiche. Questa tecnica, analizzata in [4], si basa sull'idea che l'algoritmo è tanto più efficiente, quanto più il pemo x è vicino alla k-esima posizione. Scegliere il perno come k-esimo elemento di un opportuno campione (invece di sceglierlo casualmente) può fornirci qualcosa di più vicino alla k-esima posizione. Lo pseudocodice dell'algoritmo di selezione basato su campionamento è mostrato in Figura 5.9. Le scelte di m e j influiscono naturalmente sul tempo di esecuzione, ma un'analisi dettagliata è al di là dello scopo di questo libro e rimandiamo il lettore interessato al già citato [4].

```
algoritmo sampleSelect(array A, interi n e k) → elem
    if ( |A| ≤ 10 ) then
        ordina A
        return k-esimo elemento di A
    scegli due parametri m e j "opportunamente"
    S ← sottoinsieme random di A contenente m elementi
    x ← sampleSelect(S, m, j)
    partiziona A rispetto ad x calcolando:
        A1 = { y ∈ A : y < x }
        A2 = { y ∈ A : y = x }
        A3 = { y ∈ A : y > x }
    if ( k ≤ |A1| ) then return sampleSelect(A1, |A1|, k)
    else if ( k > |A1| + |A2| ) then
            return sampleSelect(A3, |A3|, k − |A1| − |A2|)
    else return x
```

Figura 5.9 L'algoritmo `sampleSelect`.

5.3 Calcolo deterministico del mediano

Come abbiamo visto nel Paragrafo 5.2, l'algoritmo `quickSelect` è un algoritmo randomizzato per il calcolo del mediano e, in generale, per la selezione del k-esimo elemento: il valore atteso dei numeri di confronti è $O(n)$, ed anche in pratica `quickSelect` risulta molto veloce. Dal punto di vista teorico, però, sarebbe più soddisfacente avere anche un algoritmo lineare deterministico: Blum, Floyd, Pratt, Rivest e Tarjan [2] hanno risolto per primi questo problema in un modo molto elegante. Ricordiamo che l'algoritmo randomizzato `quickSelect` sceglie a caso un perno x, partiziona l'array in elementi più grandi e più piccoli di x, e richiama se stesso ricorsivamente su uno dei due sottoarray. L'algoritmo più veloce `sampleSelect` opera in modo simile, ma sceglie il perno in un modo leggermente più complicato, richiamando se stesso ricorsivamente su un "campione" statistico dell'input scelto casualmente. L'algoritmo deterministico che presenteremo in questo paragrafo segue la stessa idea, ovvero sceglie x tramite una chiamata ricorsiva, ma riesce a selezionare opportunamente il campione in modo deterministico.

Se potessimo scegliere il perno x come mediano di tutti i valori, ogni chiamata ricorsiva sarebbe su al più la metà dei valori: ma ovviamente, se sapessimo come scegliere il mediano, avremmo già risolto il nostro problema, senza necessità di ulteriori chiamate ricorsive. Il progetto sembra quindi troppo ambizioso. Rilassiamo le nostre pretese, e cerchiamo di selezionare come perno qualcosa di vicino al mediano, ad esempio, un valore che dista al più $n/4$ da esso. In tal caso, ogni chiamata ricorsiva sarebbe su una frazione dell'input al più grande $3n/4$, e questo sarebbe comunque sufficiente per ottenere un algoritmo lineare.


```
1.  algoritmo select(array A, intero k) → elem
2.      if ( |A| ≤ 10 ) then
3.          ordina A
4.          return k-esimo elemento di A
5.      partiziona A in ⌈n/5⌉ sottoinsiemi S_i di (al più) 5 elementi ciascuno
6.      for i = 1 to ⌈n/5⌉ do m_i ← mediano di S_i
7.      M ← select({m_i : i ∈ [1, ⌈n/5⌉]}, ⌈n/10⌉)
8.      partiziona A rispetto ad M calcolando:
9.          A1 = { y ∈ A : y < M }
10.         A2 = { y ∈ A : y = M }
11.         A3 = { y ∈ A : y > M }
12.     if ( k ≤ |A1| ) then return select(A1, k)
13.     else if ( k > |A1| + |A2| ) then return select(A3, k − |A1| − |A2|)
14.     else return M
```

Figura 5.10 L'algoritmo deterministico `select`.

5.3.1 Mediano dei mediani

Come ottenere efficientemente un valore vicino al mediano? Come nel caso dell'algoritmo `sampleSelect`, invece di trovare il mediano dell'intero insieme, cerchiamo il mediano di un opportuno campione. E come scegliere il campione? Ancora una volta usando i mediani! L'algoritmo, il cui pseudocodice è mostrato in Figura 5.10, può essere descritto informalmente come segue:

Passo 1. Raccogli gli elementi in gruppi di 5. Per semplicità, chiameremo l'i-esimo gruppo S_i, con $i \in [1, \lfloor n/5 \rfloor]$. Se n non è divisibile per 5, crea un ulteriore gruppo $S_{\lceil n/5 \rceil}$ contenente gli elementi rimanenti (al più quattro).

Passo 2. Trova il mediano m_i di ogni gruppo S_i.

Passo 3. Tramite una chiamata ricorsiva, trova il mediano M dei mediani m_i individuati nel Passo 2.

Passo 4. Usa M come perno e richiama l'algoritmo ricorsivamente sull'array opportuno, esattamente come in `quickSelect`.

Nello pseudocodice, per motivi di efficienza abbiamo fermato la ricorsione quando A è sufficientemente piccolo (al più 10 elementi). Osserviamo inoltre che la costante 5 nel Passo 1 dell'algoritmo (riga 4 in Figura 5.10) non è cruciale: come si evincerà dall'analisi, potremmo infatti usare senza problemi anche altre costanti ≥ 5, come ad esempio 7 o 9: rimandiamo al Problema 5.1 per ulteriori considerazioni. La seguente proprietà dell'algoritmo `select` sarà poi utile in sede di analisi.

Proprietà 5.1 *Il calcolo dei mediani m_i delle quintuple S_i (riga 5 in Figura 5.10) può essere implementato eseguendo al più $6\lceil n/5 \rceil$ confronti.*

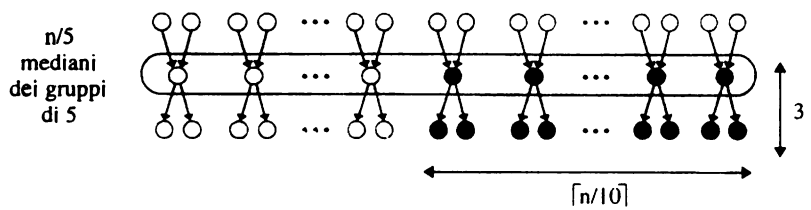


Figura 5.11 Dimostrazione visuale del Lemma 5.4.

Il mediano di ogni gruppo di 5 elementi può essere infatti calcolato eseguendo al più 6 confronti, come il Problema 5.2 richiede di dimostrare. Il calcolo del mediano dei mediani M (riga 6 in Figura 5.10) è implementato tramite una chiamata ricorsiva su un insieme di $\lceil n/5 \rceil$ elementi, e richiede pertanto tempo $T(\lceil n/5 \rceil)$. La maggiore difficoltà nell'analisi sembra nel Passo 4: non sappiamo, infatti, quanto è grande l'insieme su cui si procede ricorsivamente, data la scelta del pemo M effettuata dall'algoritmo. Per dire qualcosa sulla dimensione di questo insieme, ragioniamo in termini degli elementi scartati, cioè non inclusi nella chiamata ricorsiva.

Lemma 5.4 *La chiamata ricorsiva nel Passo 4 dell'algoritmo* `select` *viene effettuata su un insieme contenente al più* $(7n/10 + 3)$ *elementi.*

Dimostrazione. Ad ogni passo, dopo aver partizionato intorno al mediano dei mediani M, scartiamo sempre o tutti i valori più grandi o tutti i valori più piccoli di M (oltre a quelli uguali). Nel primo caso la chiamata ricorsiva del Passo 4 viene effettuata su A_1, nel secondo caso su A_3: supponiamo senza perdere di generalità di ricorrere su A_1, e quindi di scartare i valori di $A_3 \cup A_2$.

Tra gli $\lceil n/5 \rceil$ valori m_i, la metà sono maggiori o uguali ad M, essendo M per definizione il mediano degli m_i. Per le proprietà della parte intera superiore

$$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil = \left\lceil \frac{n}{10} \right\rceil$$

elementi sono quindi maggiori o uguali ad M. Ricordiamo inoltre che m_i è il mediano dei 5 valori nel gruppo S_i, per ogni $i \in [1, \lfloor n/5 \rfloor]$: quindi, per ogni $m_i \geq M$ il cui gruppo contiene 5 elementi, altri due valori di S_i sono $\geq m_i \geq M$. Da quanto detto, almeno $\lceil n/10 \rceil - 1$ gruppi contengono almeno tre elementi maggiori o uguali ad M. Quindi:

$$|A_2 \cup A_3| \geq 3\left(\left\lceil \frac{n}{10} \right\rceil - 1\right) \geq \frac{3n}{10} - 3$$

elementi. Con un ragionamento analogo, possiamo dimostrare che

$$|A_1 \cup A_2| \geq \frac{3n}{10} - 3$$

La chiamata ricorsiva finale sarà pertanto su un array contenente al più $7n/10 + 3$ elementi. □

L'algoritmo `select`, per quanto dimostrato nel Lemma 5.4, ha la proprietà che desideravamo: ogni chiamata ricorsiva viene fatta su una frazione costante dei dati in ingresso, scelta in modo deterministico. Calcoliamo ora il namero di confronti effettuati.

Teorema 5.4 *L'algoritmo* `select` *esegue nel caso peggiore* $O(n)$ *confronti.*

Dimostrazione. Il numero $T(n)$ di confronti effettuati può essere trovato risolvendo la seguente relazione di ricorrenza

$$T(n) \leq \frac{11}{5}n + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 3\right) \tag{5.2}$$

che deriva dalle seguenti considerazioni.

- Per la Proprietà 5.1, il calcolo dei mediani m_i richiede al più $6\lceil n/5 \rceil \leq 6(n/5) + 1$ confronti.
- Il calcolo del mediano dei mediani è effettuato eseguendo una chiamata ricorsiva su un insieme contenente $\lceil n/5 \rceil$ elementi. Ciò richiede tempo $T(\lceil \frac{n}{5} \rceil)$.
- Partizionare l'array A intorno al perno richiede al più $(n - 1)$ confronti, come mostrato nel Paragrafo 4.5 del Capitolo 4.
- La chiamata ricorsiva che restituisce la soluzione viene eseguita su un insieme contenente al più $(\frac{7}{10}n + 3)$ elementi, come dimostrato nel Lemma 5.4.

Dimostriamo ora, usando il metodo di sostituzione introdotto nel Paragrafo 2.5.2 del Capitolo 2, che $T(n) \leq (cn - 4c)$ per una opportuna constante c. Nel nostro caso abbiamo:

$$T(n) \leq \frac{11}{5}n + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 3\right) \leq \frac{11}{5}n + T\left(\frac{n}{5} + 1\right) + T\left(\frac{7}{10}n + 3\right)$$

Per ipotesi induttiva risulta quindi

$$T(n) \leq \frac{11}{5}n + c\left(\frac{n}{5} + 1\right) - 4c + c\left(\frac{7n}{10} + 3\right) - 4c = n\left(\frac{11}{5} + \frac{9c}{10}\right) - 4c$$

Se vogliamo che questo sia al più $c \cdot n - 4c$, abbiamo bisogno che valga la disuguaglianza

$$n\left(\frac{11}{5} + \frac{9c}{10}\right) \leq c \cdot n$$

ovvero

$$\frac{11}{5} + \frac{9c}{10} \leq c$$

che implica $11/5 \leq c/10$ e quindi vale per $c \geq 22$. □

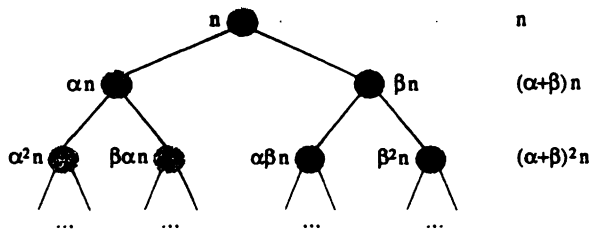


Figura 5.12 Albero della ricorsione corrispondente alla relazione di ricorrenza $T(n) \leq c \cdot n + T(\alpha n) + T(\beta n)$.

La dimostrazione del Teorema 5.4 mostra che l'algoritmo `select` esegue al più $22n$ confronti, ovvero richiede tempo lineare nel caso peggiore. Con tecniche più complesse, si può ridurre il numero dei confronti a $2.95n$ [7], che è meno del doppio dei confronti richiesti dalla selezione randomizzata, ma l'algoritmo è molto più complicato e meno efficiente in pratica.

Concludiamo questo paragrafo con alcune considerazioni sul tipo di relazione di ricorrenza che ci siamo trovati a risolvere per analizzare l'algoritmo `select`. La relazione di ricorrenza (5.2) non ha infatti la forma adatta per poter utilizzare il teorema fondamentale delle ricorrenze (Teorema 2.1 nel Capitolo 2), ma sembra comunque essere generata da un algoritmo di tipo *divide et impera* che divide il problema originario in due sottoproblemi in maniera non bilanciata. Tralasciando gli addendi costanti ed usando costanti moltiplicative generiche, possiamo considerare la ricorrenza (5.2) come una caso particolare della seguente relazione:

$$T(n) \leq c \cdot n + T(\alpha n) + T(\beta n) \tag{5.3}$$

dove $\alpha + \beta < 1$. Mostreremo ora, con un procedimento basato sull'analisi dell'albero della ricorsione, come risolvere questa ricorrenza. Il fatto che la somma delle parti, $\alpha n + \beta n$, sia strettamente minore della dimensione originaria, n, giocherà un ruolo determinante nell'analisi.

Teorema 5.5 *La relazione di ricorrenza (5.3), con $\alpha + \beta < 1$, ha soluzione*

$$T(n) \leq \frac{c}{1 - \alpha - \beta} n$$

Dimostrazione. Consideriamo l'albero delle chiamate ricorsive mostrato in Figura 5.12, indicando le dimensioni dei problemi di ogni chiamata ricorsiva, ed analizziamo la dimensione totale dei problemi ad ogni livello dell'albero. Nel nostro caso al primo livello abbiamo $(\alpha + \beta) n$, al secondo $(\alpha + \beta)^2 n$, e così via. Il tempo di esecuzione totale è quindi dato da:

$$T(n) \leq \left(c \cdot n + c(\alpha + \beta) n + c(\alpha + \beta)^2 n + \ldots\right) =$$

$$= c \cdot n \left(1 + (\alpha + \beta) + (\alpha + \beta)^2 + (\alpha + \beta)^3 + \ldots\right) \leq$$

$$\leq c \cdot n \sum_{i=0}^{\infty} (\alpha + \beta)^i = c \cdot n \frac{1}{1 - (\alpha + \beta)} = \frac{c}{1 - \alpha - \beta} n$$

Osserviamo che la precedente uguaglianza è una semplice applicazione della serie geometrica riportata in Appendice:

$$\sum_{i=0}^{k} d^i = \frac{d^{k+1} - 1}{d - 1}$$

con $|d| = \alpha + \beta < 1$ e per $k \to \infty$. □

5.4 Problemi

Problema 5.1 L'analisi dell'algoritmo `select` descritto nel Paragrafo 5.3 funzionerebbe se gli elementi fossero divisi in gruppi di 3, anziché 5? Quale sarebbe in tal caso il tempo di esecuzione? E se gli elementi fossero partizionati in gruppi di 7? Motivare le risposte.

Problema 5.2 Progettare un algoritmo per trovare il mediano di 5 elementi con 6 confronti. Dimostrare formalmente che l'algoritmo è corretto e disegnare il corrispondente albero di decisione.

Problema 5.3 Siano X ed Y due array ordinati, ciascuno di n elementi. Progettare un algoritmo per trovare il mediano dei $2n$ elementi in tempo $O(\log n)$.

Problema 5.4 Sia data, a scatola chiusa, una procedura in grado di trovare il mediano di n elementi in tempo $O(n)$ nel caso peggiore. Senza fare ipotesi sulla procedura, mostrare come utilizzarla per trovare il k-esimo elemento, per qualunque valore di $k \in [1, n]$, in tempo $O(n)$.

Problema 5.5 Il Professor De Sempliciottis sostiene che l'algoritmo mostrato in Figura 5.13 è il suo risultato di ricerca più entusiasmante.

- Cosa restituisce la chiamata `DeSempliciottis`$(A, 1, |A|, |A|/2)$?
- Scrivere una relazione di ricorrenza che descriva il tempo di esecuzione dell'algoritmo `DeSempliciottis` nel caso peggiore.
- Risolvendo la relazione di ricorrenza, dimostrare che l'algoritmo non risolve in modo ottimo il problema per cui è stato progettato.

Problema 5.6 (*) Dimostrare che $\lceil \frac{3}{2} n \rceil - 2$ confronti sono sufficienti per trovare il minimo ed il massimo di un insieme disordinato di n elementi.

```
algoritmo DeSempliciottis(array A, interi ℓ, r e k) → elem
1.   if ( ℓ = r ) then return A[ℓ]
2.   x ← DeSempliciottis(A, ℓ, (ℓ + r)/2, ⌈(r − ℓ + 1)/4⌉)
3.   partiziona A rispetto a x calcolando:
4.       A1 = { y ∈ A : y < x }
5.       A2 = { y ∈ A : y = x }
6.       A3 = { y ∈ A : y > x }
7.   if ( k ≤ |A1| ) then return DeSempliciottis(A1, 1, |A1|, k)
8.   else if ( k > |A1| + |A2| )
9.       then return DeSempliciottis(A3, 1, |A3|, k − |A1| − |A2|)
10.      else return x
```

Figura 5.13 L'algoritmo del Professor De Sempliciottis.

Problema 5.7 (**) Dimostrare che $\lceil \frac{3}{2} n \rceil - 2$ è una delimitazione inferiore per il problema del calcolo del massimo e del minimo, ovvero che $\lceil \frac{3}{2} n \rceil - 2$ confronti sono *necessari* per trovare il minimo ed il massimo di un qualunque insieme disordinato di n elementi.

Suggerimento: usare un'argomentazione basata su avversario. Immaginare che ci sia un avversario che fornisce le risposte ai confronti in modo da forzare l'algoritmo a lavorare il più possibile, facendo attenzione a non cadere mai in contraddizione. L'avversario classifica gli elementi in: (1) elementi ancora non confrontati; (2) elementi che hanno perso tutti i confronti; (3) elementi che hanno vinto tutti i confronti; e (4) elementi che hanno sia perso che vinto. La risposta alla domanda "$x < y$?" viene scelta dall'avversario in base alla classe cui x ed y appartengono.

5.5 Sommario

In questo capitolo abbiamo analizzato il problema del calcolo del mediano di un insieme di n elementi, ovvero dell'elemento che occuperebbe la posizione $\lceil n/2 \rceil$ se l'insieme fosse ordinato. Ne abbiamo dapprima introdotto una generalizzazione, affrontando il problema della selezione del k-esimo elemento, per ogni k nell'intervallo $[1, n]$. Partendo dall'algoritmo `quickSort` e tramite modifiche progressive, abbiamo derivato un algoritmo randomizzato che esegue un numero atteso di confronti lineare (Paragrafo 5.2). Poiché nel caso peggiore il numero di confronti potrebbe però essere $O(n^2)$, abbiamo successivamente affrontato il problema di progettare un algoritmo lineare deterministico (Paragrafo 5.3). Tale algoritmo si basa su una elegante tecnica di campionamento deterministico: il pemo attorno a cui partizionare viene scelto come mediano di un campione di $n/5$ elementi, e questi ultimi sono a loro volta i mediani di gruppi di 5 elementi.

Dimostrare che l'algoritmo basato sul mediano dei mediani ha tempo di esecuzione $O(n)$ nel caso peggiore ci ha permesso di introdurre una nuova tipologia di relazioni di ricorrenza derivanti da algoritmi di tipo *divide et impera* che non

può essere risolta tramite il teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2). Queste relazioni, che hanno la forma

$$T(n) \leq c \cdot n + T(\alpha n) + T(\beta n)$$

con $\alpha + \beta < 1$ e c costante, hanno soluzione $T(n) = O(n)$, come si può dimostrare analizzando l'albero della ricorsione o il metodo della sostituzione (Paragrafo 2.5.2 del Capitolo 2).

5.6 Note bibliografiche

Il problema della selezione è uno dei problemi di calcolo la cui storia è iniziata ancor prima della nascita dei moderni calcolatori. Il problema fu posto da Charles L. Dogson (meglio noto come Lewis Carroll, l'autore di "Alice nel paese delle meraviglie"), che nel 1883 notò che il secondo premio nei tornei di tennis era tipicamente assegnato in maniera non equa [3]. Nel 1929, Hugo Steinhaus pose il problema di trovare il minimo numero di incontri di tennis per determinare equamente sia il primo che il secondo vincitore in un torneo, problema che fu risolto nel 1932 da J. Schreier mostrando che $n + \lceil \log_2 n \rceil - 2$ incontri sono sempre sufficienti, dove n è il numero di giocatori [16]. Schreier cercò anche di dimostrare che questa quantità è ottima, ma la sua dimostrazione si rivelò errata: una dimostrazione corretta fu successivamente data nel 1964 da Kislitsyn [12].

L'algoritmo randomizzato lineare per il problema della selezione è attribuito a Hoare [11]; Floyd e Rivest hanno proposto la versione basata su campionamento [9], che esegue un numero atteso di confronti all'incirca proporzionale a $(n + k)$ per selezionare il k-esimo elemento. Canto e Munro dimostrarono poi che il risultato di Floyd e Rivest è sostanzialmente ottimo [4].

Diversamente dall'ordinamento, per cui algoritmi ottimi nel caso peggiore erano noti già dagli anni '40, è sorprendente che il primo algoritmo di selezione deterministico lineare fu proposto solo nel 1973 da Blum, Floyd, Pratt, Rivest e Tarjan [2]. Ma anche questo elegante risultato non ha affatto concluso la ricerca relativa a questo problema! La costante moltiplicativa è infatti piuttosto alta, e molti sforzi sono stati compiuti sia per progettare algoritmi con costanti più basse, sia per per derivare delimitazioni inferiori precise nel modello dei confronti [10]. Il risultato di Blum *et al.* è stato migliorato da Schönhage, Paterson, e Pippenger, che descrivono in [15] un algoritmo lineare con constante moltiplicativa pari a 3. Quest'ultimo è stato a sua volta migliorato riducendo la costante a 2.9423 [7]. In [2] si dimostra anche che $(1 + \alpha)n$ confronti sono necessari per trovare l'αn-esimo elemento, con $0 \leq \alpha \leq 1/2$. Bent e John [1] hanno poi dato una delimitazione inferiore più stretta, dimostrando che all'incirca $2n$ confronti sono necessari, ed il loro risultato è stato recentemente migliorato in [6] e [8].

Altri studi hanno affrontato il problema della selezione con operazioni diverse dai confronti binari [17] o il problema di selezionare contemporaneamente più di un elemento. Ad esempio, nel Problema 5.6 richiediamo di dimostrare che $\lceil \frac{3}{2} n \rceil - 2$ confronti sono sufficienti per trovare il minimo ed il massimo di un

insieme disordinato di n elementi: questa delimitazione è ottima, come provato da Pohl in [14] e come si richiede di dimostrare nel Problema 5.7. In [5] viene infine affrontato il problema di selezionare contemporaneamente il mediano ed i due quartili, ovvero gli elementi che occuperebbero le posizioni $n/4$ e $3n/4$ se l'insieme fosse ordinato.

Riferimenti bibliografici

[1] S. W. Bent, J. W. John, "Finding the median requires $2n$ comparisons", *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, 213–216, 1985.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, R. E. Tarjan, "Time bounds for selection", *Journal of Computer and System Sciences*, 7:448–461, 1973.

[3] L. Carroll, "Lawn tennis tournaments", *St. James's Gazette*, 5–6, 1883.

[4] W. Cunto, I. Munro, "Average Case Selection", *Journal of the ACM*, 36(2), 270–279, 1989.

[5] W. Cunto, I. Munro, M. Rey, "Selecting the median and two quartiles in a set of numbers", *Software – Practice and Experience*, 22(6), 439–454, 1992.

[6] D. Dor, J. Håstad, S. Ulfberg, U. Zwick, "On Lower Bounds for Selecting the Median", *SIAM Journal on Discrete Mathematics*, 14(3), 299–311, 1997.

[7] D. Dor, U. Zwick, "Selecting the Median", *Proceedings of the 6th ACM SIAM Symposium on Discrete Algorithms*, 28–37, 1995.

[8] D. Dor, U. Zwick, "Median Selection Requires $(2+\epsilon)n$ comparisons", *SIAM Journal on Discrete Mathematics*, 14(3), 312–325, 2001.

[9] R. W. Floyd, R. L. Rivest, "Expected time bounds for selection", *Communications of the ACM*, 18:165–172, 1975.

[10] L. R. Ford, S. M. Johnson, "A tournament problem", *American Mathematical Monthly*, 66:387–389, 1959.

[11] C. A. R. Hoare, "Algorithm 63 (partition) and algorithm 65 (find)", *Communications of the ACM*, 4(7):321–322, 1961.

[12] S. S. Kislitsyn, "On the selection of the k-th element of an ordered set by pairwise comparisons", *Sibirsk. Mat. Zh.*, 5:557–564, 1964.

[13] I. Munro e V. Raman, "Sorting multisets and vectors in place", *Proceedings of the 2nd Workshop on Algorithms and Data Structures*, 473–480, 1991.

[14] I. Pohl, "A sorting problem and its complexity", *Communications of the ACM*, 15, 462–464, 1972.

[15] A. Schönhage, M. Paterson, N. Pippenger, "Finding the median", *Journal of Computer and System Sciences*, 13:184–199, 1976.

[16] J. Schreier, "On tournament elimination systems", *Mathesis Polska*, 7:154–160, 1932 (in polacco).

[17] A. C. Yao, "On selecting the k largest with median tests", *Algorithmica*, 4:293–300, 1989.

6

Alberi di ricerca

Regis Iusfu Cantio Et Reliqua Canonica Arte Refolula.
(Johann Sebastian Bach)

In questo capitolo affronteremo in dettaglio il problema del dizionario introdotto nel Capitolo 3. Consideriamo un insieme S di elementi cui sono associate chiavi prese da un dominio totalmente ordinato. L'insieme è dinamico, nel senso che nuovi elementi possono essere inseriti in S (`insert`) oppure elementi esistenti possono essere cancellati da S (`delete`). Vorremmo anche poter verificare se un elemento con una data chiave appartiene a S (`search`). Ad esempio, S potrebbe essere l'insieme degli utenti di una compagnia telefonica: la chiave associata a ciascun utente potrebbe essere la coppia cognome - nome, e la relazione d'ordine è quindi data dall'ordinamento alfabetico di tali coppie. A ciascun utente possono essere associate informazioni supplementari, quali l'indirizzo e l'elenco dei numeri telefonici a suo carico. L'operazione `search`, in questo esempio, permette di verificare se un utente con un certo nome e cognome ha un contratto attivo con la compagnia telefonica: in caso affermativo, permette anche di ritrovare un recapito dell'utente (indirizzo e numero di telefono). Un *dizionario* è una struttura dati che supporta (almeno) le tre operazioni `search`, `insert` e `delete`: richiamiamo in Figura 6.1 il tipo di dato `Dizionario`, già mostrato nel Capitolo 3.

Abbiamo osservato nel Paragrafo 2.4.3 del Capitolo 2 che, quando l'insieme S è statico, possiamo facilmente supportare l'operazione `search` mediante l'algoritmo di ricerca binaria, e quindi in tempo $O(\log n)$, dove n è la cardinalità di S. Inserire o cancellare elementi costa però tempo $O(n)$ se pretendiamo di mantenere un insieme dinamico in un array ordinato. Viceversa, usando un array non ordinato (o una lista) gli inserimenti costerebbero tempo $O(1)$, ma ricerche e cancellazioni richiederebbero tempo $O(n)$. Rimandiamo al Paragrafo 3.1 nel Capitolo 3 per maggiori dettagli sull'implementazione del tipo `Dizionario` tramite semplici strutture indicizzate o collegate.

In questo capitolo vedremo varie implementazioni di dizionari basate su alberi di ricerca che supportano tutte e tre le operazioni in tempo logaritmico. Molte di queste strutture possono essere opportunamente modificate in modo da realizzare in modo efficiente anche altre operazioni, quali la ricerca del minimo o l'unione

tipo `Dizionario`:
dati: un insieme S di coppie $(elem, chiave)$
operazioni:
`insert`$(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k)

`delete`$(elem\ e)$
cancella da S l'elemento e

`search`$(chiave\ k) \rightarrow elem$
se la chiave k è presente in S restituisce un elemento e ad essa associato, e `null` altrimenti

Figura 6.1 Il tipo di dato `Dizionario`.

di insiemi. Una soluzione alternativa al problema del dizionario, non basata su alberi, è fornita dalle tavole hash, e sarà discussa nel Capitolo 7.

6.1 Alberi binari di ricerca

Prima di definire il concetto di albero binario di ricerca, che è alla base di molte delle realizzazioni di dizionari che studieremo in questo capitolo, ricordiamo l'algoritmo di ricerca binaria in un array ordinato X che abbiamo introdotto nel Paragrafo 2.4.3 del Capitolo 2. Al primo passo, confrontando l'elemento da ricercare con l'elemento $X[n/2]$, ci era possibile restringere la ricerca nella metà inferiore o in quella superiore dell'array X. La procedura veniva poi applicata ricorsivamente. Immaginiamo ora di associare all'array ordinato X un albero binario T così costruito: la radice di T contiene il valore $X[n/2]$, mentre i valori più piccoli e più grandi di $X[n/2]$ sono assegnati rispettivamente ai nodi del sottoalbero sinistro e destro della radice di T, in modo ricorsivo. È facile vedere che una visita in ordine simmetrico dell'albero così ottenuto corrisponde ad una scansione di X dall'elemento più piccolo a quello più grande, e che ricercare in X un elemento corrisponde a tracciare un cammino dalla radice ad una foglia di T. La corrispondenza appena vista suggerisce la seguente definizione di albero binario di ricerca.

Definizione 6.1 (Albero binario di ricerca) *Un* albero binario di ricerca *è un albero binario che soddisfa le seguenti proprietà:*

1. *ogni nodo v contiene un elemento $elem(v)$ cui è associata una chiave $chiave(v)$ presa da un dominio totalmente ordinato;*
2. *le chiavi nel sottoalbero sinistro di v sono $\leq chiave(v)$;*
3. *le chiavi nel sottoalbero destro di v sono $\geq chiave(v)$.*

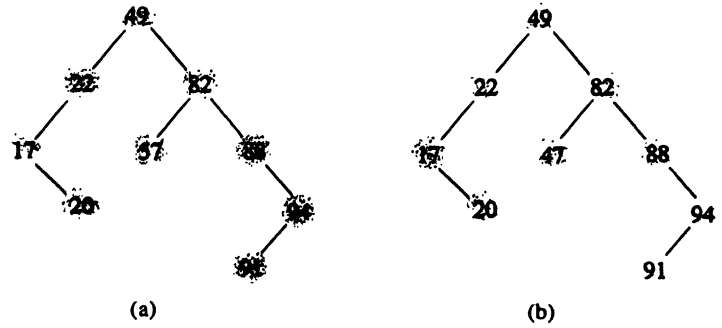


Figura 6.2 (a) Un albero binario di ricerca; (b) un albero binario non di ricerca. Per semplicità, mostriamo solo la chiave associata ad ogni nodo.

Le chiavi possono essere valori di qualunque tipo, purché su di esse sia definita una relazione d'ordine totale. I punti 2 e 3 nella Definizione 6.1 sono noti come *proprietà di ricerca*, e garantiscono che visitando un albero binario di ricerca in ordine simmetrico (come descritto nel Capitolo 3) si ottengano le chiavi in ordine non decrescente. Un esempio di albero binario di ricerca è mostrato in Figura 6.2(a). Il nodo con chiave 47 nell'albero in Figura 6.2(b), invece, non soddisfa la proprietà di ricerca rispetto alla radice.

Le idee di alto livello di una realizzazione di un dizionario tramite alberi binari di ricerca sono descritte in Figura 6.3. Discuteremo ora in maggior dettaglio come le singole operazioni possano essere implementate. Presenteremo anche la realizzazione di due operazioni per il calcolo del massimo di un sottoalbero e del predecessore di un nodo, utili per implementare l'operazione `delete`.

`search`$(chiave\ k) \rightarrow elem$

Grazie alla proprietà di ricerca, implementare l'operazione `search` è molto semplice. Confrontiamo la chiave x che stiamo cercando con la chiave $chiave(v)$ della radice v dell'albero di ricerca. Se sono uguali, abbiamo trovato l'elemento. Altrimenti, proseguiamo la ricerca nel sottoalbero sinistro se $x < chiave(v)$ o in quello destro se $x > chiave(v)$. Ripetiamo la ricerca a partire dal figlio sinistro o destro della radice usando questa stessa strategia, finché troviamo x oppure arriviamo ad un albero vuoto, nel qual caso $x \notin S$. Lo pseudocodice dell'algoritmo `search` è riportato in Figura 6.4. Poiché dopo ogni confronto, se non abbiamo trovato l'elemento, scendiamo di un livello nell'albero T, il tempo richiesto dalla ricerca è $O(h)$, dove h è l'altezza di T. Osserviamo che se l'albero è molto profondo e sbilanciato, nel caso peggiore una ricerca costerà tempo $O(n)$, il che equivale ad usare una lista o un array disordinato.

Poiché l'approccio utilizzato dalla procedura `search` ricorda da vicino la ricerca binaria descritta nel Paragrafo 2.4.3 del Capitolo 2, è naturale chiedersi perché il tempo non sia $O(\log n)$. Osserviamo innanzitutto che gli insiemi di

classe `AlberoBinarioDiRicerca` **implementa** `Dizionario`:
dati: $S(n) = O(n)$
un albero binario di ricerca T di altezza h e con n nodi, ciascuno contenente coppie $(elem, chiave)$ in cui le chiavi sono prese da un universo totalmente ordinato.
operazioni:
`search`$(chiave\ k) \rightarrow elem$ $T(h) = O(h)$
partendo dalla radice, traccia un cammino nell'albero per cercare un elemento con chiave k. Su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro.
`insert`$(elem\ e, chiave\ k)$ $T(h) = O(h)$
crea un nuovo nodo v contenente la coppia (e, k), e lo aggiunge all'albero come foglia nella posizione opportuna, in modo da mantenere la proprietà di ricerca.
`delete`$(elem\ e)$ $T(h) = O(h)$
se il nodo v contenente l'elemento e ha al più un figlio, elimina v collegando il figlio all'eventuale padre. Altrimenti scambia il nodo v con il suo predecessore ed elimina il predecessore.

Figura 6.3 Dizionario realizzato mediante alberi binari di ricerca.

valori $\geq X[n/2]$ e $\leq X[n/2]$ hanno cardinalità bilanciate, pari a circa $n/2$, e ricordiamo come è stato costruito l'albero T che abbiamo immaginato di associare all'array ordinato X: T ha un nodo al livello 0 (la radice), 2 nodi al livello 1, $2^2 = 4$ nodi al livello 2, e via dicendo. Sia h la profondità dell'ultimo livello, che contiene al più 2^h nodi. Poiché deve risultare $2^h \leq n$, si ottiene $h = O(\log n)$. Il numero di confronti eseguiti per effettuare una qualsiasi ricerca sull'albero T così costruito è quindi $O(\log n)$ nel caso peggiore, poiché l'altezza di T è limitata. Una tale limitazione non è però vera per un qualunque albero binario di ricerca.

`insert`$(elem\ e, chiave\ k)$

Un nuovo nodo con elemento e e chiave k viene sempre inserito come foglia dell'albero di ricerca. L'operazione `insert` può quindi essere implementata in due passi:

Passo 1. Cerca il nodo v che diventerà genitore del nuovo nodo.

Passo 2. Crea un nuovo nodo u con elemento e e chiave k ed appendilo come figlio sinistro o destro di v rispettando la proprietà di ricerca.

Il Passo 1 equivale ad effettuare una ricerca della chiave k nell'albero, e richiede quindi tempo $O(h)$. Rappresentando l'albero tramite puntatori ai figli come descritto nel Paragrafo 3.3 del Capitolo 3, il Passo 2 richiede di modificare solo un numero costante di puntatori e può essere implementato in tempo $O(1)$. In totale, anche il costo dell'inserimento è quindi $O(h)$.


```
algoritmo search(chiave k) → elem
    v ← radice di T
    while ( v ≠ null ) do
        if ( k = chiave(v) ) then return elem(v)
        else if ( k < chiave(v) ) then v ← figlio sinistro di v
        else v ← figlio destro di v
    return null
```

Figura 6.4 Implementazione iterativa dell'operazione `search` in un albero binario di ricerca.

`max`$(nodo\ u) \rightarrow nodo$

Grazie alla proprietà di ricerca, per trovare il massimo di un sottoalbero di T radicato in un qualunque nodo u basta scendere verso destra nell'albero finché possibile, partendo da u, come mostrato in Figura 6.5. Il tempo di esecuzione è chiaramente $O(h)$.

`pred`$(nodo\ u) \rightarrow nodo$

Il predecessore di un nodo u è un nodo v avente massima chiave $\leq chiave(u)$. Per trovare il predecessore di u distinguiamo due casi:

1. u ha un figlio sinistro: in tal caso $pred(u)$ è il massimo del sottoalbero sinistro di u;
2. u non ha un figlio sinistro: $pred(u)$, se esiste, è il più basso antenato di u (ovvero, l'antenato di u con massima profondità nell'albero) il cui figlio destro è anch'esso antenato di u. Per trovarlo, risaliamo da u verso la radice fino ad incontrare la prima "svolta a sinistra", come mostrato in Figura 6.7.

Lo pseudocodice dell'algoritmo `pred` per la ricerca del predecessore è mostrato in Figura 6.6. Il tempo di esecuzione è chiaramente $O(h)$.

`delete`$(elem\ e)$

Come avviene in molte strutture dati, cancellare è più difficile di inserire. In

```
algoritmo max(nodo u) → nodo
    v ← u
    while ( figlio destro di v ≠ null ) do
        v ← figlio desiro di v
    return v
```

Figura 6.5 Ricerca del nodo con valore massimo nel sottoalbero dell'albero binario di ricerca T radicato nel nodo u.

```
algoritmo pred(nodo u) → nodo
    if ( u ha figlio sinistro sin(u) ) then
        return max(sin(u))
    while ( parent(u) ≠ null e u è figlio sinistro di suo padre ) do
        u ← parent(u)
    return parent(u)
```

Figura 6.6 Ricerca del predecessore di un nodo in un albero binario di ricerca.

particolare, per implementare l'operazione `delete` useremo la procedura `pred` per la ricerca del predecessore di un nodo. Sia u il nodo contenente l'elemento e da cancellare. Distinguiamo tre casi.

1. u è una foglia: in tal caso basta distaccare la foglia dal genitore ed eliminarla;
2. u ha un unico figlio: sia v l'unico figlio di u. Se u è radice, v diviene la nuova radice dell'albero. Altrimenti, dopo aver individuato il genitore w di u, l'arco (w, u) viene sostituito dall'arco (w, v) come mostrato in Figura 6.8;
3. u ha due figli: ci si riconduce ad uno dei casi precedenti operando come segue. Si individua il predecessore di u, diciamo v: tale predecessore è certamente il massimo del sottoalbero sinistro di u, poiché u ha due figli. Osserviamo che v non può avere due figli, altrimenti non sarebbe predecessore. Dopo aver copiato $chiave(v)$ in $chiave(u)$, è quindi possibile cancellare fisicamente v dall'albero applicando uno dei due casi precedenti. La Figura 6.9 mostra un esempio.

È facile verificare che la proprietà di ricerca è sempre mantenuta. Inoltre, poiché la cancellazione di un nodo interno richiede l'individuazione del nodo da cancellare nonché del suo predecessore, il costo della cancellazione è $O(h)$.

Il seguente teorema riassume le prestazioni degli alberi binari di ricerca:

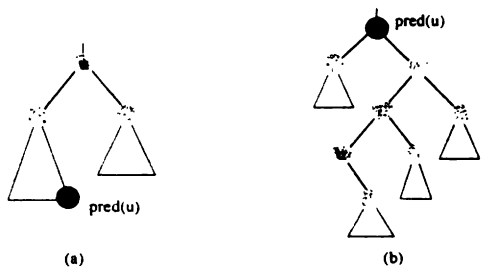


Figura 6.7 I due casi nella ricerca del predecessore del nodo u: (a) u ha un figlio sinistro; (b) u non ha un figlio sinistro.

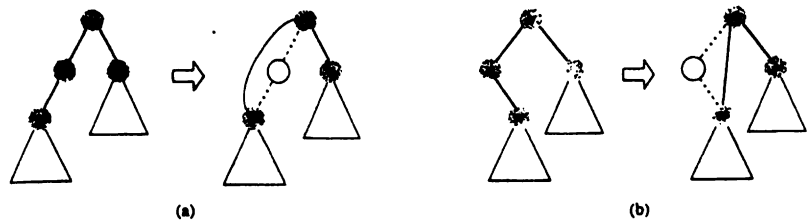


Figura 6.6 Cancellazione di un nodo con un unico figlio da un albero binario di ricerca.

Teorema 6.1 *Alberi binari di ricerca di altezza h sono in grado di supportare operazioni* `search`, `insert` *e* `delete` *in tempo $O(h)$.*

Abbiamo già osservato che nel caso peggiore l'altezza dell'albero può essere proporzionale al numero n di nodi. Se l'albero fosse bilanciato, però, avrebbe altezza logaritmica, e quindi potremmo sperare di implementare le varie operazioni in tempo $O(\log n)$. Nel prossimo paragrafo vedremo una definizione formale di bilanciamento e presenteremo una tecnica basata su rotazioni di sottoalberi utile per mantenere il bilanciamento a fronte di cambiamenti dinamici dell'insieme S.

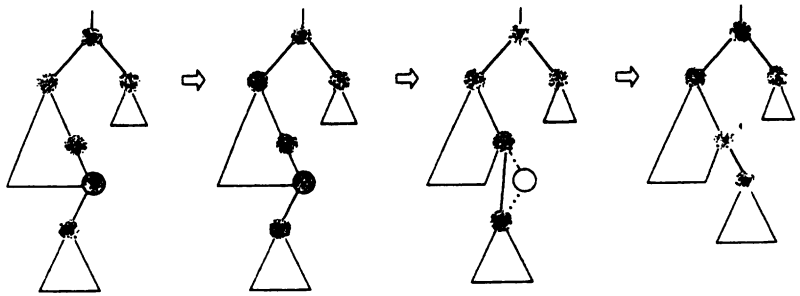


Figura 6.9 Cancellazione di un nodo con due figli da un albero binario di ricerca.

6.2 Alberi AVL

Dal Teorema 6.1 si evince che alberi di ricerca bilanciati garantiscono un tempo di ricerca logaritmico. Pur partendo da un albero bilanciato, però, a fronte di inserimenti e cancellazioni il bilanciamento dell'albero potrebbe perdersi, e quindi con l'aumentare del numero di operazioni effettuate le prestazioni del dizionario degraderebbero. È quindi opportuno mantenere il bilanciamento anche quando si inseriscono e si cancellano elementi. In questo paragrafo vedremo come ciò sia possibile. Innanzitutto, sottolineiamo che esistono varie definizioni formali di

bilanciamento, che legano ad esempio l'altezza dell'albero con il numero di nodi o le altezze dei suoi sottoalberi. Useremo qui l'idea di bilanciamento in altezza definita come segue.

Definizione 6.2 (Bilanciamento in altezza) *Un albero è* bilanciato in altezza *se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono di al più un'unità.*

Gli alberi binari di ricerca bilanciati in altezza sono anche detti *alberi AVL*, dai nomi degli ideatori Adel'son-Vel'skiĭ e Landis, che li hanno introdotti nel 1962 [1]. In un albero AVL, oltre all'elemento e alla chiave, ciascun nodo mantiene anche un'informazione sul bilanciamento così definita.

Definizione 6.3 (Fattore di bilanciamento) *Il fattore di bilanciamento $\beta(v)$ di un nodo v è la differenza tra l'altezza del sottoalbero sinistro e quella del sottoalbero destro di v:*

$$\beta(v) = altezza(sin(v)) - altezza(des(v))$$

Il fattore di bilanciamento è tanto migliore quanto più basso è il suo valore assoluto, mentre è uguale all'altezza dell'albero quando questo degenera in lista. In particolare, in un albero binario completo, il fattore di bilanciamento è 0 su ogni nodo. Un albero binario di ricerca è quindi un albero AVL, in base alle Definizioni 6.2 e Definizioni 6.3, se il valore assoluto del fattore di bilanciamento è ≤ 1 su ogni nodo.

6.2.1 Altezza di un albero AVL

Dato lo stretto vincolo sui fattori di bilanciamento dei nodi, è lecito aspettarsi che l'altezza di un albero AVL con n nodi sia $O(\log n)$. Ed infatti ciò è vero, anche se dimostrarlo non è così immediato. Per farlo, tra tutti i possibili alberi bilanciati in altezza, studieremo l'altezza di quelli più sbilanciati possibile.

Definizione 6.4 *Tra tutti gli alberi di altezza h bilanciati in altezza, un* albero di Fibonacci *ha il minimo numero di nodi.*

Ben presto scopriremo da cosa derivi il nome *albero di Fibonacci*. Osserviamo che un albero di Fibonacci di altezza h può essere costruito unendo, tramite l'aggiunta di una radice, un albero di Fibonacci di altezza $h-1$ (ad esempio come sottoalbero sinistro) ed un albero di Fibonacci di altezza $h-2$ (ad esempio come sottoalbero destro), come mostrato in Figura 6.10. La figura mostra anche alcuni esempi di alberi di Fibonacci di altezza da 0 a 4.

È facile verificare che il fattore di bilanciamento di ogni nodo interno di un albero di Fibonacci costruito come in Figura 6.10 è $+1$, e quindi questi alberi sono gli alberi bilanciati in altezza più vicini alla condizione di non bilanciamento. Studieremo ora il rapporto tra l'altezza ed il numero di nodi di un albero di Fibonacci.

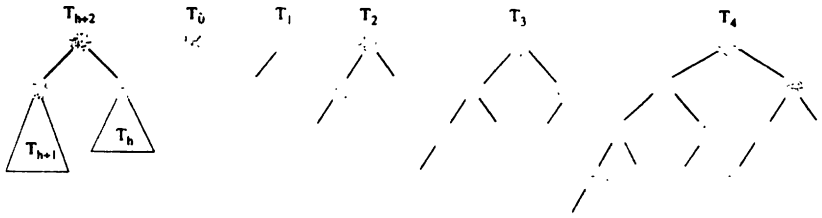


Figura 6.10 Costruzione ricorsiva di un albero di Fibonacci di altezza h ed esempi di alberi di Fibonacci di altezza da 0 a 4.

Lemma 6.1 *Sia T_h un albero di Fibonacci di altezza h e sia n_h il numero dei suoi nodi. Risulta $h = \Theta(\log n_h)$.*

Dimostrazione. Per costruzione di T_h, come mostrato in Figura 6.10, risulta $n_h = 1 + n_{h-1} + n_{h-2}$. Questo ricorda molto da vicino l'equazione dei numeri di Fibonacci: $F_i = F_{i-1} + F_{i-2}$ che abbiamo già incontrato nel Capitolo 1. E infatti dimostreremo per induzione che

$$n_h = F_{h+3} - 1$$

Il passo base, per $h = 0$, è banalmente verificato, essendo $n_0 = 1 = F_3 - 1 = 2 - 1$. Assumendo per ipotesi induttivamente che $n_k = F_{k+3} - 1$ per ogni $k < h$, ed usando le ricorrenze relative ad n_h ed a F_i, si ha:

$$n_h = 1 + n_{h-1} + n_{h-2} = 1 + F_{h+2} - 1 + F_{h+1} - 1 = F_{h+3} - 1$$

Ricordiamo ora dal Capitolo 1 che $F_h = \Theta(\phi^h)$, dove $\phi \approx 1.618$ è la sezione aurea (per una stima precisa di F_n in termini della costante ϕ rimandiamo all'Equazione 17.13 riportata in Appendice). L'altezza ed il numero di nodi di T_h sono quindi esponenzialmente correlate, e pertanto $h = \Theta(\log n_h)$. □

Adel'son-Vel'skiĭ e Landis hanno dimostrato delle stime ancor più precise sull'altezza di un albero di Fibonacci con n_h nodi. La stima asintotica data nel Lemma 6.1 ci permette comunque di concludere che un qualunque albero AVL ha altezza logaritmica nel numero di nodi, come dimostrato nel seguente corollario.

Corollario 6.1 *Un albero AVL con n nodi ha altezza $O(\log n)$.*

Dimostrazione. Sia h l'altezza dell'albero AVL. Per dimostrare che $h = O(\log n)$, consideriamo l'albero di Fibonacci di altezza h. Sia n_h il numero di nodi di tale albero. Per definizione di albero di Fibonacci, $n_h \leq n$. Il fatto che $h = O(\log n_h)$ in base al Lemma 6.1 completa la dimostrazione. □

Il Corollario 6.1 implica che la ricerca in un albero AVL, effettuata tramite lo stesso algoritmo `search` utilizzato per alberi binari di ricerca qualunque, ha costo $O(\log n)$. Dobbiamo però ancora vedere come mantenere il bilanciamento a fronte di inserimenti e cancellazioni: questo sarà l'argomento dei prossimi paragrafi.

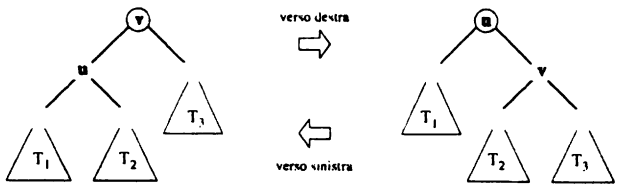


Figura 6.11 Rotazioni di base: le rotazioni verso sinistra e verso destra sono simmetriche. Il nodo v è il perno della rotazione verso destra, mentre il nodo u è il perno della rotazione verso sinistra.

6.2.2 Ribilanciamento tramite rotazioni

Poiché inserimenti e cancellazioni potrebbero far perdere il bilanciamento in un albero AVL, dovremo ripristinarlo usando opportune rotazioni. La Figura 6.11 mostra la *rotazione di base*: si può ruotare su un nodo perno o verso destra o verso sinistra, e la figura mostra che i due casi sono perfettamente simmetrici. La proprietà di ricerca è mantenuta dopo la rotazione: infatti l'ordine relativo delle chiavi nei sottoalberi T_1, T_2 e T_3 e nei nodi u e v rimane invariato. Una rotazione di base è anche detta rotazione *semplice*. Componendo opportunamente due rotazioni di base otteniamo una rotazione *doppia*.

Come vedremo, le rotazioni vengono effettuate su nodi sbilanciati, ovvero su nodi il cui fattore di bilanciamento in valore assoluto è ≥ 2. Sia v un tale nodo con fattore di bilanciamento ± 2. Intuitivamente, esisterà un sottoalbero di v che è "troppo alto", e che quindi sbilancia il nodo. A seconda della posizione di questo sottoalbero, che chiameremo T, possiamo avere quattro casi:

Sinistra - sinistra	(SS)	T è il sottoalbero sinistro del figlio sinistro di v
Destra - destra	(DD)	T è il sottoalbero destro del figlio destro di v
Sinistra - destra	(SD)	T è il sottoalbero destro del figlio sinistro di v
Destra - sinistra	(DS)	T è il sottoalbero sinistro del figlio destro di v

Poiché questi casi sono simmetrici a coppie (SS con DD, e SD con DS), nel seguito vedremo solo come trattare SS e SD.

SS: per ribilanciare il nodo v basterà applicare una rotazione semplice verso destra su v, il cui fattore di bilanciamento passerà da 2 a 0. Si noti che l'altezza del sottoalbero coinvolto nella rotazione è $h + 3$ prima della rotazione (per via del sottoalbero T_1), ed è $h + 2$ immediatamente dopo.

SD: questo caso è più complesso e richiede l'applicazione di una rotazione doppia. Sia z il figlio sinistro di v: l'albero T che sbilancia v è il sottoalbero destro di z, radicato nel nodo w. La rotazione SD consiste in una rotazione di base verso sinistra con perno in z seguita da una rotazione di base verso destra con perno in v. La Figura 6.13 mostra l'effetto finale di tale rotazione doppia: la figura presenta due casi perché, a seconda del fatto che lo sbilancimento sia causato dal sottoalbero sinistro o destro di w, i valori dei fattori di bilanciamento dei nodi coinvolti

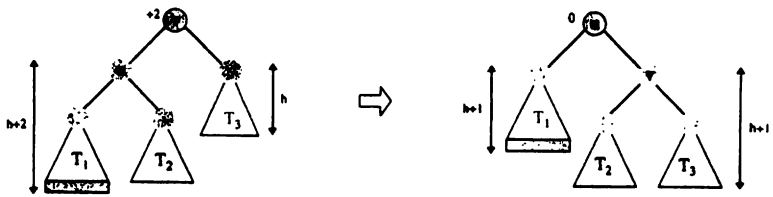


Figura 6.12 Rotazione SS: il rettangolo evidenzia il sottoalbero che sbilancia il nodo v.

nella rotazione saranno diversi. Anche in questo caso l'altezza del sottoalbero coinvolto nella rotazione è $h + 3$ prima e diventa $h + 2$ dopo. Possiamo quindi enunciare la seguente proprietà, che ci sarà utile in seguito:

Proprietà 6.1 *Un rotazione SS, SD, DD, o DS applicata ad un nodo v con fattore di bilanciamento ± 2 fa decrescere di 1 l'altezza del sottoalbero radicato in v prima della rotazione.*

6.2.3 Modifiche del dizionario

In questo paragrafo mostreremo come usare le rotazioni per ripristinare il bilanciamento dell'albero AVL dopo un inserimento o una cancellazione. Sottolineiamo che ogni rotazione (sia semplice che doppia) richiede tempo $O(1)$. Poiché è importante mantenere il tempo di esecuzione delle operazioni `insert` e `delete` proporzionale all'altezza dell'albero (come nel Teorema 6.1), cercheremo di limitare il numero di rotazioni eseguite a fronte di ciascuna operazione.

`insert`(*elem e, chiave k*)

L'inserimento avviene in tre passi:

Passo 1. Si crea un nuovo nodo u con elemento e e chiave k e lo si inserisce nell'albero come descritto nel Paragrafo 6.1: il nodo diviene quindi una foglia.

Passo 2. Si ricalcolano i fattori di bilanciamento che sono mutati in seguito all'inserimento. Sia r la radice dell'albero AVL. Osserviamo che solo i fattori di bilanciamento dei nodi nel cammino da r ad u possono mutare, e questi possono essere facilmente ricalcolati risalendo nel cammino dalla foglia verso la radice.

Passo 3. Se nel cammino da r ad u appare un fattore di bilanciamento pari a ± 2, occorre ribilanciare tramite rotazioni.

Dettagliamo ora meglio il Passo 3. Sia v il più profondo nodo con fattore di bilanciamento pari a ± 2, che chiameremo *nodo critico*: il ribilanciamento consiste nell'eseguire una rotazione con perno in v, secondo i casi descritti nel Paragrafo 6.2.2. Dimostriamo ora la correttezza di questo procedimento:

Lemma 6.2 *Una rotazione semplice o doppia sul nodo critico è sufficiente a ribilanciare in altezza un albero AVL dopo l'inserimento di un nuovo elemento.*

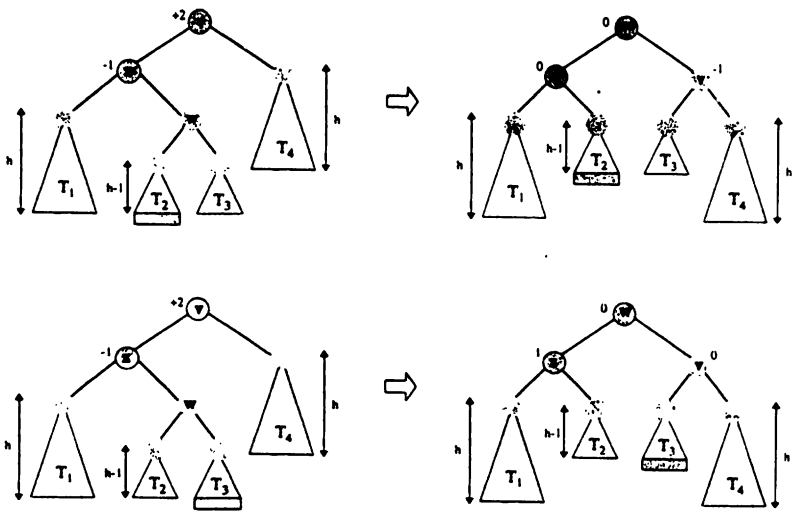


Figura 6.13 Rotazione SD: i due casi sono trattati analogamente, ma mostrano che i fattori di bilanciamento dei nodi coinvolti nella rotazione sono diversi.

Dimostrazione. Sia T_v l'albero radicato nel nodo critico v: T_v si sbilancia perché abbiamo inserito un nodo in un suo sottoalbero la cui altezza è aumentata di 1. Chiamiamo h_1 l'altezza di T_v prima dell'inserimento; h_2 l'altezza di T_v dopo l'inserimento ma prima della rotazione; ed h_3 l'altezza di T_v dopo la rotazione. Risulta $h_2 = h_1 + 1$. Inoltre, per la Proprietà 6.1 si ha $h_3 = h_2 - 1 = h_1$. In totale, l'altezza di T_v non varia, e quindi nessun antenato vede il proprio fattore di bilanciamento mutare. □

Il Lemma 6.2 implica che una volta identificato il nodo critico ed applicata l'opportuna rotazione, non occorre risalire nell'albero ed effettuare altre rotazioni, perché i fattori di bilanciamento più in alto non variano. Ricordiamo dal Corollario 6.1 che l'altezza dell'albero è logaritmica nel numero di nodi. Il costo di un inserimento è pertanto $O(\log n)$, essendo dato dalla somma del costo $O(\log n)$ per i Passi 1 e 2, e del costo $O(1)$ per il Passo 3.

`delete`(*elem e*)

Anche la cancellazione avviene in tre passi:

Passo 1. Si cancella il nodo come descritto nel Paragrafo 6.1.

Passo 2. Si ricalcolano i fattori di bilanciamento che sono mutati in seguito alla cancellazione. Osserviamo che solo i fattori di bilanciamento dei nodi nel cammino dalla radice al padre del nodo eliminato possono mutare, e questi possono essere ricalcolati risalendo nell'albero dal basso verso l'alto.

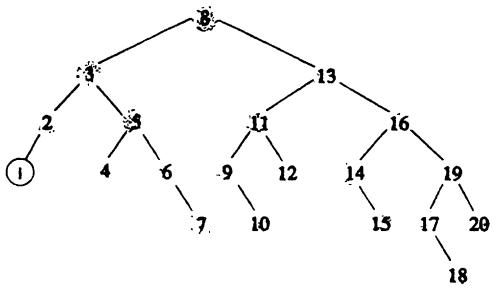


Figura 6.14 La cancellazione del nodo con chiave 1 genera una serie di rotazioni a cascata.

Passo 3. Per ogni nodo con fattore di bilanciamento pari a ± 2, procedendo dal basso verso l'alto, si opera una rotazione semplice o doppia.

Osserviamo che, diversamente dall'inserimento, nel Passo 3 potremmo eseguire $O(\log n)$ rotazioni nel caso peggiore. Sia infatti T_v l'albero radicato nel nodo critico v. Sicuramente T_v si sbilancia perché abbiamo cancellato un nodo da un suo sottoalbero la cui altezza diminuisce di 1. Chiamiamo h_1 l'altezza di T_v prima dell'eliminazione del nodo; h_2 l'altezza di T_v dopo l'eliminazione del nodo ma prima della rotazione; ed h_3 l'altezza di T_v dopo la rotazione. Diversamente dall'inserimento, risulta $h_1 = h_2$. Inoltre, per la Proprietà 6.1 si ha $h_3 = h_2 - 1 = h_1 - 1$. Quindi, in totale l'altezza di T_v diminuisce e qualche antenato di v con fattore di bilanciamento ± 1 potrebbe risentirne e sbilanciarsi a sua volta. Un esempio in cui lo sbilanciamento si propaga a cascata verso l'alto è mostrato in Figura 6.14: basta cancellare il nodo con chiave 1 ed osservare cosa accade.

Pur essendo un po' più complessa dell'inserimento, la cancellazione ha comunque lo stesso costo asintotico $O(\log n)$. Infatti potranno essere eseguite al più $O(\log n)$ rotazioni, tante quante l'altezza dell'albero, e ciascuna di essere richiede tempo $O(1)$. Esistono altri tipi di alberi binari di ricerca bilanciati, quali gli alberi *red-black*, in cui anche le cancellazioni possono essere implementate eseguendo un numero costante di rotazioni.

Mostriamo in Figura 6.15 la struttura di una possibile classe `AlberoAVL` derivata dalla classe base `AlberoBinarioDiRicerca`. L'unica informazione che ogni nodo dell'albero ha in più rispetto ai nodi di un albero binario di ricerca è il fattore di bilanciamento. L'operazioni `search` è direttamente ereditata dalla classe base. Le operazioni di aggiornamento `insert` e `delete` sono invece realizzate richiamando prima le corrispondenti operazioni ereditate, che eseguono il grosso del lavoro, e poi ricalcolando i fattori di bilanciamento nel cammino critico e ribilanciando tramite opportune rotazioni. Riassumiamo le prestazioni degli alberi AVL nel seguente teorema:

classe `AlberoAVL` **estende** `AlberoBinarioDiRicerca`:
dati: $S(n) = O(n)$
albero binario di ricerca T ereditato, più il fattore di bilanciamento di ogni nodo.

operazioni:
`search`(*chiave k*) $\rightarrow$ *elem* $T(n) = O(\log n)$
ereditata.

`insert`(*elem e*, *chiave k*) $T(n) = O(\log n)$
chiama `insert()` ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(1)$ rotazioni.

`delete`(*elem e*) $T(n) = O(\log n)$
chiama `delete()` ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(\log n)$ rotazioni.

Figura 6.15 Dizionario `AlberoAVL` come estensione della classe `AlberoBinarioDiRicerca` di Figura 6.3.

Teorema 6.2 *Alberi AVL con n nodi sono in grado di supportare operazioni* `search`, `insert` *e* `delete` *in tempo $O(\log n)$ nel caso peggiore.*

6.3 * Alberi auto-aggiustanti

Gli alberi AVL descritti nel Paragrafo 6.2 mantengono esplicitamente una condizione di bilanciamento, basata sul fattore di bilanciamento dei nodi, per ottenere un tempo di esecuzione logaritmico nel caso peggiore per tutte le operazioni. Se siamo disposti ad accettare tempi di esecuzione logaritmici ammortizzati su una sequenza di operazioni, non abbiamo bisogno di mantenere alcuna condizione di bilanciamento in maniera esplicita, come vedremo in questo paragrafo. Descriveremo infatti una struttura dati che gode di questa proprietà: gli *alberi binari di ricerca auto-aggiustanti* (o anche *splay trees*) dovuti a Sleator e Tarjan [10]. L'idea fondamentale degli splay trees è che l'albero viene automaticamente riaggiustato ogni volta che si esegue una qualunque operazione: in particolare, l'elemento cui si accede viene fatto risalire alla radice ed i sottoalberi incontrati lungo il cammino sono riposizionati tramite opportune rotazioni.

6.3.1 L'operazione splay

L'euristica di ristrutturazione proposta da Sleator e Tarjan è anche nota come operazione `splay`. Lo splay di un nodo u consiste nel partire da u e risalire fino alla radice dell'albero eseguendo una sequenza di rotazioni. Per brevità, denoteremo con $p(u)$ il padre di u nell'albero. Chiameremo inoltre *nonno* di u, e lo indicheremo con $p^2(u)$, il padre di $p(u)$. In ciascun passo di splay, la rotazione da eseguire viene scelta come segue:

- se u è figlio della radice, ruotiamo su $p(u)$;
- se u ha un nonno, $p^2(u)$, ed u e $p(u)$ sono entrambi figli sinistri o destri, ruotiamo prima su $p^2(u)$ e poi su $p(u)$;
- se u ha un nonno, $p^2(u)$, ed u è figlio sinistro mentre $p(u)$ è figlio destro o viceversa, ruotiamo su $p(u)$ e poi sul nuovo genitore di u (ovvero $p^2(u)$ prima della rotazione).

I vari casi sono illustrati in Figura 6.16. Ciascun passo di splay richiede tempo $O(1)$. L'effetto dell'euristica di ristrutturazione è di far risalire u alla radice, riarrangiando il resto del cammino. Per rendere un albero binario di ricerca auto-aggiustante, eseguiamo l'operazione splay in corrispondenza di ciascuna operazione di accesso o di modifica dell'albero. In particolare:

- dopo aver cercato una chiave, eseguiamo uno splay sul nodo che contiene la chiave o sulla foglia su cui la ricerca è terminata;
- dopo aver inserito un elemento, eseguiamo uno splay sul nuovo nodo che contiene l'elemento;
- dopo aver eliminato un nodo v (che, per quanto detto nel Paragrafo 6.1, potrebbe essere il predecessore dell'elemento di cui si richiedeva la cancellazione), eseguiamo uno splay sul genitore di v appena prima dell'eliminazione.

In ciascun caso, il tempo dell'operazione è proporzionale alla lunghezza del cammino su cui lo splay procede. Mostriamo in Figura 6.17 la struttura di una possibile classe `AlberoSplay` derivata dalla classe base `AlberoBinarioDiRicerca`. Osserviamo che non esistono informazioni aggiuntive a carico dei nodi o degli archi di T, come invece avveniva nel caso degli alberi AVL. Tutte le operazioni sono realizzate richiamando prima le corrispondenti operazioni ereditate, e poi eseguendo l'euristica `splay` su un nodo opportunamente scelto, a seconda dell'operazione. Il tempo ammortizzato delle operazioni su una sequenza è $O(\log n)$, come dimostreremo nel prossimo paragrafo. L'effetto dell'euristica splay è illustrato in Figura 6.18.

6.3.2 Analisi basata sul potenziale

Mostreremo ora che il costo ammortizzato di una operazione di `splay` è $O(\log n)$. Useremo il metodo del potenziale introdotto nel Paragrafo 2.7 del Capitolo 2.

Definizione 6.5 *Sia T uno splay tree. Per ogni nodo x, sia $d(x)$ il numero di discendenti di x, incluso se stesso. Definiamo rank di x il valore $r(x) = \log d(x)$ e la funzione potenziale come*

$$\Phi(T) = \sum_{x \in T} r(x)$$

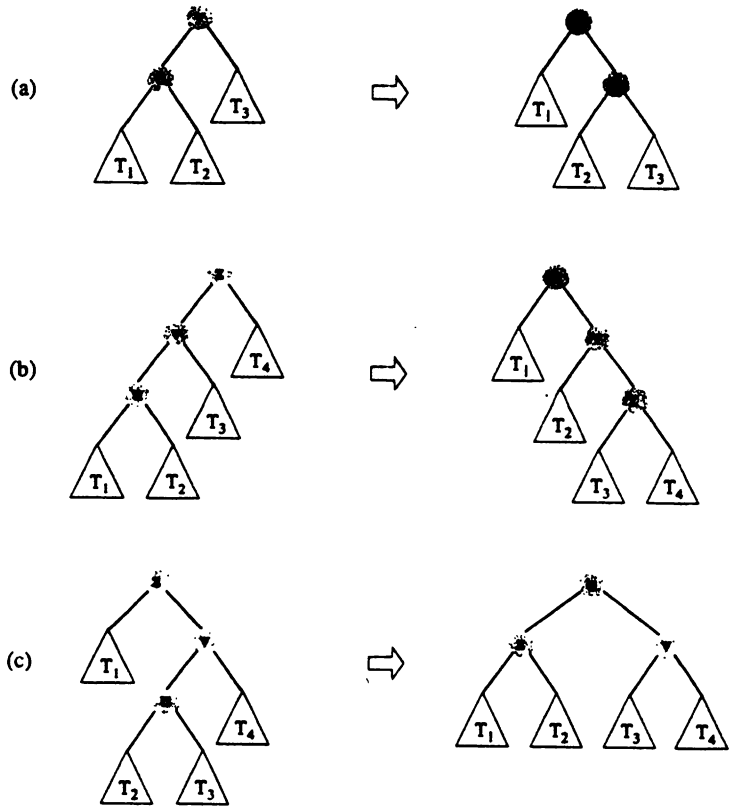


Figura 6.16 Rotazioni eseguite in un passo di splay. Ciascun caso ne ha uno simmetrico che non mostriamo.

È facile verificare che più l'albero T è bilanciato, più basso è il valore del potenziale $\Phi(T)$. Per convincersene, basta provare a calcolare il potenziale degli alberi dati in Figura 6.18.

Lemma 6.3 *Sia u un nodo in un albero, con figli v e w. Allora*

$$r(u) > 1 + \min\{r(v), r(w)\}$$

Dimostrazione. Chiaramente, $d(u) = d(v) + d(w) + 1$, e $d(u) \geq 2\min\{d(v), d(w)\} + 1$. Prendendo il logaritmo di ambo i membri, si ottiene la disuguaglianza nell'enunciato. □

Daremo ora una limitazione superiore alla variazione di potenziale causata da un singolo passo di splay. Questa limitazione sarà poi utile per calcolare il co-

classe `AlberoSplay` **estende** `AlberoBinarioDiRicerca`:
dati: $S(n) = O(n)$
albero binario di ricerca T ereditato, senza alcuna informazione aggiuntiva.

operazioni:
`search`(*chiave k*) → *elem* $T_{am} = O(\log n)$
chiama `search()` ereditata, poi esegue l'euristica `splay` sul nodo u su cui è terminata la ricerca.

`insert`(*elem e, chiave k*) $T_{am} = O(\log n)$
chiama `insert()` ereditata, poi esegue l'euristica `splay` sul nodo u contenente l'elemento inserito.

`delete`(*elem e*) $T_{am} = O(\log n)$
chiama `delete()` ereditata. Sia v il nodo eliminato fisicamente e sia $p(v)$ il padre di v appena prima dell'eliminazione. Esegue l'euristica `splay` sul nodo $p(v)$.

Figura 6.17 Dizionario `AlberoSplay` come estensione della classe `AlberoBinarioDiRicerca` di Figura 6.3.

sto ammortizzato di una singola operazione in base alla formula spiegata nel Paragrafo 2.7 del Capitolo 2.

Lemma 6.4 *Sia x un nodo in uno splay tree. Siano $r(x)$ ed $r'(x)$ i rank del sottoalbero radicato in x prima e dopo un passo di splay ad x, rispettivamente. In modo simile, siano T e T' l'albero prima e dopo il passo di splay. Risulta:*

(1) $r'(x) \geq r(x)$
(2) Se $p(x)$ è radice, allora $\Phi(T') - \Phi(T) < r'(x) - r(x)$.
(3) Se $p(x)$ non è radice, allora $\Phi(T') - \Phi(T) < 3(r'(x) - r(x)) - 1$.

Dimostrazione. Il punto (1) è ovvio, poiché i discendenti di x aumentano quando x sale. Per brevità, sia $\Delta\Phi = \Phi(T') - \Phi(T)$. Relativamente al punto (2), osservando la Figura 6.16b si vede che $r'(x) = r(y)$. Poiché solo x e y cambiano rank in T':

$$\Delta\Phi = (r'(x) - r(x)) + (r'(y) - r(y)) = r'(y) - r(x) < r'(x) - r(x)$$

Per il punto (3), consideriamo solo il caso in Figura 6.16c, essendo gli altri casi analoghi. In questo caso solo x, y e z cambiano rank in T', e quindi risulta

$$\Delta\Phi = (r'(x) - r(x)) + (r'(y) - r(y)) + (r'(z) - r(z))$$

Guardando agli alberi iniziali e finali, abbiamo $r'(y) < r'(x)$ e $r(x) < r(y)$, e quindi, sommando:

$$r'(y) - r(y) < r'(x) - r(x)$$

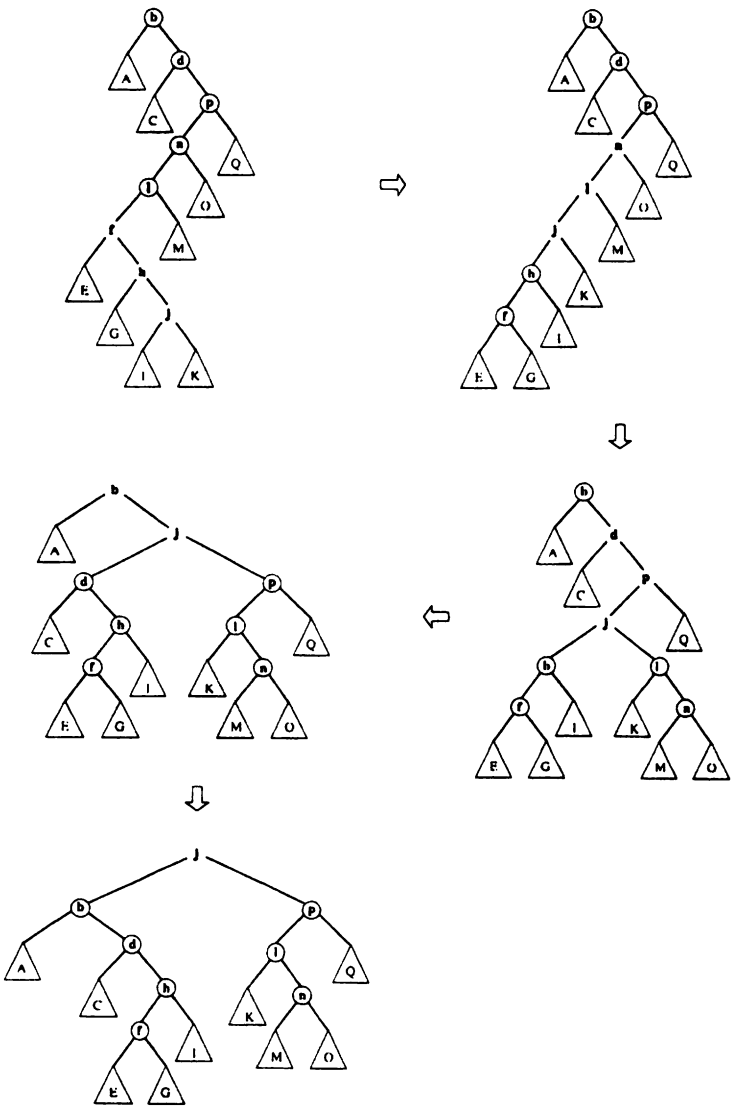


Figura 6.18 Effetto dell'euristica splay sul nodo j.

che implica

$$\Delta\Phi < 2(r'(x) - r(x)) + (r'(z) - r(z)) \tag{6.1}$$

Per completare la dimostrazione, è sufficiente mostrare che

$$r'(z) - r(z) < r'(x) - r(x) - 1$$

Denoteremo con r'' il rank nell'albero intermedio T'', ottenuto dopo la prima rotazione. Grazie al Lemma 6.3: $r''(y) > 1 + \min\{r''(x), r''(z)\}$, e guardando agli alberi iniziale T, intermedio T'' e finale T', risulta $r''(x) = r(x)$, $r''(y) = r'(x) = r(z)$, e $r''(z) = r'(z)$, così che:

$$r'(x) = r(z) > 1 + \min\{r(x), r'(z)\}$$

Quindi, o abbiamo $r'(x) > 1 + r(x)$ oppure $r(z) > 1 + r'(z)$. Nel primo caso, $r'(x) - r(x) > 1$ e poiché $r'(z) < r(z)$, otteniamo

$$r'(z) - r(z) < 0 < r'(x) - r(x) - 1$$

Nel secondo caso, $r'(z) - r(z) < -1$ e poiché $r'(x) - r(x) > 0$, otteniamo ancora

$$r'(z) - r(z) < -1 < r'(x) - r(x) - 1$$

La dimostrazione è facilmente completata usando la disuguaglianza che lega z ed x nell'Equazione (6.1). □

Teorema 6.3 *Il tempo ammortizzato di una operazione di splay in un albero autoaggiustante con n nodi è $O(\log n)$.*

Dimostrazione. Usiamo il Lemma 6.4 per calcolare il costo ammortizzato di una operazione di splay al nodo x, che consiste di una sequenza di passi di splay (rotazioni). Per ciascun passo di splay, siano T e T' gli alberi prima e dopo la rotazione, rispettivamente. Come visto nel Paragrafo 2.7, il costo ammortizzato a di un passo di splay è dato dal tempo di esecuzione effettivo t più la variazione di potenziale:

$$a = t + \Phi(T') - \Phi(T)$$

Assumiamo che il tempo di esecuzione reale t di un passo di splay sia 1 (se non lo è, $t = c$ ed è sufficiente aggiungere il fattore moltiplicativo c nella definizione di potenziale, ottenendo $\Phi(T) = c\sum_{x \in T} r(x)$, per far funzionare la dimostrazione). Consideriamo i due casi dell'enunciato del Lemma 6.4

- se x è figlio della radice, e quindi siamo all'ultimo passo di splay:

$$a = 1 + \Phi(T') - \Phi(T) < 1 + (r'(x) - r(x)) < 1 + \Delta r < 1 + 3\Delta r.$$

- se x non è figlio della radice:

$$a = 1 + \Phi(T') - \Phi(T) < 1 + 3(r'(x) - r(x)) - 1 < 3\Delta r$$

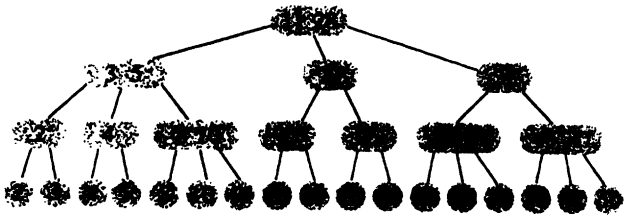


Figura 6.19 Esempio di albero 2-3. I nodi interni con due figli mantengono il massimo del sottoalbero sinistro (S); i nodi interni con tre figli mantengono il massimo del sottoalbero sinistro e di quello centrale (S ed M).

Per l'intera operazione di splay, sia $T = T_0, T_1, T_2, \ldots, T_k$ la serie di alberi prodotta dalla sequenza di k passi di splay, e siano $r = r_0, r_1, r_2, \ldots, r_k$ le corrispondenti funzioni di rank. Il costo ammortizzato totale è dato da

$$\sum_{i=1}^{k} a_i < 1 + \sum_{i=1}^{k} 3(r'_i - r_i)$$

La somma nel lato destro è una serie telescopica, poiché $r'_i = r_{i+1}$, e quindi il costo ammortizzato totale è:

$$\sum_{i=1}^{k} a_i < 1 + 3\,(\text{rank finale di } x - \text{rank iniziale di } x).$$

Poiché il rank finale di x è al più $\log n$, otteniamo il tempo desiderato. □

Possiamo ora riassumere le prestazioni degli alberi splay nel seguente teorema.

Teorema 6.4 *Il tempo totale richiesto da σ operazioni* `search`, `insert` *e* `delete` *in un albero binario di ricerca auto-aggiustante è $O(\sigma \log m)$, dove m è il massimo numero di nodi che l'albero raggiunge durante la sequenza.*

6.4 Alberi 2-3

Nei Paragrafi 6.2 e 6.3 abbiamo visto come usare rotazioni per implementare le operazioni su dizionari in tempo logaritmico (nel caso peggiore o ammortizzato). In questo paragrafo studieremo una tecnica alternativa alle rotazioni, basata sull'idea di permettere una maggiore flessibilità nel grado dei nodi. Se il grado non è vincolato ad essere 2, il bilanciamento può infatti essere mantenuto tramite opportune separazioni (`split`) e fusioni (`fuse`) di nodi.

Definizione 6.6 (Albero 2-3) *Un albero 2-3 è un albero che in cui ogni nodo interno ha 2 o 3 figli e tutti i cammini radice-foglia hanno la stessa lunghezza.*


```
algoritmo search(radice r di un albero 2-3, chiave x) → elem
    if ( r è una foglia ) then
        if ( x = chiave(r) ) then return elem(r)
        else return null
    v1 ← i-esimo figlio di r
    if ( x ≤ S[r] ) then return search(v1, x)
    else if ( r ha due figli oppure x ≤ M[r] ) then return search(v2, x)
        else return search(v3, x)
```

Figura 6.20 Implementazione dell'operazione `search` in un albero 2-3.

Dimostriamo innanzitutto una limitazione sull'altezza degli alberi 2-3.

Lemma 6.5 *Sia T un albero 2-3 con n nodi, f foglie ed altezza h. Le seguenti disuaglianze sono soddisfatte da n, f e h: $2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$ e $2^h \leq f \leq 3^h$.*

Dimostrazione. Mostriamo la limitazioni contemporaneamente usando induzione su h. Se $h = 0$, l'albero consiste di un singolo nodo, che è anche foglia, e le disuguaglianze sono banalmente verificate. Supponiamo ora che l'ipotesi induttiva sia verificata sino ad altezza h e consideriamo un albero 2-3 T di altezza $h+1$. Sia T' ottenuto da T eliminando l'ultimo livello e siano n' e f' il numero di nodi e di foglie di T', rispettivamente. Per ipotesi induttiva, $2^{h+1} - 1 \leq n' \leq (3^{h+1} - 1)/2$ e $2^h \leq f' \leq 3^h$. Poiché ogni foglia di T' ha almeno due ad al più tre figli in T, avremo $2 \cdot 2^h \leq f \leq 3 \cdot 3^h$, ovvero $2^{h+1} \leq f \leq 3^{h+1}$. Poiché $n = n' + f$, è facile completare la dimostrazione sfruttando l'ipotesi induttiva su n'. □

Usando la limitazione sul numero di nodi dimostrata nel Lemma 6.5, e passando al logaritmo, otteniamo che l'altezza di un albero 2-3 è $\Theta(\log n)$.

Le chiavi e gli elementi del dizionario sono assegnati alle foglie dell'albero, in modo che le chiavi appaiano in ordine crescente da sinistra verso destra. Ogni nodo interno v mantiene invece due informazioni supplementari: $S[v]$ e $M[v]$. $S[v]$ è la massima chiave nel sottoalbero radicato nel figlio sinistro di v, e $M[v]$ è la massima chiave nel sottoalbero radicato nel figlio centrale di v. Un esempio è mostrato in Figura 6.19. Grazie alle informazioni S ed M, una ricerca può essere implementata in una maniera simile agli alberi binari di ricerca classici.

`search`(*chiave* k) $\rightarrow$ *elem*

Confrontiamo la chiave cercata k sia con $S[v]$ che con $M[v]$. Se $k \leq S[v]$ proseguiamo nel sottoalbero sinistro; se $S[v] < k \leq M[v]$ proseguiamo nel sottoalbero centrale; altrimenti proseguiamo nel sottoalbero destro. Lo pseudocodice è mostrato in Figura 6.20. Il tempo richiesto da `search` è proporzionale all'altezza dell'albero, che è $\Theta(\log n)$ per il Lemma 6.5.

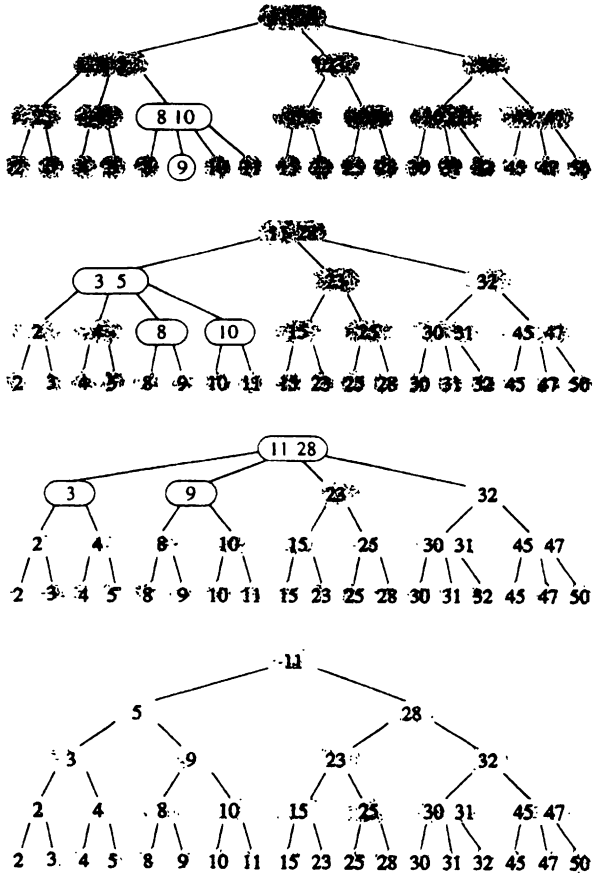


Figura 6.21 Inserimento del nodo con chiave 9 in un albero 2-3.

6.4.1 Fusioni e separazioni di nodi

In questo paragrafo mostreremo come sfruttare le possibili variazioni di grado per aggiornare un albero 2-3 a fronte di inserimenti e cancellazioni di elementi.

`insert`(*elem e*, *chiave k*)

Creiamo un nuovo nodo u con elemento e e chiave k. Localizziamo la corretta posizione per l'inserimento di u ricercando la chiave k nell'albero. Identifichiamo così un nodo v, sul penultimo livello, che dovrebbe diventare genitore di u. Abbiamo ora due casi:

---

algoritmo `split`(*nodo v*)
1. crea un nuovo nodo w
2. sia v_i l'i-esimo figlio di v in T, $1 \leq i \leq 4$
3. rendi v_1 e v_2 figli sinistro e destro di w
4. **if** (*parent*[v] = `null`) **then**
5. crea un nuovo nodo r
6. rendi w e v figli sinistro e destro di r
7. **else**
8. aggiungi w come figlio di *parent*[v] immediatamente precedente a v
9. **if** (*parent*[v] ha quattro figli) **then** `split`(*parent*[v])

Figura 6.22 Implementazione della procedura ausiliaria `split` usata per implementare l'inserimento in un albero 2-3.

- v ha due figli: possiamo aggiungere u come nuovo figlio di v, inserendolo opportunamente come figlio sinistro, centrale, o destro in modo da mantenere l'ordinamento crescente delle chiavi. Questo può comportare dover aggiornare i valori S ed M del nodo v e dei suoi antenati.
- v ha tre figli: non potendo aggiungere un ulteriore figlio a v, separiamo il nodo in due, con un'operazione chiamata `split`. Creiamo un nuovo nodo w e calcoliamo la posizione corretta che u dovrebbe avere rispetto ai tre figli di v. Rendiamo le due foglie con chiavi minime figlie di w e le rimanenti due figlie di v. Attacchiamo poi w come figlio del padre di v immediatamente precedente a v. Se il padre di w aveva due figli, possiamo fermarci. Altrimenti, dobbiamo eseguire un nuovo `split` sul padre di v, procedendo nell'albero verso l'alto. Nel caso peggiore, quando tutti i nodi lungo il cammino avevano già tre figli, aggiungeremo all'albero una nuova radice. I valori S e M dei nodi incontrati lungo la risalita vanno anch'essi aggiornati opportunamente.

Un esempio di inserimento è illustrato in Figura 6.21. Lo pseudocodice della procedura `split`, richiamata su un nodo v con 4 figli, è inoltre mostrato in Figura 6.22 (lo pseudocodice non mostra come aggiornare i campi S ed M di ciascun nodo: aggiungere le opportune istruzioni può essere un utile esercizio). Poiché ciascuno split richiede tempo costante e l'altezza dell'albero è $\Theta(\log n)$, nel caso peggiore l'inserimento richiede tempo $O(\log n)$.

`delete`(*elem e*)

L'operazione di cancellazione è simmetrica a quella di inserimento. Sia v il nodo contenente l'elemento e da eliminare. Abbiamo tre casi:

- v è la radice: basta rimuoverla ottenendo un albero vuoto.
- Il padre di v ha tre figli: è possibile rimuovere v, aggiornando eventualmente i campi S e M di v e dei suoi antenati (vedi Figura 6.23a).

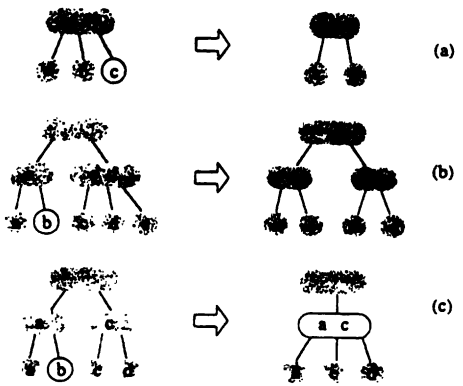


Figura 6.23 Cancellazione da un albero 2-3: vari casi.

- Il padre di v ha due figli: se il padre di v è la radice, basta eliminare v e suo padre, lasciando l'altro figlio come radice. Altrimenti, eseguiamo una operazione simmetrica allo `split`, che chiameremo `fuse`. Sia w il padre di v; assumiamo che w abbia un fratello l alla sua sinistra (nessun nodo può infatti essere figlio unico, e il caso in cui w abbia un unico fratello a destra viene trattato in modo simile). Se l ha tre figli, spostiamo il figlio destro di l come figlio sinistro di w e cancelliamo v (vedi Figura 6.23b). Altrimenti, dopo aver rimosso v, attacchiamo l'unico figlio rimanente di w come figlio destro di l e richiamiamo ricorsivamente la procedura per cancellare w (vedi Figura 6.23c).

Un esempio di cancellazione è mostrato in Figura 6.24. Anche in questo caso, poiché ciascuna `fuse` richiede tempo costante e l'altezza dell'albero è $\Theta(\log n)$, nel caso peggiore la cancellazione richiede tempo $O(\log n)$.

Riassumiamo i dettagli sull'implementazione della classe `Albero23` in Figura 6.25 e le prestazioni degli alberi 2-3 nel seguente teorema.

Teorema 6.5 *Un albero 2-3 con n nodi supporta operazioni* `search`, `insert` *e* `delete` *in tempo $O(\log n)$ nel caso peggiore.*

6.5 B-alberi

In questo paragrafo affronteremo il problema di come realizzare un dizionario in memoria secondaria. Come abbiamo visto nel Paragrafo 2.8 del Capitolo 2, la memoria secondaria è tipicamente molto più grande di quella principale, ma accedervi è molto più costoso in termini di tempo. È quindi importante minimizzare

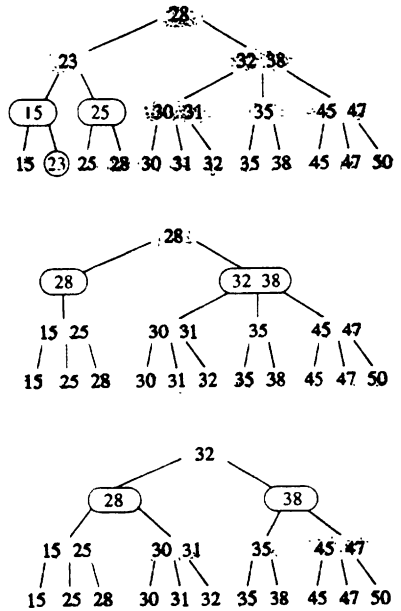


Figura 6.24 Cancellazione del nodo con chiave 23 da un albero 2-3.

il numero di letture e scritture su memoria esterna (I/O). Per ammortizzare il tempo speso in ogni accesso, i dati sono gestiti in blocchi di dimensione B: per poter sfruttare questo fatto, è importante che un algoritmo o una struttura dati esibisca località nell'accesso ai dati.

Le implementazioni dei dizionari proposte nei paragrafi precedenti non esibiscono buona località: intuitivamente, non offrono nessuna garanzia che i nodi su un certo cammino di ricerca risiedano nello stesso blocco, e quindi si potrebbero avere $\Theta(\log n)$ I/O nel caso peggiore. Come vedremo, è possibile fare molto meglio. L'idea è di aumentare l'informazione a carico di ciascun nodo ed il grado del nodo rendendoli proporzionali a B. In questo modo, potremo usare tutta l'informazione contenuta in ogni blocco che porteremo in memoria principale, e, poiché diminuiremo considerevolmente l'altezza dell'albero, avremo un numero minore di accessi a memoria secondaria. I B-alberi che presentiamo in questo paragrafo sono dovuti a Bayer e McCreight [4] e sono un'estensione naturale degli alberi 2-3 che abbiamo descritto nel Paragrafo 6.4.

6.5.1 Definizioni e proprietà

Sia t un intero fissato ≥ 2, detto *grado minimo* e tipicamente scelto in modo proporzionale alla dimensione B di un blocco.

classe `Albero23` **implementa** `Dizionario`:
dati: $S(n) = O(n)$
un albero 2-3 T con n nodi: le foglie mantengono le chiavi e gli elementi del dizionario, mentre i nodi interni mantengono le informazioni supplementari S e M.

operazioni:
`search`($chiave\ k$) $\rightarrow elem$ $T(n) = O(\log n)$
traccia un cammino nell'albero usando la proprietà di ricerca e le informazioni S e M per decidere se proseguire nel sottoalbero sinistro, centrale (se esiste) o destro.

`insert`($elem\ e, chiave\ k$) $T(n) = O(\log n)$
crea un nuovo nodo v con elemento e e chiave k, e lo aggiunge all'albero come foglia mantenendo la proprietà di ricerca. Se il padre di v aveva già tre figli, separa il nodo in due tramite uno `split` e propaga le separazioni verso l'alto, fino al primo nodo con due figli o fino alla creazione di una nuova radice.

`delete`($elem\ e$) $T(n) = O(\log n)$
elimina il nodo v con elemento e. Se il padre w di v aveva solo due figli, unisciilo al fratello tramite un'operazione `fuse` e propaga le fusioni verso l'alto, fino al primo nodo con tre figli o fino alla cancellazione della radice.

Figura 6.25 Dizionario realizzato mediante alberi 2-3.

Definizione 6.7 (B-albero) *Un B-albero non vuoto di grado minimo t è un albero radicato con le seguenti proprietà:*

1. *tutte le foglie hanno la stessa profondità;*
2. *ogni nodo v diverso dalla radice mantiene $k(v)$ chiavi ordinate, $chiave_1(v) \leq chiave_2(v) \leq \ldots \leq chiave_{k(v)}(v)$, tali che $t - 1 \leq k(v) \leq 2t - 1$;*
3. *la radice mantiene almeno 1 ed al più $2t - 1$ chiavi ordinate;*
4. *ogni nodo interno v ha $k(v) + 1$ figli;*
5. *le chiavi $chiave_i(v)$ separano gli intervalli di chiavi memorizzati in ciascun sottoalbero: se c_i è una qualunque chiave nell'i-esimo sottoalbero di un nodo v, allora $c_1 \leq chiave_1(v) \leq c_2 \leq chiave_2(v) \leq \ldots \leq c_{k(v)} \leq chiave_{k(v)}(v) \leq c_{k(v)+1}$.*

Diremo che un nodo è *pieno* se contiene esattamente $(2t - 1)$ chiavi e *quasi vuoto* se ne contiene $(t - 1)$. Dalle Proprietà 2 e 4 segue che il grado di ogni nodo interno diverso dalla radice è compreso tra t e $2t$ (il grado minimo della radice può invece scendere fino a 2 in base alla Proprietà 3). Esaminiamo ora l'altezza di un B-albero.

Lemma 6.6 *Ogni B-albero di altezza h con n chiavi soddisfa $h \leq \log_t \frac{n+1}{2}$.*

Dimostrazione. L'altezza massima, a parità di numero dei nodi, si raggiunge quando ogni nodo ha grado minimo. Assumiamo quindi che la radice abbia 2 figli e che ogni altro nodo interno ne abbia t. Avremo un nodo a profondità 0 (la radice), due nodi a profondità 1 (i figli della radice), $2t$ nodi a profondità 2, $2t^2$ nodi a profondità 3, e così via. In generale, il numero dei nodi al livello i è t volte il numero dei nodi al livello $i - 1$, ed è pari a $2t^{i-1}$. Poiché le chiavi sono n ed ogni nodo ne contiene $t - 1$, la seguente disuguaglianza deve essere soddisfatta:

$$1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \leq n$$

Usando la serie geometrica, sappiamo che

$$\sum_{i=1}^{h} t^{i-1} = \sum_{i=0}^{h-1} t^i = \frac{t^h - 1}{t - 1}$$

da cui si ottiene:

$$1 + 2(t-1)\frac{t^h - 1}{t - 1} \leq n$$

e quindi

$$t^h \leq \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

Passando al logaritmo in base t, otteniamo la limitazione nell'enunciato. □

`search`($chiave\ k$) $\rightarrow elem$

Possiamo implementare l'operazione `search` su un B-albero basandoci sul fatto che la Proprietà 5 della Definizione 6.7 è una generalizzazione della proprietà di ricerca introdotta nel Paragrafo 6.1: invece di prendere una decisione binaria, prenderemo ad ogni passo una decisione $\Theta(t)$-aria, poichè ogni nodo ha $\Theta(t)$ figli. Al generico passo, trovandoci su un nodo v, identifichiamo la più piccola chiave di v maggiore dell'elemento cercato x: se non esiste, proseguiamo sull'ultimo figlio di v, altrimenti proseguiamo sul figlio immediatamente precedente la chiave individuata. Lo pseudocodice è mostrato in Figura 6.26.

Nel caso peggiore, poiché ad ogni chiamata si scende di un livello, il numero di chiamate ricorsive è pari all'altezza dell'albero, e quindi $O(\log_t n)$. Se ogni nodo attraversato entrasse interamente in un blocco, il numero di I/O sarebbe quindi $\approx \log_t n$, che può essere sostanzialmente inferiore rispetto a $\log_2 n$. Il vantaggio maggiore si ottiene proprio scegliendo t proporzionale a B, nel qual caso avremo $\approx \log_B n$ I/O. Consideriamo ora il numero di operazioni. Osserviamo che su ogni nodo, nell'implementazione data, spediamo tempo $O(t)$. Essendo le chiavi ordinate, però, potremmo effettuare una ricerca binaria all'interno del nodo, il che richiederebbe tempo $O(\log t)$. Con questo accorgimento, eseguiremmo $O(\log t \cdot \log_t n)$ operazioni elementari. Usando le regole del cambiamento di base dei logaritmi, è facile vedere che questa quantità è $O(\log n)$, esattamente come negli alberi AVL o 2-3.

algoritmo `search`(*radice v di un B-albero, chiave x*) → *elem*
1. $i \leftarrow 1$
2. **while** ($i \leq k(v)$ e $x > chiave_i(v)$) **do** $i \leftarrow i + 1$
3. **if** ($i \leq k(v)$ e $chiave_i(v) = x$) **then return** $elem_i(v)$
4. **if** (v è foglia) **then return** `null`
5. **else return** `search`(*i-esimo figlio di v, x*)

Figura 6.26 Implementazione dell'operazione `search` in un B-albero.

6.5.2 Inserimenti e cancellazioni di chiavi

In questo paragrafo mostreremo come inserire nuovi elementi e come cancellare elementi esistenti ripristinando le proprietà dei B-alberi nella Definizione 6.7.

`insert`(*elem e, chiave k*)

Per inserire chiavi faremo uso di una operazione `split` analoga a quella vista nel Paragrafo 6.4.1 per gli alberi 2-3. In particolare, una split viene eseguita ogni qualvolta si richede di aggiungere una nuova chiave ad un nodo pieno (ovvero già contenente $2t - 1$ chiavi). Il nodo verrebbe infatti a contenere $2t$ chiavi, violando la Proprietà 2. È però possibile dividere il nodo in due soddisfacendo localmente i vincoli ed eventualmente propagando il problema verso l'alto. Lo `split` è mostrato in Figura 6.27: il nodo sinistro contiene le $t - 1$ chiavi minime e quello destro le t chiavi massime, mentre la chiave t viene spinta verso l'alto per separare i due nodi.

Data l'operazione `split`, l'inserimento può essere effettuato come segue. Dopo aver identificato, tramite una ricerca cbe termina con un insuccesso, la foglia f in cui la chiave k e l'elemento e andrebbero aggiunti, si possono verificare due casi:

- se la foglia non è piena, si inserisce la chiave nell'opportuna posizione e l'algoritmo termina;
- se la foglia è piena si esegue uno split promuovendo la t-esima chiave, diciamo x', al livello superiore. La chiave x' viene aggiunta al padre della foglia f nella posizione opportuna, con i puntatori a sinistra e a destra che puntano ai due nuovi nodi creati. Potrebbe accadere che anche il padre di f sia pieno, e in tal caso si eseguirà uno split facendo risalire una chiave x'' al livello superiore. Nel

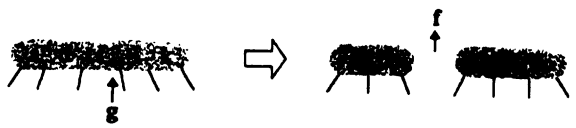


Figura 6.27 Split in un B-albero con grado minimo $t = 3$.

classe `BAlbero` **implementa** `Dizionario`:
dati: $S(n) = O(n)$
un intero $t \geq 2$ e un B-albero T di grado minimo t con n nodi.

operazioni:

`search`(*chiave k*) → *elem* $\approx \log_t n$ I/O, $T(n) = O(\log n)$
traccia un cammino nell'albero prendendo una decisione $\Theta(t)$-aria su ogni nodo, in base alla Proprietà 5 nella definizione di B-albero.

`insert`(*elem e, chiave k*) $\approx \log_t n$ I/O, $T(n) = O(\log n)$
identifica tramite una ricerca la foglia f in cui l'elemento va inserito, in modo da mantenere la proprietà di ricerca. Se la foglia è piena, esegui uno `split` promuovendo la t-esima chiave verso l'alto. Se tutti i nodi nel cammino da f alla radice sono pieni, crea una nuova radice.

`delete`(*elem e*) $\approx \log_t n$ I/O, $T(n) = O(\log n)$
sia v il nodo contenente l'elemento da eliminare. Se v non è una foglia, elimina il predecessore, dopo aver copiato chiave ed elemento di quest'ultimo in v. Altrimenti elimina v: se il padre w di v era quasi vuoto, esegui un'operazione `fuse` e propaga le fusioni verso l'alto, fino al primo nodo con più di t figli o fino alla cancellazione della radice.

Figura 6.28 Dizionario realizzato mediante B-alberi di grado minimo t.

caso peggiore, quando tutti i nodi lungo il cammino erano già pieni, dovremo aggiungere all'albero una nuova radice contenente una sola chiave.

Con argomentazioni simili a quelle usate per l'operazione `search` si può dimostrare che il numero di I/O ed il tempo di esecuzione sono rispettivamente $O(\log_t n)$ e $O(\log n)$.

`delete`(*elem e*)

Per cancellare chiavi faremo uso di una operazione `fuse` analoga a quella vista nel Paragrafo 6.4.1 per gli alberi 2-3. Sia u il nodo contenente l'elemento e da cancellare. Se u non è una foglia, troviamo il suo predecessore y, rimpiazziamo $chiave(u)$ ed $elem(u)$ con $chiave(y)$ ed $elem(y)$, rispettivamente, e ricorsivamente cancelliamo y, che è necessariamente contenuto in una foglia. Vediamo quindi come cancellare un elemento da una foglia:

- se la foglia non è quasi vuota (ovvero contiene almeno t chiavi), è sufficiente cancellare l'elemento;
- se la foglia è quasi vuota, esaminiamo la situazione del fratello sinistro e destro:
  - se uno dei due fratelli non è quasi vuoto, operiamo una redistribuzione degli elementi tra fratelli in modo da soddisfare i vincoli di pienezza, modificando anche la chiave separatrice contenuta nel padre;

– se entrambi i fratelli sono quasi vuoti, operiamo una operazione di `fuse`: aggiungiamo al fratello, mantenendo l'ordine, sia le chiavi della foglia sia la chiave separatrice. Eliminiamo poi la foglia. In tal modo il numero di figli del padre, così come le sue chiavi, diminuisce di 1. Se il padre era quasi vuoto, e quindi ora ha $t - 2$ chiavi, dobbiamo propagare le redistribuzioni di chiavi o fusioni verso l'alto. Nel caso peggiore, quando tutti i nodi lungo il cammino ed i loro fratelli erano quasi vuoti pieni, la radice dell'albero sarà eliminata e l'altezza diminuirà di 1.

Anche in questo caso si può dimostrare che il numero di I/O ed il tempo di esecuzione sono rispettivamente $O(\log_t n)$ ed $O(\log n)$.

Riassumiamo l'implementazione della classe `BAlbero` in Figura 6.28 e le prestazioni dei B-alberi nel seguente teorema:

Teorema 6.6 *Un B-albero con grado minimo t ed n nodi è in grado di supportare le operazioni* `search`, `insert` *e* `delete` *in tempo* $O(\log_2 n)$ *e con* $O(\log_t n)$ *I/O.*

6.6 Alberi 2-3-4 ed alberi red-black

Nei precedenti paragrafi abbiamo descritto due diverse tipologie di alberi di ricerca. Gli alberi AVL e gli alberi auto-aggiustanti sono alberi di ricerca binari, e mantengono il bilanciamento durante operazioni `insert` e `delete` tramite opportune rotazioni. Gli alberi 2-3 ed i B-alberi sono invece caratterizzati dall'avere nodi con grado variabile, e proprio la variabilità del grado viene sfruttata per dividere o fondere nodi mantenendo il bilanciamento. In questo paragrafo mostreremo che esiste una stretta relazione tra i due approcci, presentando una prospettiva unificata al problema di mantenere il bilanciamento dell'albero. A tal fine, descriveremo due nuovi tipi di alberi di ricerca, gli *alberi* 2-3-4 e gli *alberi red-black*, mostrando un modo per trasformare gli uni negli altri.

Alberi 2-3-4. Un albero 2-3-4 può essere facilmente definito a partire da un B-albero.

Definizione 6.8 (Albero 2-3-4) *Un albero* 2-3-4 *è un B-albero avente grado minimo* 2.

In base alle Definizioni 6.7 e 6.8, in un albero 2-3-4 il grado di ogni nodo diverso dalla radice può quindi variare da 2 a 4, ed il nodo può contenere da 1 a 3 chiavi. Le operazioni `search`, `insert` e `delete` sono implementate esattamente come nei B-alberi, usando come sottoprocedure le operazioni `split` e `fuse`. Rimandiamo al Paragrafo 6.5 per maggiori dettagli.

Alberi red-black. Gli alberi red-black sono invece alberi binari di ricerca che si basano su una proprietà di colorazione dei nodi per mantenere il bilanciamento. In particolare, ad ogni nodo è assegnato uno tra due colori, *rosso* o *nero*, in modo da soddisfare le proprietà specificate nella seguente definizione.

Definizione 6.9 (Albero red-black) *Un albero red-black è un albero binario di ricerca che soddisfa le seguenti proprietà:*

- *ogni nodo ha colore rosso o nero;*
- *ogni foglia è nera e contiene elemento* `null`*;*
- *se un nodo è rosso, entrambi i suoi figli sono neri;*
- *ogni cammino da un nodo ad una foglia nel suo sottoalbero contiene lo stesso numero di nodi neri.*

Chiameremo *altezza nera* di un nodo v, e la indicheremo con $h_n(v)$, il numero di nodi neri in un qualunque cammino da v ad una foglia nel suo sottoalbero, senza contare v stesso. Il fatto che ogni cammino radice - foglia abbia la stessa altezza nera può essere usato per provare che un albero red-black ha altezza logaritmica nel numero di nodi.

Lemma 6.7 *Ogni nodo v in un albero red-black contiene nel suo sottoalbero un numero di nodi interni maggiore o uguale a* $2^{h_n(v)} - 1$.

Dimostrazione. La dimostrazione procede per induzione sull'altezza del sottoalbero T_v radicato in v. Il passo base si ha quando v è una foglia, ed è banalmente verificato essendo $h_n(v) = 0$. Assumiamo ora cbe T_v abbia altezza h e cbe valga l'ipotesi induttiva su alberi di altezza $\leq h - 1$. I figli del nodo v hanno altezza pari ad $h - 1$, ed altezza nera pari ad $h_n(v)$ oppure $h_n(v) - 1$. Per ipotesi induttiva, i sottoalberi radicati in ciascuno dei due figli contengono quindi almeno $2^{h_n(v)-1} - 1$, da cui il numero di nodi nel sottoalbero radicato in v è

$$\geq 2^{h_n(v)-1} - 1 + 2^{h_n(v)-1} - 1 + 1 = 2^{h_n(v)} - 1$$

come volevamo dimostrare. □

Corollario 6.2 *Un albero red-black con n nodi ha altezza* $\leq 2\log(n+1)$.

Dimostrazione. Sia h l'altezza dell'albero e sia r la sua radice. Poiché ogni nodo rosso ha entrambi i figli neri, si ha $h_n(r) \geq h/2$. Per il Lemma 6.7, risulta quindi $n \geq 2^{h_n(r)} - 1 \geq 2^{h/2} - 1$, da cui segue l'enunciato con semplici manipolazioni algebriche. □

Il Corollario 6.2 implica che l'operazione `search` su un albero red-black ha costo $O(\log n)$. Per mantenere le proprietà sull'altezza nera a fronte di inserimenti e cancellazioni di nodi, gli algoritmi `insert` e `delete` eseguono ricolorazioni di nodi ed opportune rotazioni. Questi algoritmi presentano una casistica molto articolata e tecnica, descritta dettagliatamente in [6]. Nel resto di questo paragrafo mostreremo un approccio alternativo più semplice, basato su una corrispondenza tra alberi red-black ed alberi 2-3-4.

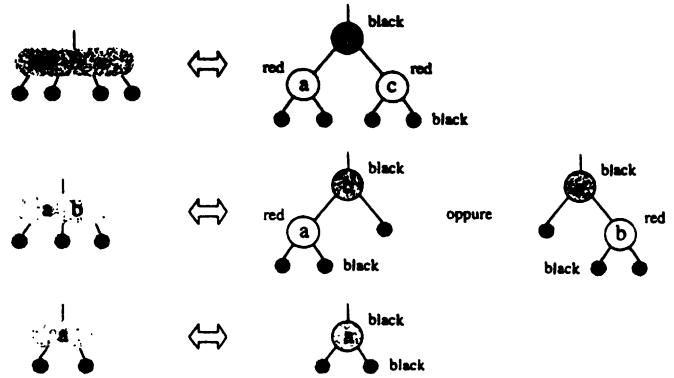


Figura 6.29 Regole di trasformazione tra alberi red-black ed alberi 2-3-4.

Lemma 6.8 *Un albero red-black può essere trasformato in un albero* 2-3-4 *e viceversa.*

Dimostrazione. Le regole di trasformazione sono mostrate in Figura 6.29. Ad esempio, un sottoalbero costituito da un nodo nero e due figli rossi corrisponde in un albero 2-3-4 ad un nodo con tre chiavi e quattro figli. Le altre regole sono analoghe. Una qualunque applicazione di queste regole permette di trasformare un albero di un tipo in un corrispondente albero dell'altro tipo.

Nel caso in cui si stia trasformando un albero 2-3-4 in un red-black occorre accertarsi che le proprietà nella Definizione 6.9 siano verificate. Poiché le radici dei sottoalberi usati in ciascuna regola sono nere, le regole garantiscono che nodi rossi abbiano sempre figli neri. La proprietà sull'altezza nera è assicurata dal fatto che tutte le foglie nell'albero 2-3-4 hanno la stessa profondità, ed ogni regola introduce esattamente un nodo nero. Analoghe considerazioni permettono di dimostrare che le proprietà degli alberi 2-3-4 sono soddisfatte nel caso in cui si stia trasformando un albero red-black in un albero 2-3-4. □

Grazie al Lemma 6.8, le operazioni `insert` e `delete` negli alberi red-black possono essere derivate direttamente da inserimenti e cancellazioni in alberi 2-3-4, che abbiamo illustrato nel Paragrafo 6.5 nel caso generale. Possiamo quindi riassumere le prestazioni degli alberi red-black come segue:

Teorema 6.7 *Alberi red-black con n nodi sono in grado di supportare le operazioni* `search`, `insert` *e* `delete` *in tempo $O(\log n)$ nel caso peggiore.*

Osserviamo che usando il Lemma 6.8 ciascuna operazione di modifica del dizionario potrebbe comportare nel caso peggiore $O(\log n)$ `split` e `fuse`. Con tecniche più sofisticate [6] è possibile dimostrare che inserimenti e cancellazioni negli alberi red-black possono essere implementati in tempo $O(\log n)$ eseguendo solo un numero costante di rotazioni.

6.7 Problemi

Problema 6.1 Definiamo una operazione `concatenate` il cui input è costituito da due insiemi S_1 ed S_2 tali che le chiavi in S_1 sono tutte minori o uguali delle chiavi in S_2 ed il cui output è la fusione dei due insiemi in uno. Progettare un algoritmo per concatenare due alberi binari di ricerca in un albero binario di ricerca. L'algoritmo deve avere tempo di esecuzione $O(h)$ nel caso peggiore, dove h è la massima altezza dei due alberi.

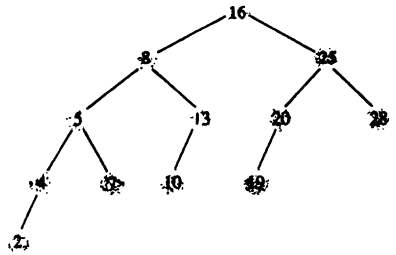
Problema 6.2 (*) Consideriamo l'operazione `concatenate` descritta nel Problema 6.1. Progettare un algoritmo per concatenare due alberi AVL in un unico albero AVL. L'algoritmo deve avere tempo di esecuzione $O(h)$ nel caso peggiore, dove h è la massima altezza dei due alberi.

Problema 6.3 (*) Considerare alberi AVL con chiavi reali ed estenderli in modo da poter supportare due nuove operazioni:

- `aggiungi`(k, x): aggiungi il valore reale x al nodo con chiave k;
- `sommaParziale`(k): restituisci la somma di tutte le chiavi minori di k.

Per semplicità, si può assumere che il nodo con chiave k, se esiste, è unico. Entrambe le operazioni devono essere supportate in tempo $O(\log n)$, dove n è il numero di nodi dell'albero.

Problema 6.4 Si consideri il seguente albero AVL:



a) Si mostrino le modifiche apportate all'albero (e le eventuali rotazioni) in seguito all'inserimento delle chiavi 1, 3, 12, e 15.

b) Si caratterizzi l'insieme D dei nodi la cui rimozione non fa sbilanciare l'albero.

Problema 6.5 (*) Si dimostri che gli unici nodi che possono essere cancellati da un albero di Fibonacci senza eseguire rotazioni sono i tre nodi più in basso del cammino più a sinistra dell'albero.

Problema 6.6 Assumiamo che tutti i nodi di un albero AVL T abbiano un campo *size* che contiene il numero di nodi nel sottoalbero sinistro $+1$. Mostrare come modificare inserimenti e cancellazioni in modo da mantenere correttamente i

valori *size*. Scrivere poi l'algoritmo `trova`(T, k) che localizza la k-esima chiave nell'albero T. L'algoritmo deve avere tempo di esecuzione $O(\log n)$ nel caso peggiore, dove n è il numero nodi di T.

Problema 6.7 Dimostrare che l'altezza di uno splay tree può aumentare in seguito ad uno splay. Dare poi una sequenza di n inserimenti che genera uno splay tree di altezza n. Che costo ha l'intera sequenza?

Problema 6.8 (**) Usare il metodo dei crediti per dimostrare che il costo ammortizzato di una operazione di splay è $O(\log n)$. Suggerimento: assegnare $\lfloor r(x) \rfloor$ monete a ciascun nodo, dove $r(x)$ denota il rank del nodo x come definito nel Paragrafo 6.3.2. Mostrare poi che, quando c'è una rotazione, i nodi il cui rank diminuisce hanno moneta eccedente per pagare per l'operazione.

Problema 6.9 Progettare algoritmi efficienti per trovare il predecessore e successore di un elemento in un albero 2-3 ed in un B-albero.

Problema 6.10 Mostrare che l'altezza di un B-albero è $\Theta(\log_t n)$. In particolare, qual è l'altezza minima di un B-albero? Esprimere una disuguaglianza simmetrica a quella data nel Lemma 6.6 in funzione di n, t ed h.

Problema 6.11 Un *B*-albero* è un B-albero in cui le chiavi sono tutte contenute nelle foglie: i nodi interni contengono le chiavi estreme dei diversi sottoalberi, in modo da poter guidare la ricerca dalla radice alla foglia contenente uno specifico elemento (in modo simile agli alberi 2-3 descritti nel Paragrafo 6.4). Implementare le operazioni `search`, `insert` e `delete` in un B*-albero.

6.8 Sommario

Il problema del dizionario richiede di realizzare una struttura dati che supporti operazioni di ricerca (`search`), inserimenti di nuovi elementi (`insert`), e cancellazioni di elementi già presenti (`delete`). A partire dalla metà degli anni '50, sono state proposte molte strutture dati sofisticate per realizzare dizionari. Tali strutture possono essere partizionate in due gruppi: *strutture basate su confronti*, esemplificate dagli alberi di ricerca che abbiamo descritto in questo capitolo, e *strutture basate sulla rappresentazione*, esemplificate dalle tavole hash che descriveremo nel Capitolo 7.

Come abbiamo visto nel Paragrafo 6.1, un'operazione di accesso ad un albero di ricerca richiede tempo proporzionale all'altezza dell'albero. Lo scopo è quindi mantenere l'altezza bassa. Nel Paragrafo 6.2 abbiamo introdotto la nozione di bilanciamento in altezza su cui si basano gli alberi AVL, abbiamo dimostrato cbe alberi bilanciati in altezza hanno altezza $O(\log n)$, ed abbiamo discusso come mantenere il bilanciamento di alberi binari di ricerca durante operazioni di inserimento e cancellazione tramite semplici rotazioni di sottoalberi. Un diverso approccio al mantenimento del bilanciamento si basa sul permettere una maggiore

flessibilità nel grado dei nodi: se il grado non è vincolato ad essere 2, possiamo infatti eseguire opportune separazioni (`split`) e fusioni (`fuse`) di nodi che mantengono la struttura bilanciata. È questo il caso degli alberi 2-3, dei B-alberi e degli alberi 2-3-4, che abbiamo presentato nei Paragrafi 6.4, 6.5 e 6.6. Nel Paragrafo 6.6 abbiamo anche mostrato che in realtà i due approcci sono equivalenti: in particolare, abbiamo brevemente descritto gli alberi red-black (che nell'implementazione classica matengono il bilanciamento tramite rotazioni) ed abbiamo presentato una corrispondenza uno ad uno con gli alberi 2-3-4, tramite semplici regole di trasformazione degli uni negli altri.

Tutti questi alberi mantengono esplicitamente una condizione di bilanciamento, basata sul fattore di bilanciamento, sul grado o sul colore dei nodi, e tutte le operazioni hanno costo $O(\log n)$ nel caso peggiore. Come abbiamo visto nel Paragrafo 6.3, è anche possibile non mantenere alcuna condizione di bilanciamento esplicita: in tal caso si possono ancora ottenere tempi di esecuzione logaritmici per le varie operazioni, ma ammortizzati su una intera sequenza.

6.9 Note bibliografiche

Il bilanciamento è un parametro cruciale nelle prestazioni degli alberi di ricerca, ed esistono in letteratura due tipi principali di alberi di ricerca bilanciati: alberi bilanciati nel peso o in altezza. Tra gli alberi bilanciati nel peso ricordiamo, ad esempio, gli alberi BB[α] descritti in [8]. Tra gli alberi bilanciati in altezza, ritroviamo invece tutti gli alberi descritti in questo capitolo (ad eccezione degli alberi splay). Gli alberi AVL, dovuti ad Adel'son-Vel'skiĭ e Landis [1], rappresentano una delle prime implementazioni efficienti delle operazioni su dizionari. Gli alberi 2-3 sono attribuiti a John Hopcroft, che li introdusse in un articolo scientifico mai pubblicato, e sono descritti in [2]. I B-alberi furono introdotti da Bayer e McCreight e, grazie alle ottime prestazioni su memoria secondaria, hanno trovato innumerevoli applicazioni [5]. Gli alberi red-black furono inventati da Bayer con il nome di B-alberi binari simmetrici [3]: Guibas e Sedgewick introdussero la convenzione basata sui colori e ne analizzarono molte proprietà [6]. Gli eleganti alberi splay, infine, sono stati introdotti da Sleator e Tarjan in [10]. Descrizioni esaustive di molti alberi di ricerca sono presentate in [7] e [9].

Riferimenti bibliografici

[1] G. M. Adel'son-Vel'skiĭ e Y. M. Landis, "An algorithm for the organization of information", *Dokl. Akad. Nauk. Sssr*, 146:263–266, 1962.

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *Data structures and algorithms*, Addison-Wesley, Reading, 1983.

[3] R. Bayer, "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", *Acta Informatica*, 1:290–306, 1972.

[4] R. Bayer ed E. McCreight, "Organization and maintenance of large ordered indexes", *Acta Informatica*, 1:173–189, 1972.

[5] D. Comer, "The Ubiquitous B-tree", *ACM Computing Surveys*, 11(2):121–137, 1979.
[6] L. J. Guibas e R. Sedgewick, "A dichromatic framework for balanced trees", in *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, 8–21, 1978.
[7] D. E. Knuth, *Sorting and searching*, volume 3 di *The Art of Computer Programming*, Addison-Wesley, 1973.
[8] I. Nievergelt ed E. M. Reingold, "Binary search trees of bounded balance", *SIAM Journal on Computing*, 2:33–43, 1973.
[9] R. Sedgewick, *Algorithms*, Addison-Wesley, seconda edizione, 1988.
[10] D. D. Sleator e R. E. Tarjan, "Self-Adjusting Binary Search Trees", *Journal of the ACM*, 32(3):652–686, 1985.

7

Tavole hash

Hash yourselves to your seats.
(Samir Khuller)

Nel Capitolo 3 abbiamo abbiamo mostrato realizzazioni di un dizionario in cui sempre almeno un'operazione ha costo lineare nel caso peggiore. Nel Capitolo 6, abbiamo visto come supportare le operazioni su un dizionario con n elementi in tempo $O(\log n)$. La tecnica che considereremo in questo capitolo, che sfrutta la proprietà di accesso indicizzato alle celle di un array, permetterà di supportare le operazioni su un dizionario in tempo medio $O(1)$.

7.1 Tavole ad accesso diretto

Supponiamo di voler realizzare un'implementazione efficiente del tipo di dato `Dizionario` sapendo che le chiavi associate agli n elementi che si vogliono memorizzare sono interi appartenenti all'intervallo $[0, m-1]$, con $n \leq m$. Assumiamo inoltre che agli elementi nel dizionario siano associate chiavi distinte. Una tecnica semplicissima consiste nel mantenere un array v di dimensione m tale che, se c'è un elemento $elem$ con chiave k nel dizionario, allora $v[k] = elem$. Se una chiave k non è associata a nessun elemento nel dizionario, poniamo $v[k]$ =`null` per segnalare che la cella è vuota. In altre parole, usiamo le chiavi degli elementi come indici nell'array che contiene gli elementi stessi: un array di questo tipo viene chiamato *tavola ad accesso diretto*. In questo capitolo assumeremo che gli indici di un array di dimensione m siano compresi tra 0 e $m-1$.

Una possibile realizzazione di un dizionario rappresentato come tavola ad accesso diretto è mostrata in Figura 7.1. Si noti che il tempo richiesto da ogni operazione è costante: questa è probabilmente la più efficiente realizzazione possibile per un dizionario. Tuttavia, si noti che se $m >> n$ questo metodo può implicare un enorme spreco di memoria, come nell'esempio seguente.

Esempio 7.1 Supponiamo di dover mantenere un dizionario con i nomi degli studenti che frequentano il corso di *Algoritmi e Strutture Dati*, in modo da poterli facilmente cercare per numero di matricola. Assumendo che ciascun nome abbia

classe `TavolaAccessoDiretto` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione $m \geq n$ in cui $v[k] = elem$ se c'è un elemento $elem$ con chiave k nel dizionario, e $v[k] =$ `null` altrimenti. Le chiavi k devono essere interi distinti nell'intervallo $[0, m-1]$.

operazioni:
`insert`($elem\ e, chiave\ k$) $T(n) = O(1)$
$v[k] \leftarrow e$

`delete`($chiave\ k$) $T(n) = O(1)$
$v[k] \leftarrow$ `null`

`search`($chiave\ k$) $\rightarrow elem$ $T(n) = O(1)$
return $v[k]$

Figura 7.1 Dizionario realizzato mediante tavola ad accesso diretto.

come chiave un numero di matricola a sei cifre, per mantenere nel dizionario anche solo 100 studenti dovremmo comunque allocare un array con circa 1 milione di celle! □

Per valutare il grado di riempimento di una tavola, utilizzeremo la nozione di fattore di carico.

Definizione 7.1 (Fattore di carico) *Definiamo* fattore di carico *di una tavola il rapporto* $\alpha = n/m$ *tra il numero* n *di elementi in essa memorizzati e la sua dimensione* m.

La tavola con i nomi degli studenti dell'esempio precedente avrebbe un fattore di carico $\alpha = 100/1000000$, pari allo $0,01\%$.

7.2 Tavole hash

Sia U l'universo delle chiavi associabili agli elementi del nostro dizionario. Nel Paragrafo 7.1 abbiamo visto una soluzione efficiente nel caso in cui l'universo $U = \{0, \ldots, m-1\}$. Possiamo ancora utilizzare un approccio simile se U non contiene gli interi in $[0, m-1]$, o i numeri non sono interi, o addirittura non sono numeri?

La tecnica che studieremo nel resto di questo capitolo, chiamata *hashing*, è un'estensione della tecnica delle tavole ad accesso diretto. L'idea è quella di non usare direttamente le chiavi come indici nella tavola, ma piuttosto di trasformare le chiavi in indici tramite una opportuna *funzione hash*.

classe `TavolaHashPerfetta` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione $m \geq n$ in cui $v[h(k)] = e$ se c'è un elemento e con chiave $k \in U$ nel dizionario, e $v[h(k)] =$ `null` altrimenti. La funzione $h : U \rightarrow \{0, \ldots, m-1\}$ è una funzione hash perfetta calcolabile in tempo $O(1)$.

operazioni:
`insert`($elem\ e, chiave\ k$) $T(n) = O(1)$
$v[h(k)] \leftarrow e$

`delete`($chiave\ k$) $T(n) = O(1)$
$v[h(k)] \leftarrow$ `null`

`search`($chiave\ k$) $\rightarrow elem$ $T(n) = O(1)$
return $v[h(k)]$

Figura 7.2 Dizionario realizzato mediante tavola hash con funzione hash perfetta.

Definizione 7.2 (Funzione hash) *Una* funzione hash[1] *è una funzione* $h : U \rightarrow \{0, \ldots, m-1\}$ *che trasforma chiavi in indici di una tavola.*

Chiameremo *tavola hash* una tavola indicizzata tramite funzioni hash.

Definizione 7.3 (Funzione hash perfetta) *Una funzione hash* h *è* perfetta *se è iniettiva, cioè per ogni* $u, v \in U$, $[u \neq v] \Rightarrow [h(u) \neq h(v)]$.

Questo implica che, affinché una funzione hash sia perfetta, deve essere $|U| \leq m$, cioè ci deve essere spazio per tanti elementi quante sono le chiavi possibili. Una possibile realizzazione del tipo di dato `Dizionario` basata su una tavola hash con funzione hash perfetta è mostrata in Figura 7.2. Si noti che tutte le operazioni richiedono tempo costante.

Esempio 7.2 Riprendiamo l'Esempio 7.1 con i numeri di matricola e supponiamo di sapere che gli studenti non fuori corso attualmente immatricolati abbiano matricola compresa tra 234717 e 235717. Poniamo quindi $U = [234717, 235717]$ e usiamo la funzione hash perfetta $h(k) = k - 234717$. La nostra tavola hash avrà dimensione $m = |U| = 1000$, e quindi per 100 studenti avremo un fattore di carico $\alpha = 100/1000$, pari quindi al 10%. □

[1] to hash *v.tr.* 1 tritare, sminuzzare (carne) 2 (*fig.*) pasticciare. *Il Nuovo Dizionario Hazon Garzanti inglese-italiano italiano-inglese*, 1996. Nel nostro contesto, il termine hash indica proprio l'operazione di "tritare" una chiave, stravolgendone il valore per trasformarla in un indice.

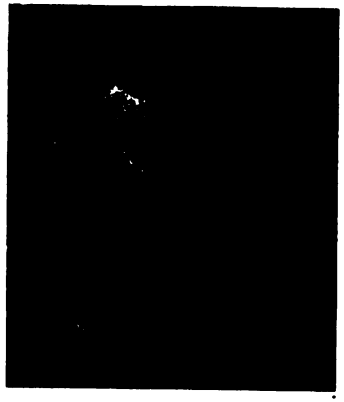


Figura 7.3 Miniatura da un libro d'ore di maestro Fastolf, Francia (1440-1450). Oxford, Biblioteca Bodleian.

Sfortunatamente, in molti casi pratici l'universo delle chiavi è molto grande, e dunque l'uso di funzioni hash perfette richiede comunque un grande spreco di memoria. Se una funzione hash non è perfetta, allora esistono due o più chiavi diverse che hanno associato lo stesso indice. Se nel nostro dizionario vengono a trovarsi simultaneamente chiavi con lo stesso valore della funzione hash, abbiamo una *collisione*. Usare funzioni hash non perfette rende quindi necessario applicare delle strategie di risoluzione delle collisioni, come vedremo nel Paragrafo 7.3. In ogni caso, minore è la probabilità di collisione, migliori saranno le prestazioni di un dizionario basato su tavola hash. In questo contesto, misureremo la "bontà" di una funzione hash in base al numero atteso di collisioni che si avrebbero assumendo di inserire nel dizionario chiavi prese a caso dall'universo U.

Osserviamo infine che indipendentemente dalla funzione hash usata potrebbe essere comunque impossibile evitare collisioni, come nel caso in cui più di un elemento ha la stessa chiave.

Esempio 7.3 (Paradosso del compleanno) In modo simile a quanto visto nell'Esempio 7.1, supponiamo di dover mantenere un dizionario con i nomi degli studenti che frequentano il corso di *Algoritmi e Strutture Dati*, in modo da poterli facilmente cercare questa volta per data di nascita. In questo scenario, se due studenti hanno la stessa data di nascita si ha una collisione. Assumendo per semplicità che tutti gli studenti siano nati in un giorno a caso nello stesso anno non bisestile, ci chiediamo con quale probabilità troviamo almeno due studenti che festeggiano il compleanno nello stesso giorno. Sorprendentemente, in una classe con soli 23 studenti abbiamo una probabilità di poco inferiore al 50% che questo accada! Questo risultato matematico, discusso nel Problema 7.7, è noto come il *paradosso del compleanno*, non perché conduca a una contraddizione logica, ma in quanto contraddice una intuizione comune. □

7.2.1 Definizione di funzioni hash

Assumiamo di sapere che ciascuna chiave $k \in U$ abbia probabilità $\mathcal{P}(k)$ di essere presente nel nostro dizionario. Così come nella coltivazione dei campi una semina è tanto più efficace quanto più i semi sono distribuiti uniformemente nei solchi (vedi Figura 7.3), per funzionare bene una funzione hash deve essere in grado di distribuire uniformemente le chiavi nello spazio degli indici $\{0, \ldots, m-1\}$, tenendo conto della loro probabilità di essere usate. Questa proprietà, che chiameremo *uniformità semplice*, può essere formalizzata come segue.

Definizione 7.4 (Uniformità semplice) *Sia*

$$Q(i) = \sum_{k:h(k)=i} \mathcal{P}(k)$$

la probabilità che, scegliendo una chiave a caso, questa finisca nella cella i. Una funzione hash h gode della proprietà di uniformità semplice *se per ciascun indice* $i \in \{0, \ldots, m-1\}$,

$$Q(i) = \frac{1}{m}.$$

Usando una funzione hash che gode dell'uniformità semplice, ogni cella ha la stessa probabilità di essere usata, cioè non ci sono fenomeni di *agglomerazione*, dove più chiavi tendono a collidere su alcune celle più che su altre.

Esempio 7.4 Se U è l'insieme dei numeri reali in $[0, 1]$ e ogni chiave ha la stessa probabilità di essere scelta, allora si può dimostrare che $h(k) = \lfloor km \rfloor$ soddisfa la proprietà di uniformità semplice. □

Per definire una funzione hash che soddisfi l'uniformità semplice, la distribuzione di probabilità delle chiavi $\mathcal{P}$ deve essere nota, e questo non è sempre il caso. In mancanza di informazioni statistiche su quali chiavi occorrono con più frequenza di altre, una possibilità è quella di assumere che ogni chiave sia equiprobabile, cioè $\mathcal{P}(k) = 1/|U|$. In generale comunque, più è accurata la conoscenza di $\mathcal{P}$ nello scenario applicativo di interesse, migliore è la funzione hash che siamo in grado di costruire per quello scenario.

Vediamo ora come costruire funzioni hash che abbiano buone caratteristiche di uniformità. Nel seguito, assumeremo che ogni chiave ha uguale probabilità di essere scelta da U ($\mathcal{P}$ uniforme). Inoltre, salvo indicazione contraria, assumeremo di avere a che fare sempre con chiavi numeriche intere non negative. Per trattare con chiavi arbitrarie non intere o non numeriche, osserviamo che ci si può sempre ricondurre al caso delle chiavi intere: consideriamo un qualsiasi valore y (ad es. un numero in virgola mobile, una stringa, ecc.), e sia $y_0 \cdot y_1 \ldots y_\ell$ la rappresentazione binaria di y con $y_i \in \{0, 1\}$ per ogni $i \in \{0, \cdots, \ell\}$. Allora possiamo associare a y il numero

$$b(y) = \sum_{i=0}^{\ell} y_i \cdot 2^i$$

ottenuto guardando i bit della rappresentazione di y come un numero binario. Se dunque y non è un intero, possiamo usare come chiave l'intero non negativo $b(y)$ al posto di y.

Metodo della divisione. Un primo metodo semplicissimo per definire una funzione hash è quello della *divisione*:

$$h(k) = k \bmod m. \tag{7.1}$$

Il metodo calcola il resto della divisione della chiave k per m, dove m è la dimensione della tavola. Sebbene nella maggior parte degli scenari pratici questo metodo dà buoni risultati, vi sono situazioni in cui potremmo essere particolarmente sfortunati ed andare incontro a molte collisioni. Come situazione estrema, se l'universo fosse fatto interamente da chiavi che sono multipli di m, allora si avrebbe $h(k) = 0$ per ogni chiave k, e quindi tutte le chiavi collideremmo nella stessa cella. Questo implicherebbe che $Q(0) = 1$ e $Q(i) = 0$ per ogni $i \in \{1, \ldots, m-1\}$; tutto il contrario dell'uniformità semplice che vorremmo avere. In genere, la bontà del metodo della divisione dipende dalla scelta di m, come discusso nel seguente esempio.

Esempio 7.5 Supponiamo di voler mantenere la tavola degli identificatori di un compilatore usando una tavola hash e il metodo della divisione, in modo da poter verificare velocemente se un identificatore è definito. In un programma, è frequente avere identificatori che hanno lo stesso prefisso (es. `temp1`, `temp2`, `temp3`). Se scegliamo ad esempio $m = 2^i$ per qualche i, tutte le chiavi che hanno i primi i bit uguali collideranno nella stessa cella (come succederebbe con `temp1`, `temp2`, `temp3` se i non fosse abbastanza grande), pur avendo spazio a volontà per contenerli in celle diverse. La scelta $m = 2^i$ non è quindi in genere una buona scelta per il metodo della divisione. □

Una buona funzione hash dovrebbe dipendere da tutti i bit di una chiave. In generale, una buona scelta pratica è prendere m uguale a un numero primo non troppo vicino a una potenza di due.

Metodo del ripiegamento. Supponiamo di suddividere la rappresentazione di una chiave k in parti: $k_1, k_2, \ldots, k_\ell$. Questo ci permette di usare il metodo del *ripiegamento*:

$$h(k) = f(k_1, k_2, \ldots, k_\ell) \tag{7.2}$$

dove f è una opportuna funzione a più variabili avente codominio $\{0, \ldots, m-1\}$.

Esempio 7.6 Supponiamo che le nostre chiavi siano numeri di carta di credito. Se k = 4772 6453 7348 1881, possiamo ottenere le sottochiavi $k_1 = 4772$, $k_2 = 6453$, $k_3 = 7348$, $k_4 = 1881$ e definire $f(k_1, k_2, k_3, k_4) = (k_1 + k_2 + k_3 + k_4) \bmod m$. □

classe `TavolaHashListeColl` **implementa** `Dizionario`:
dati: $S(m, n) = \Theta(m + n)$
un array v di dimensione m in cui ogni cella contiene un puntatore a una lista di coppie $(elem, chiave)$. Un elemento e con chiave $k \in U$ è nel dizionario se e solo se (e, k) è nella lista puntata da $v[h(k)]$, con $h : U \to \{0, \ldots, m-1\}$ funzione hash con uniformità semplice calcolabile in tempo $O(1)$.

operazioni:
`insert`$(elem\ e, chiave\ k)$ $T(n) = O(1)$
aggiungi la coppia (e, k) alla lista puntata da $v[h(k)]$.

`delete`$(chiave\ k)$ $T_{avg}(n) = O(1 + n/m)$
rimuovi la coppia (e, k) nella lista puntata da $v[h(k)]$.

`search`$(chiave\ k) \to elem$ $T_{avg}(n) = O(1 + n/m)$
se (e, k) è nella lista puntata da $v[h(k)]$, allora restituisci e, altrimenti restituisci `null`.

Figura 7.4 Dizionario realizzato mediante tavola hash con liste di collisione.

7.3 Risoluzione delle collisioni

Come abbiamo visto nell'Esempio 7.3, non sembra facile evitare situazioni di collisione, indipendentemente dalla bontà della funzione hash usata. In questo paragrafo studieremo le due principali tecniche di risoluzione delle collisioni: le *liste di collisione* e l'*indirizzamento aperto*.

7.3.1 Liste di collisione

Questo metodo consiste nell'associare ad ogni cella della tavola hash una lista di chiavi, detta *lista di collisione*, piuttosto che una singola chiave. In questo modo, se due chiavi collidono sulla stessa cella, verranno entrambe a trovarsi nella lista di collisione di quella cella. Per verificare se una chiave k è presente nel dizionario, dovremmo quindi cercarla nella lista di collisione della cella con indice $h(k)$. Una soluzione comune è quella di rappresentare le liste di collisione come liste concatenate non ordinate, mantenendo nella cella della tavola hash un puntatore al primo elemento della lista. Si osservi che la lunghezza media di una lista di collisione sarà pari al fattore di carico $\alpha = n/m$. Pertanto, assumendo di usare una funzione hash che goda della uniformità semplice, il tempo necessario per rispondere mediante ricerca sequenziale ad una operazione di ricerca sarà in media $T_{avg}(n, m) = O(1 + n/m)$ [2]. Si osservi che, diversamente dalle tavole ad accesso diretto e dalle tavole hash con funzione hash perfetta, usando liste di

[2] Il +1 è presente in $O(1 + n/m)$ perché una ricerca non può richiedere meno di tempo costante.

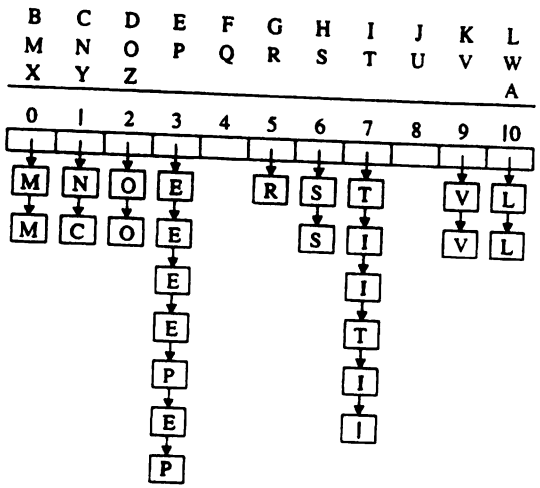


Figura 7.5 Esempio di tavola hash con liste di collisione a fronte dell'inserimento delle lettere della parola "PRECIPITEVOLISSIMEVOLMENTE".

collisione possiamo avere fattori di carico $\alpha > 1$. Notiamo inoltre che con questo metodo una stessa chiave può occorrere più volte nel dizionario.

Una possibile realizzazione di un dizionario mediante tavola hash con liste di collisione è descritta in Figura 7.4. Poiché la cancellazione richiede prima la ricerca della chiave da cancellare, l'operazione `delete` richiede anch'essa tempo $O(1 + n/m)$ in media.

Le tavole hash con liste di collisione forniscono un ottimo esempio di *bilanciamento spazio-tempo*: per $m = 1$, tutte le n chiavi sono in una sola lista, e la tavola diventa una unica struttura a ricerca sequenziale che richiede tempo $T(n, m) = O(n/m) = O(n)$ per ogni `search` e spazio $S(n) = O(n)$; se invece siamo disposti a usare molto spazio, possiamo usare una funzione hash perfetta ottenendo tempo $T(n) = O(1)$ per ricerca, ma spazio $O(|U|)$ (si ricordi l'Esempio 7.2 con i numeri di matricola). Mostriamo ora un esempio che illustra il funzionamento di una tavola hash con liste di collisione.

Esempio 7.7 Supponiamo che l'universo delle chiavi contenga le lettere dell'alfabeto $U = \{A, B, C, D, \ldots, Z\}$ e poniamo:

$$h(k) = \text{ascii}(k) \bmod m$$

dove $\text{ascii}(k)$ è il codice ASCII del carattere k (ad esempio, $\text{ascii}(A) = 65$, $\text{ascii}(B) = 66$, ecc.). Infine, scegliamo m primo, ad esempio $m = 11$. Figura 7.5 illustra il contenuto della tavola a fronte dell'inserimento delle lettere della parola "PRECIPITEVOLISSIMEVOLMENTE". Il fattore di carico è $\alpha = 26/11 \approx 2{,}36$, che è anche la lunghezza media delle liste di collisione. Si noti il fenomeno

classe `TavolaHashAperta` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

```
    insert(elem e, chiave k)
1.    for i = 0 to m − 1 do
2.      if (v[c(k,i)].elem = null) then
3.        v[c(k,i)] ← (e, k)
4.        return
5.    errore tavola piena

    delete(chiave k)
      errore operazione non supportata

    search(chiave k) → elem
1.    for i = 0 to m − 1 do
2.      if (v[c(k,i)].elem = null) then
3.        return null
4.      if (v[c(k,i)].chiave = k) then
5.        return v[c(k,i)].elem
6.    return null
```

Figura 7.6 Dizionario realizzato mediante tavola hash con indirizzamento aperto. La funzione $c(k, i)$ determina il tipo di indirizzamento aperto usato. Si noti che questa realizzazione non supporta l'operazione `delete`.

di agglomerazione intorno alle celle con indici 4 e 8, mentre le celle 3 e 7 rimangono vuote: questo è in accordo con il fatto che la funzione hash che abbiamo usato non gode della proprietà di uniformità semplice. Infatti, le lettere dell'alfabeto hanno frequenze diverse: ad esempio, la lettera E è molto più frequente della lettera Q. La funzione hash che abbiamo usato non tiene evidentemente conto di questo. □

7.3.2 Indirizzamento aperto

Diversamente dal metodo delle liste di collisione, la tecnica dell'*indirizzamento aperto* mantiene tutte le chiavi nelle celle della tavola, per cui si ha $n \leq m$ e $\alpha \leq 1$. Supponiamo di voler inserire una chiave k e che la sua posizione "naturale" $h(k)$ sia già occupata. L'indirizzamento aperto consiste nell'occupare un'altra cella vuota, anche se essa potrebbe spettare di diritto ad un'altra chiave. Per fare un'analogia con la vita reale, consideriamo un cinema in cui i posti sono assegnati alle persone al momento dell'acquisto del biglietto. D'inverno, portiamo cappotti

ingombranti e spesso, oltre ad occupare il posto che ci spetta, li appoggiamo temporaneamente occupando un posto vicino. Se ci sono molti posti liberi e il posto che ci spetta è occupato dal cappotto di qualcuno, ci può succedere di sederci in un posto diverso da quello a noi assegnato. Mano mano che il cinema si riempie, sempre più persone saranno fuori posto e dovranno passare in rassegna diversi posti prima di trovarne uno vuoto. Per illustrare più precisamente il metodo dell'indirizzamento aperto, vediamo ora come realizzare le operazioni `insert` e `search` di un dizionario (si veda Figura 7.6). Alla fine di questo paragrafo vedremo una realizzazione che supporta anche l'operazione `delete`. Per rappresentare il dizionario usiamo un array v di dimensione m in cui ogni cella contiene una coppia di tipo $(elem, chiave)$. Per segnalare che una cella è vuota, assegneremo il valore speciale `null` al suo campo $elem$.

`insert`$(elem\ e, chiave\ k)$

Verifichiamo dapprima se $v[h(k)]$ è vuota, nel qual caso (e, k) viene scritta in $v[h(k)]$. Altrimenti, si ha una collisione e si deve trovare una cella vuota in cui depositare elemento e chiave. Quello che faremo è partire da $h(k)$, ispezionando le celle della tavola secondo una sequenza opportuna di indici $c(k,0) = h(k), c(k,1), c(k,2), \ldots, c(k,m-1)$ (riga 1), e fermarci non appena viene trovata una cella vuota (riga 2). La sequenza considerata deve contenere tutti gli indici in $\{0, \ldots, m-1\}$ in modo da garantire che, se esiste una cella vuota, il metodo prima o poi la considera.

`search`$(chiave\ k)$

L'operazione `search` è molto simile alla `insert`. Se durante la scansione delle celle viene trovata una cella che contiene la chiave cercata (riga 4) l'operazione restituisce l'elemento trovato. Se invece si arriva ad una cella vuota (riga 2) o si è scandita senza successo tutta la tavola (riga 6), si restituisce `null`. Il fatto che fermarsi alla prima cella vuota sia corretto è una conseguenza del seguente teorema.

Teorema 7.1 *Sia v una tavola hash mantenuta con le operazioni descritte in Figura 7.6 e sia k una chiave qualsiasi. Se j è il più piccolo intero per cui $v[c(k,j)].elem =$ `null`, allora $v[c(k,q)].chiave \neq k$ per ogni $j < q < m$.*

Dimostrazione. Supponiamo per assurdo che esista un intero $q > j$ per cui $v[c(k,q)].chiave = k$. Consideriamo l'operazione di `insert` che ha inserito k in $v[c(k,q)]$. Siccome le chiavi non vengono mai cancellate, allora $v[c(k,j)]$ deve essere stata vuota già a quel tempo. Ma allora, in base alla riga 2, `insert` avrebbe posto k in $v[c(k,j)]$, che è la prima cella vuota, e non in $v[c(k,q)]$. □

Osserviamo che le prestazioni delle operazioni implementate, che discuteremo alla fine del capitolo, dipendono dalla particolare funzione $c(k,i)$ utilizzata. Poiché come abbiamo visto $n \leq m$, lo spazio richiesto da una tavola hash con indirizzamento aperto è $\Theta(m)$.

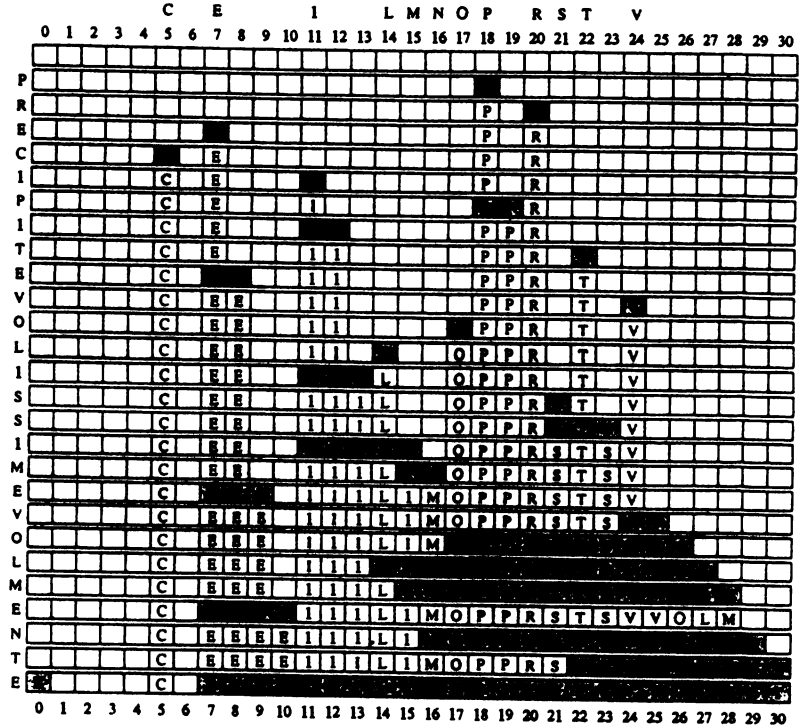


Figura 7.7 Successive tavole hash ottenute a partire dalla tavola vuota inserendo le lettere della parola "PRECIPITEVOLISSIMEVOLMENTE" usando indirizzamento aperto con scansione lineare. In questo esempio, il numero complessivo di celle scandite durante tutti gli inserimenti è 124, che diviso per 26 inserimenti dà circa 4,8 celle scandite in media per ogni inserimento.

Scansione lineare. Il metodo di *scansione lineare* si ottiene se la sequenza di scansione è definita come segue:

$$c(k,i) = (h(k) + i) \bmod m \tag{7.3}$$

per i tale che $0 \leq i < m$.

Esempio 7.8 Come nell'Esempio 7.7, supponiamo che l'universo delle chiavi sia $U = \{A, B, C, D, \ldots, Z\}$ e poniamo:

$$h(k) = \text{ascii}(k) \bmod m$$

dove $\text{ascii}(k)$ è il codice ASCII del carattere k. In Figura 7.7 sono illustrate le celle ispezionate inserendo ciascuna delle lettere della parola "PRECIPITEVOLISSI-

MEVOLMENTE" usando il metodo di scansione lineare. Le celle scandite sono evidenziate in grigio. Nell'esempio, il numero complessivo di celle scandite durante tutti gli inserimenti è 124, che diviso per 26 inserimenti dà circa 4,8 celle scandite in media per ogni inserimento. Si noti che la situazione peggiora drasticamente quando la tavola inizia a riempirsi. □

Si noti come nell'Esempio 7.8 tendano a formarsi degli agglomerati sempre più lunghi di celle piene (come quello che inizia alla chiave I). Intuitivamente, se un gruppo di celle consecutive sono piene, ogni chiave k tale che $h(k)$ punta nel gnippo verrà accodata alla fine del gnippo stesso, che si allungherà di 1. In effetti, più un agglomerato diventa lungo, più è alta la probabilità che si allunghi ulteriormente. Nel caso di scansione lineare si parla di *agglomerazione primaria*.

Scansione quadratica. Un altro modo classico di definire $c(k,i)$, che dà luogo al metodo di *scansione quadratica*, è il seguente:

$$c(k,i) = \lfloor h(k) + c_1 \cdot i + c_2 \cdot i^2 \rfloor \bmod m \qquad (7.4)$$

per ogni $0 \leq i < m$ e per c_1 e c_2 opportuni. Si può dimostrare che, scegliendo ad esempio $c_1 = c_2 = 0.5$ e m potenza di due, la sequenza $c(k,i)$ contiene tutti gli indici in $\{0,\ldots,m-1\}$, e quindi permette di scandire tutta la tavola.

Nonostante la scansione quadratica distribuisca le chiavi in modo da evitare agglomerazione primaria, ogni coppia di chiavi k_1 e k_2 con $h(k_1) = h(k_2)$ continua a generare la stessa sequenza di scansione. Questo dà luogo alla cosiddetta *agglomerazione secondaria*.

Hashing doppio. Un metodo per eliminare virtualmente il fenomeno dell'agglomerazione è quello dell'*hashing doppio*. L'idea è quella di far dipendere dalla chiave anche il passo di incremento dell'indice usando una seconda funzione hash:

$$c(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m \qquad (7.5)$$

dove h_1 ed h_2 sono due funzioni hash distinte. In questo modo, anche se due chiavi collidono, la sequenza di scansione sarà diversa. Affinché per una certa chiave k la sequenza $c(k,i)$ contenga tutti gli indici in $\{0,\ldots,m-1\}$, è necessario che m e $h_2(k)$ siano primi fra loro. Ad esempio, possiamo scegliere m primo, $h_1(k) = k \bmod m$, e $h_2(k) = (k \bmod (m-1)) + 1$.

Esempio 7.9 Riprendiamo ancora l'Esempio 7.8. In Figura 7.8 sono illustrate le celle ispezionate ad ogni inserimento delle lettere della pamla "PRECIPITEVOLISSIMEVOLMENTE" usando il metodo di hashing doppio con: $m = 31$, $h_1(k) = \text{ascii}(k) \bmod 31$, e $h_2(k) = (h_1(k) \bmod (30)) + 1$. Le celle scandite sono evidenziate in grigio. Nell'esempio, il namero complessivo di celle scandite durante tutti gli inserimenti è 82, che diviso per 26 inserimenti dà circa 3,1 celle scandite in media per ogni inserimento. Si confronti questo risultato con quello ottenuto nell'Esempio 7.8 usando scansione lineare. □

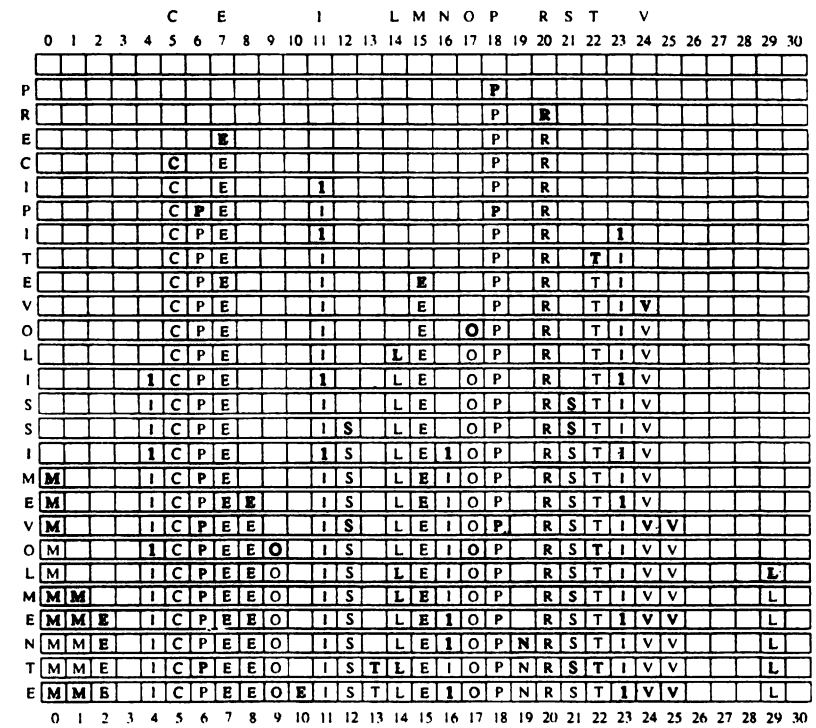


Figura 7.8 Successive tavole hash ottenute a partire dalla tavola vuota inserendo le lettere della parola "PRECIPITEVOLISSIMEVOLMENTE" usando indirizzamento aperto con hashing doppio. In questo esempio, il numero complessivo di celle scandite durante tutti gli inserimenti è 82, che diviso per 26 inserimenti dà circa 3,1 celle scandite in media per ogni inserimento.

Cancellazione di chiavi. Supponiamo di voler cancellare una chiave da una tavola hash con indirizzamento aperto. La prima cosa che ci verrebbe in mente sarebbe forse di marcare in qualche modo come *vuota* la cella dell'array che contiene la chiave. Purtroppo, così facendo, non potremmo più dimostrare il Teorema 7.1 e la `search` come realizzata in Figura 7.6 non sarebbe più corretta: infatti, la ricerca potrebbe interrompersi prima di trovare l'elemento cercato per via di un "buco" creato da una cancellazione. Una soluzione possibile è quella di marcare le celle che hanno subito cancellazione con un valore speciale, ad esempio ponendo il campo *elem* a `canc`. L'inserimento considererà una cella cancellata come se fosse vuota fermandosi su di essa, mentre una ricerca la oltrepasserà, come mostrato in Figura 7.9. L'operazione `delete` è del tutto simile alla `search`: l'unica differenza è che, una volta trovata la cella da liberare, il campo

```
classe TavolaHashApertaBis implementa Dizionario:
dati:                                                  S(m) = Θ(m)
   un array v di dimensione m in cui ogni cella contiene una coppia
   (elem, chiave).

operazioni:
   insert(elem e, chiave k)
1.    for i = 0 to m − 1 do
2.       if (v[c(k,i)].elem = null or v[c(k,i)].elem = canc) then
3.          v[c(k,i)] ← (e, k)
4.          return
5.    errore tavola piena

   delete(chiave k)
1.    for i = 0 to m − 1 do
2.       if (v[c(k,i)].elem = null) then
3.          errore chiave non in dizionario
4.       if (v[c(k,i)].chiave = k and v[c(k,i)].elem ≠ canc) then
5.          v[c(k,i)].elem ← canc
6.    errore chiave non in dizionario

   search(chiave k) → elem
1.    for i = 0 to m − 1 do
2.       if (v[c(k,i)].elem = null) then
3.          return null
4.       if (v[c(k,i)].chiave = k and v[c(k,i)].elem ≠ canc) then
5.          return v[c(k,i)].elem
6.    return null
```

Figura 7.9 Dizionario realizzato mediante tavola hash con indirizzamento aperto in modo da supportare l'operazione `delete`.

elem viene posto al valore speciale `canc`. Questa soluzione non peggiora il tempo di inserimento, ma purtroppo tende a rallentare le operazioni di ricerca, poiché celle cancellate dovranno comunque essere esaminate senza contribuire in alcun modo al successo o all'insuccesso di una ricerca.

Analisi del costo di scansione. Sebbene usando indirizzamento aperto la scansione per la ricerca di un elemento o di una cella vuota possa richiedere tempo $O(n)$ nel caso peggiore, nel caso medio le cose vanno molto meglio, come discusso nel seguente teorema di cui rimandiamo la dimostrazione a Knuth [1].

Teorema 7.2 *Se le chiavi associate agli n elementi in una tavola hash di dimensione m sono prese dall'universo delle chiavi con probabilità uniforme, il numero medio di passi richiesto da una operazione* `search` *(contando in n anche le celle marcate* `canc`*) è:*

esito ricerca	*sc. lineare*	*sc. quadratica / hashing doppio*
chiave trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$-\frac{1}{\alpha}\log_e(1-\alpha)$
chiave non trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$

dove $\alpha = n/m$ è il fattore di carico della tavola.

È facile vedere che il costo di un'operazione `insert` è al più quello di un'operazione `search` nel caso di insuccesso. Il costo di un'operazione `delete` è dominato dalla ricerca della chiave da cancellare, e quindi ha lo stesso costo di una operazione `search`. Partendo quindi dal caso di ricerca con insuccesso discusso nel Teorema 7.2 e usando la definizione $\alpha = n/m$, possiamo concludere che il tempo medio richiesto da ogni operazione `search`, `insert` o `delete` è $O(\frac{m}{(m-n)^2})$ usando scansione lineare, e $O(\frac{m}{m-n})$ usando scansione quadratica o hashing doppio. Si noti che, se solo una frazione costante della tabella è riempita, cioè $\alpha \in [0,1)$, questo vuol dire tempo costante nel caso medio.

7.4 Problemi

Problema 7.1 Proporre una funzione hash perfetta nel caso in cui le chiavi siano stringhe di lunghezza 3 sull'alfabeto {A,B,C}.

Problema 7.2 Considera la realizzazione di un dizionario tramite tavole hash con risoluzione delle collisioni tramite concatenazione. Supponi che le chiavi del dizionario siano stringhe di lettere e che la seguente funzione di hash sia stata definita per una tavola di dimensione $m = 5$: sommare i codici ASCII delle lettere (ascii$(A) = 65$, ascii$(B) = 66$, ecc.), dividere il risultato della somma per 5 e prendere il resto della divisione. Mostra il contenuto della tavola dopo l'inserimento delle stringhe:

MARTE LUNA COMETA PIANETI SOLE SOLI

Ripeti l'esercizio su una tavola hash con risoluzione delle collisioni tramite indirizzamento aperto a scansione lineare.

Problema 7.3 Considera le seguenti realizzazioni di dizionari:

- tavole ad accesso diretto;
- tavole hash con risoluzione delle collisioni tramite concatenazione;
- tavole hash con risoluzione delle collisioni tramite indirizzamento aperto a scansione lineare;
- tavole hash con risoluzione delle collisioni tramite indirizzamento aperto a scansione quadratica.

Dato un elemento con chiave k, discuti come trovare il predecessore dell'elemento in ciascuno dei casi precedenti, dando delimitazioni precise dei tempi di esecuzione in funzione del numero n di elementi nella tavola e della dimensione m della tavola stessa. Discutere quale implementazione è preferibile per supportare l'operazione di ricerca del predecessore.

Problema 7.4 Svolgi il Problema 7.3 assumendo che la funzione hash soddisfi la seguente proprietà di monotonia: date due chiavi $k_1 \leq k_2$, $h(k_1) \leq h(k_2)$.

Problema 7.5 Sia t un qualunque numero intero e sia m la dimensione di una tavola hash. Dimostra che se la cardinalità dell'universo U soddisfa $|U| \geq t\,m$, allora esistono almeno t chiavi distinte, $k_1, \ldots k_t$, tali che $h(k_1) = h(k_2) = \ldots = h(k_t)$.

Problema 7.6 Il professor Del Caso dichiara di aver scoperto un modo semplicissimo per definire una funzione hash che distribuisce uniformemente le chiavi in una tabella:

$$h(k) = \text{numero a caso in } \{1, \ldots, m-1\}$$

Una tavola hash basata su questa funzione randomizzata si comporterebbe correttamente?

Problema 7.7 (Paradosso del compleanno) Dato un insieme di n persone e assumendo che ciascuna di esse sia nata in un giorno a caso in uno stesso anno non bisestile, calcolare la probabilità di avere almeno due persone nate nello stesso giorno. Suggerimento: calcolare dapprima la probabilità che abbiano tutti date di nascita diverse. Assumendo di mettere in fila le n persone, calcolare la probabilità che la seconda abbia una data di nascita diversa dalla prima, la terza dalle prime due, la quarta dalle prime tre, ecc.

7.5 Sommario

In questo capitolo abbiamo visto come realizzare dizionari in cui le operazioni richiedono tempo costante in media. Diversamente dagli alberi di ricerca visti nel Capitolo 6, in cui le posizioni degli elementi vengono determinate in base a confronti fra chiavi, la nuova tecnica che abbiamo introdotto, chiamata *hashing*, posiziona gli elementi e le loro chiavi nelle celle di un array usando come indici le chiavi stesse opportunamente trasformate mediante manipolazioni aritmetiche. Un array di questo tipo è chiamato *tavola hash*. Un problema connesso con questo metodo è quello delle *collisioni*: se chiavi distinte vengono trasformate nello stesso indice, oppure abbiamo elementi con la stessa chiave, ci troviamo a dover memorizzare elementi diversi in una stessa cella eccedendone la capacità. Per ridurre la probabilità di collisioni, la *funzione hash* che trasforma chiavi in indici deve essere in grado di distribuire il più possibile uniformemente le chiavi in una tavola. A questo scopo, abbiamo visto vari modi di definire funzioni hash con buone caratteristiche di uniformità.

Nella seconda parte del capitolo, abbiamo studiato due diverse tecniche di risoluzione delle collisioni: le *liste di collisione* e l'*indirizzamento aperto*. Il primo metodo consiste semplicemente nel mantenere in ogni cella un puntatore a una lista esterna di elementi, piuttosto che un singolo elemento. In base al secondo metodo, ogni cella contiene invece un solo elemento; se la cella che spetterebbe di diritto ad un elemento da inserire è già occupata, si cerca un'altra cella libera esplorando una opportuna sequenza di indici.

7.6 Note bibliografiche

Secondo Knuth, le tavole hash sono state inventate intorno agli anni '50 da H. P. Luhn, che ha anche introdotto il metodo delle liste di collisione. Circa nello stesso periodo, G. M. Amdahl propose la tecnica dell'indirizzamento aperto con scansione lineare. Sebbene il termine "hashing" fosse già in uso negli anni sessanta in ambito informatico, non si ha notizia di pubblicazioni che lo usassero fino al 1967 circa. Ottime trattazioni dell'hashing appaiono nei testi di Knuth [1] e Gonnet [2].

Riferimenti bibliografici

[1] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[2] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.

8

Code con priorità

Un professore di filosofia era in piedi davanti alla sua classe, prima della lezione e aveva davanti a sè alcuni oggetti. Quando la lezione cominciò, senza proferire parola il professore prese un grosso vaso di vetro vuoto e lo riempì con delle rocce di 5-6 cm di diametro. Quindi chiese agli studenti se il vaso fosse pieno ed essi annuirono. Allora il professore prese una scatola di sassolini e li versò nel vaso di vetro, scuotendolo appena. I sassolini, ovviamente, rotolarono negli spazi vuoti fra le rocce. Il professore quindi chiese ancora se il vaso ora fosse pieno ed essi furono d'accordo. Gli studenti cominciarono a ridere, quando il professore prese una scatola di sabbia e la versò nel vaso. La sabbia riempì ogni spazio vuoto. "Ora", disse il professore, "voglio che voi riconosciate che questa è la vostra vita. Le rocce sono le cose importanti – la famiglia, il partner, la salute, i figli – anche se ogni altra cosa dovesse mancare, e solo queste rimanere, la vostra vita sarebbe comunque piena. I sassolini sono le altre cose che contano, come il lavoro, la casa, la moto, l'auto. La sabbia rappresenta qualsiasi altra cosa, le piccole cose. Se voi riempite il vaso prima con la sabbia, non ci sarà più spazio per rocce e sassolini. Lo stesso per la vostra vita; se voi spendete tutto il vostro tempo ed energie per le piccole cose, non avrete mai spazio per le cose veramente importanti. Quindi stabilite le vostre priorità, il resto è solo sabbia."
Dopo queste parole, uno studente si alzò e prese il vaso contenente rocce, sassolini e sabbia, che tutti consideravano pieno e cominciò a versagli dentro un bicchiere di birra. Ovviamente la birra si infilò nei rimanenti spazi vuoti e riempì veramente il vaso fino all'orlo. Lo studente si rivolse al professore e alla classe e a questo punto disse: "Non importa quali priorità abbiate stabilito, non importa quanto piena sia la vostra vita, c'è sempre spazio per una buona birra!"

(Anonimo)

Una coda con priorità è un tipo di dato che permette di mantenere il minimo (o il massimo) in un insieme di chiavi su cui è definita una relazione d'ordine totale. Le operazioni fondamentali che una coda con priorità deve supportare sono l'inserimento di un nuovo elemento (`insert`), la ricerca del minimo (`findMin`) e l'estrazione del minimo (`deleteMin`). Ulteriori operazioni tipicamente supportate sono la cancellazione di un elemento (`delete`), l'incremento di una chiave (`increaseKey`) e il decremento di una chiave (`decreaseKey`). Le implementazioni più avanzate permettono anche la fusione (`merge`) di due code con priorità in una unica coda con priorità. In Figura 8.1 diamo una descrizione schematica del tipo di dato `CodaPriorita`.

tipo `CodaPriorita`:
dati:
un insieme S di n elementi di tipo *elem* a cui sono associate chiavi di tipo *chiave* prese da un universo totalmente ordinato.

operazioni:
`findMin()` $\rightarrow$ *elem*
restituisce l'elemento in S con la chiave minima.

`insert`(*elem e, chiave k*)
aggiunge a S un nuovo elemento e con chiave k.

`delete`(*elem e*)
cancella da S l'elemento e.

`deleteMin()`
cancella da S l'elemento con chiave minima.

`increaseKey`(*elem e, chiave d*)
incrementa della quantità d la chiave dell'elemento e in S.

`decreaseKey`(*elem e, chiave d*)
decrementa della quantità d la chiave dell'elemento e in S.

`merge`(`CodaPriorita` c_1, `CodaPriorita` c_2) $\rightarrow$ `CodaPriorita`
restituisce una nuova coda con priorità $c_3 = c_1 \cup c_2$.

Figura 8.1 Dati e operazioni di una `CodaPriorita`.

In questo capitolo vedremo tre diverse implementazioni di code con priorità, realizzate rispettivamente mediante d-heap, heap binomiali e heap di Fibonacci. Per ciascuna di esse, discuteremo lo spazio di memoria richiesto e il tempo di esecuzione di ciascuna operazione. Nel caso dei heap di Fibonacci, ricorreremo all'*analisi ammortizzata* introdotta nel Capitolo 2, analizzando quindi non tanto il costo di una singola operazione, quanto il costo di una intera sequenza di operazioni.

Nel corso del libro incontreremo numerose applicazioni delle code con priorità per progettare algoritmi efficienti: in particolare, per un uso esemplificativo, rimandiamo al Capitolo 12 e al Capitolo 13.

8.1 d-heap

In questo paragrafo vedremo come realizzare una coda con priorità utilizzando la struttura dati d-heap, che può essere definita come segue:

Definizione 8.1 (d-heap) *Un d-heap è un albero radicato d-ario con le seguenti proprietà:*

1. Struttura: *un d-heap di altezza h è completo almeno fino a profondità $h-1$.*

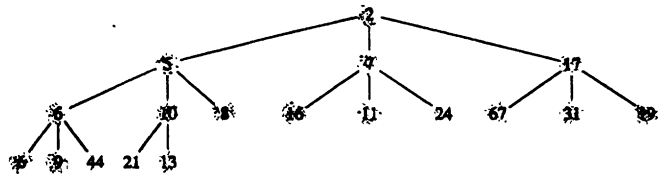


Figura 8.2 Esempio di d-heap con 18 nodi e $d = 3$. Nei nodi sono mostrate solo le chiavi.

2. Contenuto informativo: *ciascun nodo v contiene un elemento $elem(v)$ e una chiave $chiave(v)$ presa da un universo totalmente ordinato.*
3. Ordinamento a heap: *ciascun nodo v diverso dalla radice ha una chiave non inferiore a quella del suo genitore: $chiave(v) \geq chiave(padre(v))$.*

Un esempio di d-heap per $d = 3$ è mostrato in Figura 8.2. Si noti che abbiamo già incontrato la struttura dati d-heap per $d = 2$ nel Capitolo 4, quando abbiamo trattato l'algoritmo di ordinamento `heapSort`; in quel caso per motivi di convenienza mantenevamo massimi invece che minimi. La proprietà 1 dei d-heap implica che la differenza di profondità fra qualunque coppia di foglie è al più 1, e quindi un d-heap è quasi perfettamente bilanciato. Inoltre, come vedremo negli enunciati dei prossimi lemmi, l'altezza di un d-heap sarà logaritmica nel suo numero di nodi e l'elemento con chiave minima sarà sempre contenuta nella radice.

Lemma 8.1 *Un d-heap con n nodi ha altezza $O(\log_d n)$.*

Dimostrazione. Sia h l'altezza del d-heap con n nodi. Poiché un d-heap è completo almeno fino a profondità $h-1$, e un albero d-ario completo di profondità $h-1$ ha $\frac{d^h-1}{d-1}$ nodi (vedi Appendice, Relazione 17.7), allora $\frac{d^h-1}{d-1} < n$. Da questo si ricava $d^h < n(d-1)+1$, e quindi $h < \log_d[n(d-1)+1] = O(\log_d n)$. □

Lemma 8.2 *In un d-heap la radice contiene un elemento con chiave minima.*

Dimostrazione. Dimostriamo il lemma per induzione sul numero di nodi di un d-heap. Se $n \leq 1$, il caso base è banalmente verificato. Assumiamo ora per ipotesi induttiva che l'enunciato valga per ogni d-heap con al più $n-1$ nodi. Consideriamo ora un d-heap T con n nodi e siano $T_1, \ldots, T_h$, con $1 \leq h \leq d$, i sottoalberi di T radicati nei figli della sua radice. Chiaramente, ogni T_i per $1 \leq i \leq h$, ha al più $n-1$ nodi. Inoltre, poiché ogni sottoalbero di un d-heap è anch'esso un d-heap, ogni T_i per $1 \leq i \leq h$ sarà un d-heap. Quindi, per ipotesi induttiva, la radice di T_i contiene il minimo di T_i. Poiché la chiave della radice di T è minore o uguale a quella di ogni suo figlio, essa sarà il minimo di T. □

classe DHeap **implementa** CodaPriorita:
dati: $S(n) = O(n)$
un d-heap T con n nodi, ciascuno contenente un elemento di tipo *elem* e una chiave di tipo *chiave* presa da un universo totalmente ordinato.

operazioni:
findMin() → *elem* $T(n) = O(1)$
restituisce l'elemento nella radice di T.

insert(*elem e*, *chiave k*) $T(n) = O(\log_d n)$
crea un nuovo nodo v con elemento e e chiave k, in modo che diventi una foglia sull'ultimo livello di T. La proprietà dell'ordinamento a heap viene poi ripristinata spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

delete(*elem e*) $T(n) = O(d \log_d n)$
scambia il nodo v contenente l'elemento e con una qualunque foglia u sull'ultimo livello di T, e poi elimina v. Ripristina infine la proprietà dell'ordinamento a heap spingendo il nodo u verso la sua posizione corretta scambiandolo ripetutamente con il proprio padre o con il proprio figlio contenente la chiave più piccola (si veda la Figura 8.4). Questa operazione può essere usata anche per implementare deleteMin.

decreaseKey(*elem e*, *chiave d*) $T(n) = O(\log_d n)$
decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

increaseKey(*elem e*, *chiave d*) $T(n) = O(d \log_d n)$
aumenta il valore della chiave nel nodo contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso il basso tramite ripetuti scambi di nodi.

merge(CodaPriorita c_1, CodaPriorita c_2) → CodaPriorita
errore operazione non supportata

Figura 8.3 Coda con priorità realizzata mediante d-heap.

Realizzazione

In Figura 8.3 descriviamo le idee di alto livello di una possibile realizzazione di una coda con priorità basata su d-heap, che chiameremo classe DHeap. Ogni oggetto DHeap mantiene un d-albero con n nodi, ciascuno contenente un elemento di tipo *elem* e una chiave di tipo *chiave* presa da un universo totalmente ordinato. Lo spazio richiesto da ogni oggetto DHeap è quindi $O(n)$. Analizziamo ora in maggiore dettaglio come possono essere implementate le operazioni.

findMin() → *elem*

Poiché per il Lemma 8.2 la radice del d-heap contiene una chiave minima, l'operazione findMin restituisce semplicemente l'elemento in essa contenuto.

procedura muoviAlto(v) $T(n) = O(H(n))$
1. **while** ($v \neq$ *radice*(T) **and** *chiave*(v) < *chiave*(*padre*(v))) **do**
2. scambia di posto v e *padre*(v) in T

procedura muoviBasso(v) $T(n) = O(D(n) \cdot H(n))$
1. **repeat**
2. sia u il figlio di v con la minima *chiave*(u), se esiste $T(n) = O(D(n))$
3. **if** (v non ha figli o *chiave*(v) ≤ *chiave*(u)) **break**
4. scambia di posto v e u in T

Figura 8.4 Procedure muoviAlto e muoviBasso per il ripristino della proprietà di ordinamento a heap. $H(n)$ è l'altezza dell'albero e $D(n)$ è il numero massimo di figli di ogni nodo. In un d-heap, $H(n) = \log_d n$ e $D(n) = d$.

insert(*elem e*, *chiave k*)

L'inserimento avviene creando dapprima un nuovo nodo v, a cui vengono associati i dati e e k. Il nodo viene aggiunto all'albero T in modo che diventi una foglia sull'ultimo livello. Questo garantisce che la proprietà di struttura è soddisfatta. Tuttavia, il valore *chiave*(v) potrebbe essere più piccolo di *chiave*(*padre*(v)), e quindi la proprietà di ordinamento a heap potrebbe essere violata. Per ripristinare la proprietà di ordinamento, basta far risalire il nodo v verso l'alto nell'albero scambiando ripetutamente il nodo v con *padre*(v), fermandosi non appena *chiave*(v) ≥ *chiave*(*padre*(v)) o v diventa la radice. Questa procedura, chiamata muoviAlto, è descritta in Figura 8.4. Si noti che lo scambio di nodi non altera la struttura dell'albero. In base al Lemma 8.1 l'altezza dell'albero è $O(\log_d n)$, e quindi muoviAlto farà al più $O(\log_d n)$ scambi. Il tempo totale richiesto per un inserimento sarà dunque $O(\log_d n)$ nel caso peggiore.

delete(*elem e*)

Per cancellare il nodo v contenente l'elemento e, esso viene dapprima scambiato con una qualunque foglia u sull'ultimo livello dell'albero. Ora v, che viene a trovarsi al posto di u sull'ultimo livello, può essere eliminato senza distruggere la proprietà di struttura. Il problema è che u è stato cambiato di posizione, e quindi la proprietà di ordinamento a heap potrebbe essere violata. Poiché non sappiamo se u debba salire o scendere nell'albero, richiameremo su u entrambe le procedure muoviAlto e muoviBasso di Figura 8.4, in modo che quella applicabile riposizioni appropriatamente u nell'albero. Si noti che muoviBasso è più inefficiente di muoviBasso poiché ad ogni iterazione deve cercare il figlio con chiave minima fra gli al più d possibili figli, per poi effettuare lo scambio. In base al Lemma 8.1 l'altezza dell'albero è $O(\log_d n)$, e quindi muoviBasso farà anch'essa al più $O(\log_d n)$ iterazioni, ciascuna di costo $O(d)$. Il tempo totale richiesto per una cancellazione sarà dunque $O(d \log_d n)$ nel caso peggiore.

`deleteMin()`

L'operazione si realizza in tempo $O(d \log_d n)$ semplicemente richiamando `delete` in modo che cancelli la radice del d-heap.

`decreaseKey(elem e, chiave d)`

L'operazione decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto mediante una chiamata `muoviAlto`(v).

`increaseKey(elem e, chiave d)`

L'operazione incrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso il basso mediante una chiamata `muoviBasso`(v).

L'operazione di fusione di due code con priorità (`merge`) non è normalmente supportata usando i d-heap, poiché risulterebbe particolarmente inefficiente. Ad esempio, essa può essere realizzata in tempo $O(n)$ usando la procedura `heapify` vista nel Paragrafo 4.3.1 del Capitolo 4 applicata alla concatenazione dei due d-heap da fondere, assumendo che n sia il numero totale di elementi in essi contenuti. Vedremo in seguito implementazioni di code con priorità che supportano invece questa operazione in modo molto efficiente.

Il seguente teorema riassume i tempi di esecuzione di ciascuna delle operazioni della classe `DHeap` vista in questo paragrafo.

Teorema 8.1 *Usando la struttura dati d-heap con $d = 2$, è possibile realizzare l'operazione* `findMin` *in tempo costante e le operazioni* `insert`, `delete`, `deleteMin`, `increaseKey` *e* `decreaseKey` *in tempo $O(\log n)$ nel caso peggiore. L'operazione* `merge` *può essere realizzata in tempo $O(n)$.*

8.2 Heap binomiali

Un'altra possibile realizzazione del tipo `CodaPriorita` è basata su una struttura dati chiamata *heap binomiale*. Diversamente dai d-heap visti precedentemente, gli heap binomiali supportano efficientemente la fusione di due heap in uno. Essi sono basati su alberi con una particolare struttura chiamati *alberi binomiali*.

Definizione 8.2 (Albero binomiale) *Un* albero binomiale B_h *è definito ricorsivamente come segue:*

1. *B_0 consiste di un unico nodo.*
2. *Per $i \geq 0$, B_{i+1} è ottenuto fondendo due alberi binomiali B_i in modo che la radice dell'uno diventi figlia della radice dell'altro.*

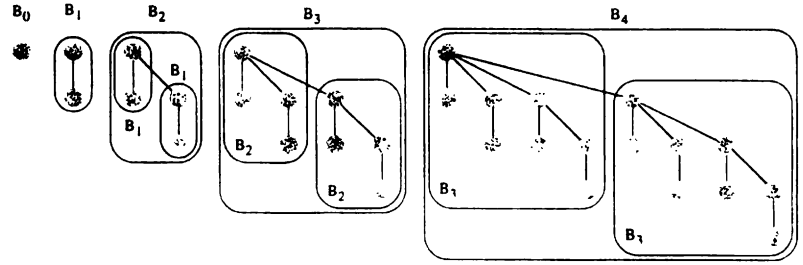


Figura 8.5 Alberi binomiali $B_0, \ldots, B_4$ e loro struttura ricorsiva.

I primi cinque alberi binomiali sono mostrati in Figura 8.5. Diamo ora alcune proprietà basilari di B_h, che possono facilmente essere dimostrate per induzione (vedi Problema 8.1).

Proprietà 8.1 *Un albero binomiale B_h gode delle seguenti proprietà:*

1. *Numero di nodi ($|B_h|$): $n = 2^h$.*
2. *Grado della radice: $D(n) = \log_2 n$*
3. *Altezza: $H(n) = h = \log_2 n$.*
4. *Figli della radice: i sottoalberi radicati nei figli della radice di B_h sono B_0, $B_1, \ldots, B_{h-1}$.*

In base a queste proprietà, altezza e grado della radice di un albero binomiale sono entrambe logaritmiche nella sua dimensione. Veniamo ora alla definizione di un *heap binomiale*.

Definizione 8.3 (Heap binomiale) *Un* heap binomiale *è una foresta di alberi binomiali B_i con le seguenti proprietà:*

1. Struttura: *ogni albero B_i nella foresta è un albero binomiale.*
2. Unicità: *per ogni i, esiste al più un B_i nella foresta.*
3. Contenuto informativo: *ogni nodo v nella foresta contiene un elemento $elem(v)$ e una* chiave *$chiave(v)$ presa da un universo totalmente ordinato.*
4. Ordinamento a heap: *in ogni B_i, ciascun nodo v diverso dalla radice ha una chiave non inferiore a quella del suo genitore: $chiave(v) \geq chiave(padre(v))$.*

Come visto nel Lemma 8.2 per i d-heap, la proprietà di ordinamento a heap garantisce che la chiave nella radice di ogni albero binomiale B_i è minima in quell'albero.

Lemma 8.3 *In un heap binomiale con n nodi, vi saranno al più $A(n) = \lfloor \log_2 n \rfloor + 1$ alberi binomiali.*

classe `HeapBinomiale` **implementa** `CodaPriorita`:
dati: $S(n) = O(n)$
una foresta H con n nodi, ciascuno contenente un elemento di tipo *elem* e una chiave di tipo *chiave* presa da un universo totalmente ordinato.

operazioni:

`findMin()` $\rightarrow$ *elem* $T(n) = O(\log n)$
scorre le radici in H e restituisce l'elemento a chiave minima.

`insert`(*elem e*, *chiave k*) $T(n) = O(\log n)$
aggiunge ad H un nuovo B_0 con dati e e k. Ripristina poi la proprietà di unicità in H mediante fusioni successive dei doppioni B_i.

`deleteMin()` $T(n) = O(\log n)$
trova l'albero T_h con radice a chiave minima. Togliendo la radice a T_h, esso si spezza in h alberi binomiali $T_0, \ldots, T_{h-1}$, che vengono aggiunti ad H. Ripristina poi la proprietà di unicità in H mediante fusioni successive dei doppioni B_i.

`decreaseKey`(*elem e*, *chiave d*) $T(n) = O(\log n)$
decrementa di d la chiave nel nodo v contenente l'elemento e. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

`delete`(*elem e*) $T(n) = O(\log n)$
richiama `decreaseKey`$(e, -\infty)$ e poi `deleteMin()`.

`increaseKey`(*elem e*, *chiave d*) $T(n) = O(\log n)$
richiama `delete`(e) e poi `insert`$(elem, k + d)$, dove k è la chiave associata all'elemento e.

`merge`(`CodaPri`. c_1, `CodaPri`. c_2) $\rightarrow$ `CodaPri`. $T(n) = O(\log n)$
unisce gli alberi in c_1 e c_2 in un nuovo heap binomiale c_3. Ripristina poi la proprietà di unicità nell'heap binomiale c_3 mediante fusioni successive dei doppioni B_i.

Figura 8.6 Coda con priorità realizzata mediante heap binomiale.

Dimostrazione. Se i è il più grande intero tale che $|B_i| = 2^i \leq n$, e cioè $i = \lfloor \log_2 n \rfloor$, allora nessun albero B_j con $j > i$ potrà essere presente nel heap. Poiché per la proprietà di unicità può esserci al più un B_j per ogni j tale che $0 \leq j \leq i$, in tutto avremo al più $A(n) = \lfloor \log_2 n \rfloor + 1$ alberi binomiali nel heap. □

Realizzazione

In Figura 8.6 descriviamo le idee di alto livello di una possibile realizzazione di una coda con priorità basata sulla struttura dati heap binomiale, che chiameremo classe `HeapBinomiale`. Ogni oggetto `HeapBinomiale` mantiene una foresta con n nodi, ciascuno contenente un elemento di tipo *elem* e una chiave di tipo

procedura `ristruttura()`
1. $i = 0$
2. **while** (esistono ancora due B_i) **do**
3. si fondono i due B_i per formare un albero B_{i+1}, ponendo la radice con chiave più piccola come genitore della radice con chiave più grande
4. $i = i + 1$

Figura 8.7 Procedura `ristruttura` per il ripristino della proprietà di unicità in un heap binomiale.

chiave presa da un universo totalmente ordinato. Per accedere agli alberi contenuti nel heap, manteniamo le loro radici in una lista. Lo spazio richiesto da ogni oggetto `HeapBinomiale` è quindi $O(n)$. Analizziamo ora in maggiore dettaglio come le operazioni possono essere implementate.

`findMin()` $\rightarrow$ *elem*

Il minimo di ciascun albero è nella sua radice, quindi abbiamo solo bisogno di trovare il minimo nella lista delle radici. In base al Lemma 8.3, questo richiede tempo $O(\log n)$.

`insert`(*elem e*, *chiave k*)

Si crea nella foresta un nuovo albero binomiale B_0 costituito da un unico nodo a cui viene associato l'elemento e e la chiave k. Il problema è che, se un albero B_0 era già presente, si violerebbe la proprietà di unicità (per ogni i, esiste al più un albero B_i). Per ripristinare l'unicità, viene usata la procedura `ristruttura()`, mostrata in Figura 8.7, che fonde gli eventuali due doppioni B_0 in un albero più grande B_1. Poiché un altro B_1 poteva essere già presente nella foresta, la fusione viene ripetuta finchè non si hanno più duplicati. Per mantenere la proprietà di ordinamento a heap, ad ogni fusione si pone la radice con chiave più piccola come genitore della radice con chiave più grande. La procedura `ristruttura` effettua al più $O(\log n)$ fusioni, e ciascuna fusione può essere implementata in tempo costante, compreso l'aggiornamento della lista delle radici. Quindi, il tempo per l'inserimento è $O(\log n)$.

`deleteMin()`

Si cerca in tempo $O(\log n)$ l'albero B_h con radice a chiave minima. Rimuovendo la radice di B_h, esso si spezza in h alberi binomiali più piccoli $B_0, B_1, \ldots, B_{h-1}$, che vengono aggiunti alla foresta. Poiché questo potrebbe violare la proprietà di unicità, la foresta viene ristrutturata mediante la procedura `ristruttura` che fonde ripetutamente i doppioni. Si noti che in ciascuna iterazione di fusione esisteranno al più tre B_i: un B_i originariamente nella foresta, un B_i che era un sottoalbero di B_h, e un B_i creato dalla fusione all'iterazione precedente. Ne consegue che, come nel caso della `insert`, basta una sola fusione per ogni i per

non lasciare doppioni, e quindi anche qui `ristruttura` richiede al più tempo $O(\log n)$. Possiamo dunque concludere che il tempo per `deleteMin` è $O(\log n)$.

`decreaseKey`(*elem e, chiave d*)

L'operazione decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto mediante una chiamata `muoviAlto`(v) (vedi di Figura 8.4). Si noti che questa realizzazione richiede tempo $O(\log n)$ ed è praticamente identica a quella corrispondente vista per i d-heap in Paragrafo 8.1.

`delete`(*elem e*)

Questa operazione può essere realizzata in tempo totale $O(\log n)$ in termini delle operazioni `decreaseKey` e `deleteMin`. Infatti, si può effettuare prima l'operazione `decreaseKey`($e, -\infty$), dove $-\infty$ denota il più piccolo valore nell'universo delle chiavi, e poi `deleteMin`().

`increaseKey`(*elem e, chiave d*)

L'operazione può essere realizzata in tempo totale $O(\log n)$ in termini delle operazioni `delete` e `insert`. Infatti, se k è la chiave corrente dell'elemento *elem*, si può prima rimuovere l'elemento con `delete`(e), per poi reinserirlo con `insert` $(e, k + d)$.

`merge`(`CodaPriorita` c_1, `CodaPriorita` c_2) $\rightarrow$ `CodaPriorita`

Per fondere due heap binomiali c_1 e c_2, basta unire le loro liste delle radici. Poi, i duplicati sono rimossi dalla nuova lista richiamando la procedura `ristruttura`. Come per `deleteMin`, durante ciascuna iterazione di `ristruttura` esisteranno al più tre B_i, e quindi sarà richiesta una fusione per ciascun i. Di conseguenza, il tempo richiesto per la `merge` è $O(\log n)$.

Il seguente teorema riassume i tempi di esecuzione di ciascuna delle operazioni della classe `HeapBinomiale` vista in questo paragrafo.

Teorema 8.2 *Usando heap binomiali, è possibile realizzare tutte le operazioni del tipo di dato coda con priorità in tempo $O(\log n)$ nel caso peggiore.*

8.3 * Heap di Fibonacci

Gli heap di Fibonacci offrono probabilmente la più efficiente implementazione degli heap che sia nota. Essi rappresentano una versione rilassata degli heap binomiali e usano uno schema di update pigro (*lazy*). In questo paragrafo, otterremo gli heap di Fibonacci mediante due estensioni successive a partire dagli heap binomiali, rilassandone due proprietà principali:

1. *Rilassamento della proprietà di unicità*: accetteremo copie multiple dei B_i nella foresta. Questo ci permetterà di ottenere una struttura dati intermedia che chiameremo *heap binomiale rilassato* e che supporta le operazioni delle code con priorità con i seguenti tempi ammortizzati:
   - `insert`, `merge` e `findMin`: $O(1)$
   - `delete`, `deleteMin`, `decreaseKey` e `increaseKey`: $O(\log n)$
2. *Rilassamento della proprietà di struttura*: rinunceremo a mantenere i B_i nella foresta come veri alberi binomiali, accettando che possano avere meno nodi. Congiuntamente al punto 1. e quindi partendo dagli heap binomiali rilassati, otterremo gli *heap di Fibonacci*, dove anche il tempo della `decreaseKey` verrà ridotto a $O(1)$ ammortizzato.

Come vedremo nel Capitolo 13, ridurre il tempo dell'operazione `decreaseKey` a $O(1)$ ammortizzato avrà conseguenze importanti per le prestazioni di alcuni algoritmi che usano al loro interno code con priorità.

8.3.1 Heap binomiali rilassati

Un *heap binomiale rilassato* è semplicemente un heap binomiale che non gode della proprietà di unicità dei B_i:

Definizione 8.4 (Heap binomiale rilassato) *Un* heap binomiale rilassato *è una foresta di alberi B_i con le seguenti proprietà:*

1. Struttura*: ogni albero B_i nella foresta è un albero binomiale.*
2. Contenuto informativo*: ogni nodo v nella foresta contiene un elemento $elem(v)$ e una* chiave *$chiave(v)$ presa da un universo totalmente ordinato.*
3. Ordinamento a heap*: in ogni B_i, ciascun nodo v diverso dalla radice ha una chiave non inferiore a quella del suo genitore: $chiave(v) \geq chiave(padre(v))$.*

Una conseguenza importante del rilassamento sull'unicità è che il Lemma 8.3 degli heap binomiali non vale più per gli heap binomiali rilassati: infatti, il numero di alberi in un heap binomiale rilassato potrebbe essere addirittura $\Theta(n)$.

Realizzazione

Mostreremo ora che con pochissime modifiche rispetto a quanto già descritto per gli heap binomiali, possiamo realizzare una coda con priorità basata su heap binomiali rilassati con tempi di `insert` e `merge` ridotti a una costante. L'idea fondamentale è di approfittare del rilassamento sull'unicità per avere un atteggiamento più "pigro": ad esempio, perché ristrutturare subito la foresta dopo ogni `insert` quando potremmo farlo dopo? Potremmo mantenere facilmente un puntatore alla radice con chiave minima, e così saremmo comunque in grado di rispondere a `findMin` velocemente nonostante il numero di radici possa crescere

classe `HeapBinomialeRilassato` **estende** `HeapBinomiale`:
dati: $S(n) = O(n)$
foresta H ereditata, più un puntatore min alla radice con chiave minima.
operazioni:
`delete`(*elem e*) $T_{am}(n) = O(\log n)$
`increaseKey`(*elem e, chiave d*) $T_{am}(n) = O(\log n)$
ereditate.
`deleteMin()` $T_{am}(n) = O(\log n)$
chiama `deleteMin()` ereditata, e poi ricalcola min.
`decreaseKey`(*elem e, chiave d*) $T_{am}(n) = O(\log n)$
chiama `decreaseKey`(e, d) ereditata, e poi aggiorna min.
`findMin()` $\rightarrow$ *elem* $T_{am}(n) = O(1)$
restituisce $elem(min)$.
`insert`(*elem e, chiave k*) $T_{am}(n) = O(1)$
aggiunge ad H un nuovo B_0 con dati e e k, e aggiorna min.
`merge`(`CodaPri`. c_1, `CodaPri`. c_2) $\rightarrow$ `CodaPri`. $T_{am}(n) = O(1)$
unisce gli alberi in c_1 e c_2 in un nuovo heap binomiale c_3, e poi calcola min in c_3.

Figura 8.8 Coda con priorità `HeapBinomialeRilassato` come estensione della classe `HeapBinomiale` di Figura 8.6.

indefinitamente. Di fatto, avere pochi alberi ci serve veramente solo quando dobbiamo effettuare una `deleteMin`, perché non potremmo evitare di scandire tutte le radici per trovare il nuovo minimo. Quindi, potremmo ristrutturare la foresta esclusivamente durante la `deleteMin`.

Queste idee sono riassunte in Figura 8.8, dove mostriamo la struttura di una possibile classe `HeapBinomialeRilassato` derivata dalla classe base `HeapBinomiale` di Figura 8.6. L'unica cosa che un oggetto `HeapBinomialeRilassato` avrà in più di un oggetto `HeapBinomiale` sarà un puntatore alla radice con chiave minima nella foresta. Operazioni come `increaseKey` e `delete`, che erano già realizzate in termini di altre operazioni, potranno essere direttamente ereditate dalla classe base. Altre operazioni come `deleteMin` e `decreaseKey` potranno essere realizzate richiamando prima le corrispondenti operazioni ereditate, che faranno il grosso del lavoro, e aggiornando poi min. Notiamo che la `findMin` diventa immediata sfruttando l'informazione in min. Infine, `insert` e `merge` saranno uguali alle loro controparti in `HeapBinomiale`, salvo che non richiameranno `ristruttura`, e aggiorneranno min.

Analisi ammortizzata

Osserviamo che, posponendo il lavoro di ristrutturazione, la foresta porebbe contenere molti B_i duplicati, e quindi non possiamo più concludere che la procedura `ristruttura` richiamata da `deleteMin` (vedi Paragrafo 8.2) richieda solo tempo $O(\log n)$ nel caso peggiore. Useremo quindi un'analisi ammortizzata, basandoci sul metodo dei crediti descritto nel Capitolo 2. Un'osservazione cruciale è che ogni iterazione della procedura `ristruttura` riduce il numero di radici di 1 ponendo la redice di un albero come figlia della redice dell'altro. Se ora immaginiamo che su ogni redice sia depositata una moneta, l'intero costo della `ristruttura` potrebbe essere "pagato" usando le monete prese dalle radici rimosse. In base a questo ragionamento, `ristruttura` avrebbe costo ammortizzato zero. Affinché il ragionamento stia in piedi, dobbiamo però sovraccaricare il costo effettivo delle operazioni che creano le radici, sommandogli il costo delle monete che vanno depositate su di esse. In questo modo, su una sequenza di operazioni, il bilancio dei costi in termini di monete non andrà mai in passivo.

Più precisamente, useremo il seguente modello di costo ammortizzato, basato sul metodo dei crediti visto nel Paragrafo 2.7.1 del Capitolo 2:

1. *Assumiamo che, in ogni istante durante una sequenza di operazioni, su ogni radice dell'heap sia depositata una moneta.*
2. *Definiamo il costo ammortizzato di un'operazione $T_{am}(n)$ come:*

$$T_{am}(n) = T(n) + deposito(n) - prelievo(n),$$

dove $T(n)$ è il numero di passi effettivo che l'operazione compie nel caso peggiore, $deposito(n)$ è il numero di monete che deposita sulle radici, e $prelievo(n)$ è il numero di quelle che preleva dalle radici.

Analizziamo ora il costo ammortizzato $T_{am}(n)$ di ogni operazione della classe `HeapBinomialeRilassato` (si veda la Tabella 8.1 per quelle fondamentali). Iniziamo la nostra analisi dalla `deleteMin`, che è il caso più interessante.

`deleteMin`

Studiamo dapprima il costo della chiamata alla `deleteMin` ereditata. Osserviamo che, poiché in un albero binomiale $D(n) = O(\log n)$ (vedi Proprietà 8.1), la rimozione del nodo radice a chiave minima richiede tempo $O(\log n)$ e crea $O(\log n)$ nuove redici su cui vanno depositate altrettante monete. Quindi si ha $deposito(n) = O(\log n)$. Inoltre, `ristruttura` richiede tempo $O(\Delta + \log n)$, dove Δ è la differenza tra il namero di alberi nel heap binomiale rilassato prima e dopo la sua esecuzione. Infatti, ogni fusione diminuisce di uno il numero di radici (e quindi $prelievo(n) = \Delta$) e in base al Lemma 8.3 `ristruttura` può scandire al più $A(n) = \log n$ indici i per cui B_i non ha doppioni. Quindi, per la chiamata alla `deleteMin` ereditata si ha $T(n) = O(\Delta + \log n)$ e quindi $T_{am} = T(n) + deposito(n) - prelievo(n) = O(\Delta + 2\log n - \Delta) = O(\log n)$. Il

	$T(n)$	*deposito*(n)	*prelievo*(n)	$T_{am}(n)$
`insert`	$O(1)$	1	0	$O(1)$
`deleteMin`	$\Delta + A(n)$	$O(D(n))$	Δ	$O(A(n) + D(n))$
`decreaseKey`	$O(H(n))$	0	0	$O(H(n))$
`findMin`	$O(1)$	0	0	$O(1)$
`merge`	$O(1)$	0	0	$O(1)$

Tabella 8.1 Costi nel caso peggiore $T(n)$ e costi ammortizzati $T_{am}(n)$ delle operazioni fondamentali della classe `HeapBinomialeRilassato`. Δ è la differenza tra il numero di alberi nella foresta prima e dopo l'esecuzione di un'operazione, *deposito*(n) è il numero di monete che l'operazione deposita sulle radici della foresta quando le crea, mentre *preleva*(n) è il numero di quelle che consuma quando rimuove radici. $A(n)$ è il numero di alberi nella foresta dopo una ristrutturazione, $H(n)$ è la profondità degli alberi della foresta e $D(n)$ è il numero massimo di figli di ciascuna radice: poiché la nostra classe mantiene alberi binomiali, $A(n) = H(n) = D(n) = O(\log n)$.

ricalcolo della radice a chiave minima dopo la ristrutturazione richiede di scandire tutte le $O(\log n)$ radici, e quindi $T_{am}(n) = O(\log n)$ per l'intera `deleteMin`.

`insert`

L'operazione, il cui costo effettivo $T(n)$ è costante, crea una radice e non ne elimina nessuna. Quindi $T_{am}(n) = O(1) + 1 - 0 = O(1)$.

`decreaseKey`

Il costo effettivo dell'operazione è dominato dalla chiamata a `decreaseKey` ereditata, il cui tempo di esecuzione nel caso peggiore è $O(H(n))$, dove $H(n)$ è l'altezza degli alberi nella foresta. Poiché *deposito*(n) = *prelievo*$(n) = 0$, concludiamo che $T_{am} = T(n) = O(H(n))$. Nel caso di alberi binomiali, $H(n) = O(\log n)$ (vedi Proprietà 8.1), e quindi $T_{am}(n) = O(\log n)$.

`findMin` e `merge`

Le operazioni, il cui costo effettivo $T(n)$ è costante, non creano né eliminano radici. Quindi per entrambe $T_{am}(n) = O(1) + 0 - 0 = O(1)$.

`delete` e `increaseKey`

Poiché sono interamente realizzate richiamando `decreaseKey`, `deleteMin`, e `insert`, per entrambe abbiamo $T_{am}(n) = O(\log n)$.

Il seguente teorema riassume i tempi di esecuzione di ciascuna delle operazioni della classe `HeapBinomialeRilassato` vista in questo paragrafo.

Teorema 8.3 *Usando heap binomiali rilassati, è possibile realizzare le operazioni* `findMin`, `insert` *e* `merge` *in tempo* $O(1)$ *ammortizzato, e le operazioni* `delete`, `deleteMin`, `increaseKey` *e* `decreaseKey` *in tempo* $O(\log n)$ *ammortizzato.*

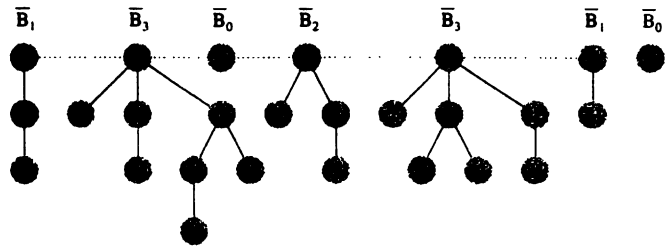


Figura 8.9 Esempio di heap di Fibonacci. Nei nodi sono mostrate solo le chiavi.

8.3.2 Heap di Fibonacci

Un *heap di Fibonacci* può essere pensato come un heap binomiale rilassato con un ulteriore rilassamento relativo alla proprietà di struttura dei suoi alberi:

Definizione 8.5 (Heap di Fibonacci) *Un* heap di Fibonacci *è una foresta di alberi* $\overline{B}_i$ *con le seguenti proprietà:*

1. Struttura: *ogni albero* $\overline{B}_i$ *nella foresta ha almeno* F_{i+2} *nodi, dove* F_{i+2} *è l'*$(i+2)$*-esimo numero di Fibonacci*[1] *e* i *è il grado (numero di figli) della radice. La proprietà vale anche per ogni sottoalbero di* $\overline{B}_i$.
2. Contenuto informativo: *ogni nodo* v *nella foresta contiene un elemento* $elem(v)$ *e una* chiave $chiave(v)$ *presa da un universo totalmente ordinato.*
3. Ordinamento a heap: *in ogni* $\overline{B}_i$, *ciascun nodo* v *diverso dalla radice ha una chiave non inferiore a quella del suo genitore:* $chiave(v) \geq chiave(padre(v))$.

Un esempio di heap di Fibonacci è mostrato in Figura 8.9. Parliamo di "rilassamento" della struttura poiché un albero $\overline{B}_i$ in un heap di Fibonacci potrebbe avere meno nodi, a parità di grado della radice i, del corrispondente albero binomiale B_i: in Tabella 8.2 confrontiamo il minimo numero di nodi che può avere un $\overline{B}_i$ con quello del corrispondente B_i, per vari valori di i. Come crescono i numeri di Fibonacci? Ricordiamo che $F_h = \Theta(\phi^h)$, dove $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ è la sezione aurea (vedi Appendice, Relazione 17.13). Questo implica che ogni albero $\overline{B}_i$ avrà un numero di nodi esponenziale nel grado i della radice, proprio come gli alberi binomiali B_i. Quindi in un heap di Fibonacci con n nodi, nonostante il rilassamento sulla struttura, i $\overline{B}_i$ continueranno ad avere radici con grado $i \leq D(n)$

[1] Ricordiamo che il primo numero di Fibonacci è $F_1 = 1$, il secondo è $F_2 = 1$, il terzo è $F_3 = 2$, il quarto è $F_4 = 3$, ecc.

Grado i della radice	*Minimo numero di nodi di un albero* $\overline{B}_i$ $(= F_{i+2})$	*Numero di nodi di un albero binomiale* B_i $(= 2^i)$
0	1	1
1	2	2
2	3	4
3	5	8
4	8	16
5	13	32
6	21	64
7	34	128

Tabella 8.2 Crescita del numero di nodi di $\overline{B}_i$ e di B_i all'aumentare di i.

logaritmico in n come i B_i: si ricordi che questo era cruciale per l'efficienza della `deleteMin` negli heap binomiali rilassati! Notiamo invece che i $\overline{B}_i$ potrebbero non avere più altezza $H(n)$ logaritmica, e questo potrebbe creare problemi per la `decreaseKey`. Riassumendo, possiamo scrivere:

Lemma 8.4 *Se n è il numero di nodi di un heap di Fibonacci, $D(n)$ è il massimo grado delle sue radici, $H(n)$ è la massima altezza di un suo albero, e $\phi = (1 + \sqrt{5})/2$ è la sezione aurea, valgono le seguenti relazioni:*

1. *Numero di nodi:* $n = \Omega(\phi^{D(n)})$
2. *Massimo grado delle radici:* $D(n) = O(\log_\phi n)$
3. *Altezza:* $H(n) = O(n)$

Vediamo ora come queste proprietà entrano in gioco nella realizzazione di una possibile classe `HeapFibonacci`.

Realizzazione

Poiché un heap di Fibonacci è semplicemente un heap binomiale rilassato che usa i $\overline{B}_i$ invece dei B_i, potremmo pensare di realizzare una classe `HeapFibonacci` come estensione della classe `HeapBinomialeRilassato` vista nel Paragrafo 8.3.1. A questo scopo, è naturale chiedersi quali operazioni possano essere ereditate dalla classe base così come sono, e quali invece necessitino cambiamenti per rimanere efficienti anche nel caso in cui si usino i $\overline{B}_i$ invece dei B_i.

Osserviamo innanzitutto che le operazioni `insert`, `merge` e `findMin`, che hanno a che fare solo con le radici, non risentono del rilassamento sulla struttura interna degli alberi, e possono essere ereditate così come sono. In base al Lemma 8.4, gli alberi in un heap di Fibonacci continuano ad avere radici con grado massimo $D(n)$ logaritmico in n come gli alberi binomiali, mentre l'altezza $H(n)$ potrebbe crescere indefinitamente. Questo ci fa pensare che anche la `deleteMin` degli heap binomiali rilassati, che ha costo ammortizzato

classe `HeapFibonacci` **estende** `HeapBinomialeRilassato`:
dati: $S(n) = O(n)$
foresta H e puntatore *min* ereditati, più un campo *lutto* per ogni nodo.

operazioni:
`insert`(*elem e*, *chiave k*) $T_{am}(n) = O(1)$
`merge`(`CodaPri`. c_1, `CodaPri`. c_2) $\rightarrow$ `CodaPri`. $T_{am}(n) = O(1)$
`findMin`() $\rightarrow$ *elem* $T_{am}(n) = O(1)$
`deleteMin`() $T_{am}(n) = O(\log n)$
`delete`(*elem e*) $T_{am}(n) = O(\log n)$
`increaseKey`(*elem e*, *chiave d*) $T_{am}(n) = O(\log n)$
ereditate senza modifiche.
`decreaseKey`(*elem e*, *chiave d*) $T_{am}(n) = O(1)$
decrementa di d la chiave del nodo v che contiene l'elemento e. Se la proprietà di ordinamento viene violata da v, invoca la procedura `staccaInCascata`(v).

operazioni ausiliarie:
`staccaInCascata`(v) $T_{am}(n) = O(1)$
vedi Figura 8.11.

Figura 8.10 Coda con priorità `HeapFibonacci` come estensione della classe `HeapBinomialRilassato`.

$T_{am} = O(D(n) + \log n)$, potrebbe funzionare bene per gli heap di Fibonacci. Purtroppo invece, usare la stessa `decreaseKey` degli heap binomiali rilassati, che ha costo ammortizzato $T_{am} = O(H(n))$, ci darebbe un costo proibitivo $O(n)$.

Nella nostra realizzazione, schematicamente descritta in Figura 8.10, ereditiamo quindi tutte le operazioni dalla classe `HeapBinomialeRilassato`, tranne la `decreaseKey`, in cui si concentra tutta la difficoltà realizzativa aggiuntiva degli heap di Fibonacci. Oltre alla foresta e al puntatore *min* ereditati dalla classe base, un oggetto `HeapFibonacci` manterrà per ogni nodo v nella foresta un campo booleano aggiuntivo che chiameremo *lutto*(v) per i motivi che vedremo a breve. Descriviamo ora come la `decreaseKey` può essere realizzata.

`decreaseKey`(*elem e*, *chiave d*)

L'operazione decrementa dapprima di d la chiave del nodo v che contiene l'elemento e. Per quanto detto sopra, se il valore risultante della chiave di v dovesse violare la proprietà di ordinamento a heap, non possiamo permetterci di far risalire il nodo nell'albero che lo contiene come abbiamo fatto finora in questo capitolo, perché questo potrebbe costarci $O(n)$.

Quello che invece facciamo è semplicemente staccare tutto il sottoalbero radicato in v dal genitore di v e aggiungerlo alla foresta, rendendo quindi v una nuova radice e ripristinando così la proprietà di ordinamento. Il problema è che, così facendo, l'albero $\overline{B}_i$ da cui stacchiamo v perde dei nodi; se il genitore di v

```
procedura staccaInCascata(v)
  u = padre(v)
  togli v dalla lista dei figli di u e aggiungi v alla lista di radici dell'heap
  lutto(v) = false
  if ( u non è una radice ) then
     if ( lutto(u) ) then staccaInCascata(u)
     else lutto(u) = true
```

Figura 8.11 Procedura `staccaInCascata` per il ripristino delle proprietà di struttura e ordinamento in un heap di Fibonacci. La procedura taglia l'arco tra v e il suo genitore u. Se u non è una radice e aveva già perso un figlio, stacca ricorsivamente anche u in modo che ogni nodo non-radice perda al più un figlio.

non è la radice, potremmo quindi rischiare di andare sotto il limite di nodi minimo dato da F_{i+2}, perdendo la proprietà di struttura degli heap di Fibonacci. L'idea fondamentale per risolvere questo problema, di cui discuteremo in seguito l'efficacia, è di accettare che ogni nodo non-radice possa perdere al più un figlio.

Invariante 8.1 *Ogni nodo nella foresta può perdere al più un figlio, a meno che non sia una radice.*

Per mantenere questa invariante quando si debba togliere un ulteriore figlio ad un nodo già in lutto, basta semplicemente trasformare quel nodo in radice, staccandolo a sua volta in cascata dal proprio genitore. È qui che il campo *lutto* entra in gioco: ogni volta che un nodo non radice u perde un figlio, poniamo il campo $lutto(u)$ a $true$ per ricordarci di questo triste evento. Se il nodo u era già in lutto, stacchiamo ricorsivamente u dal prorio genitore e poniamo $lutto(u)$ a $false$. Questo procedimento è mostrato in Figura 8.11.

Analisi di correttezza

Verifichiamo ora una foresta di alberi mantenuta sotto l'Invariante 8.1 con le operazioni viste soddisfa la proprietà di struttura degli heap di Fibonacci data in Definizione 8.5.

Teorema 8.4 *Sia v un qualunque nodo in una foresta mantenuta con le operazioni di Figura 8.10, e sia $size(v)$ la dimensione del sottoalbero radicato in v. Allora $size(v) \geq F_{h+2}$, dove h è il grado di v.*

Dimostrazione. Sia v un nodo di grado h nella foresta e siano $u_1, u_2, \ldots, u_h$ i suoi figli. dal più vecchio al più giovane (ovvero u_1 è diventato figlio di v prima di u_2, e così via).

Come primo ingrediente della dimostrazione, mostriamo che il grado di ogni figlio u_i è almeno $i-2$. Quando u_i è stato reso figlio di v (e solo una fusione durante la `deleteMin` può averlo fatto) v doveva essere una radice. A quel tempo, $u_1, \ldots, u_{i-1}$ erano già figli di v, e quindi v doveva avere almeno grado $i-1$. Poiché quando due alberi sono fusi le loro due radici devono avere necessariamente lo stesso grado, u_i doveva avere anch'esso grado almeno $i-1$. Da allora, per l'Invariante 8.1, u_i può aver perso solo un figlio, senza guadagnarne nessuno non essendo una radice. E quindi ora u_i ha necessariamente grado almeno $i-2$. Vediamo ora come sfruttare questa proprietà.

Sia m_h il più piccolo numero di nodi che può avere un albero con radice di grado h nella foresta. In base a questa definizione, $size(v) \geq m_h$; inoltre, poiché come visto u_i ha necessariamente grado $j \geq i-2$, e per come gli alberi sono costruiti $m_{i-2} \leq m_{i-1} \leq \cdots \leq m_j$, allora $size(u_i) \geq m_j \geq m_{i-2}$. Quanti nodi almeno deve avere l'albero radicato in v? Contando 1 per v, 1 per u_1 e almeno m_{i-2} per ogni u_i, $i \geq 2$, otteniamo:

$$m_h \geq 1 + 1 + m_0 + m_1 + \cdots + m_{h-2} = 2 + \sum_{i=2}^{h} m_{i-2}. \qquad (8.1)$$

Per completare la prova, mostriamo per induzione su h che $m_h \geq F_{h+2}$. Il passo base è verificato: infatti per $h = 0$, $m_0 = 1 = F_2$, e per $h = 1$, $m_1 = 2 = F_3$. Assumiamo ora per ipotesi induttiva che $m_i \geq F_{i+2}$ per ogni $i < h$ e partiamo dalla disuguaglianza (8.1): usando l'ipotesi induttiva sugli m_{i-2}, ricordando che $F_1 = 1$ e usando il Lemma 17.5 in Appendice, otteniamo infine:

$$m_h \geq 2 + \sum_{i=2}^{h} m_{i-2} \geq 2 + \sum_{i=2}^{h} F_i = 1 + \sum_{i=1}^{h} F_i = F_{h+2}.$$

Poiché come osservato $size(v) \geq m_h$, questo completa la dimostrazione. □

Analisi ammortizzata

Calcoliamo ora il costo temporale ammortizzato di ciascuna operazione della classe `HeapFibonacci`.

`insert`, `merge` e `findMin`

Queste operazioni ereditate hanno a che fare solo con le radici e quindi mantengono lo stesso costo $T_{am}(n) = O(1)$ che avevano nella classe `HeapBinomialeRilassato`.

`deleteMin`

Come riportato in Tabella 8.1, il costo dell'operazione ereditata è $O(D(n) + \log n)$, dove $D(n)$ è il massimo grado della radice. Poiché per il Lemma 8.4 degli

heap di Fibonacci $D(n) = O(\log_\phi n) = O(\log n)$, concludiamo che $T_{am}(n) = O(\log n)$.

`decreaseKey`

Il costo dell'operazione è dominato dalla chiamata a `staccaInCascata`. Per studiarne il costo ammortizzato, facciamo due osservazioni chiave. Durante ogni esecuzione della `decreaseKey`:

1. *al più un nodo viene messo in lutto:*
2. *ogni chiamata ricorsiva di* `staccaInCascata`*, tranne al più la prima, trasforma un nodo in lutto in una nuova radice.*

Se ora immaginiamo che la `decreaseKey` depositi due monete sul nodo che mette in lutto, queste pagheranno anche il costo del trasformare quel nodo in radice (togliendolo dal lutto) al momento di una successiva `decreaseKey`: infatti, una moneta pagherà il costo effettivo della chiamata ricorsiva di `staccaInCascata` che processerà il nodo in lutto, e l'altra moneta verrà semplicemente lasciata sul nodo poiché esso diventa una radice. Assumiamo quindi che:

In ogni istante durante una sequenza di operazioni, su ogni radice dell'heap è depositata una moneta e su ogni nodo in lutto ve ne sono due.

Siamo ora pronti a calcolare $T_{am}(n) = T(n) + deposito(n) - prelievo(n)$ per `decreaseKey`. Per quanto detto, tutte le chiamate ricorsive della procedura `staccaInCascata`, tranne la prima, avranno costo ammortizzato $T'_{am}(n) = 0$, poiché sia il costo effettivo che le monete messe sulle radici sono interamente pagati con le monete trovate sui nodi in lutto processati. Quanto alla prima chiamata di `staccaInCascata`, essa ha costo effettivo costante e deposita al più tre monete (due per l'eventuale antenato che verrà messo a lutto, più una per la radice che crea), e quindi ha costo ammortizzato $T''_{am}(n) = O(1) + 3 - 0$. Possiamo infine concludere che $T_{am}(n) = T'_{am}(n) + T''_{am}(n) = O(1)$.

`delete` e `increaseKey`

Poiché sono interamente realizzate richiamando `decreaseKey`, `deleteMin`, e `insert`, per entrambe abbiamo $T_{am}(n) = O(\log n)$.

Il seguente teorema riassume i tempi di esecuzione di ciascuna delle operazioni della classe `HeapFibonacci` vista in questo paragrafo.

Teorema 8.5 *Usando heap di Fibonacci, è possibile realizzare le operazioni* `insert`, `findMin`, `decreaseKey` *e* `merge` *in tempo* $O(1)$ *ammortizzato, e le operazioni* `delete`, `deleteMin` *e* `increaseKey` *in tempo* $O(\log n)$ *ammortizzato.*

8.4 Problemi

Problema 8.1 Dimostrare che un albero binomiale B_h gode delle seguenti proprietà:

1. Numero di nodi ($|B_h|$): $n = 2^h$.
2. Grado della radice: $D(n) = \log_2 n$
3. Altezza: $H(n) = h = \log_2 n$.

Problema 8.2 (*) Dimostrare che gli alberi binomiali soddisfano le seguenti proprietà:

(a) In un albero binomiale con n nodi vi sono $\binom{n}{h}$ nodi a profondità h.
(b) Un albero binomiale B_h può essere ottenuto da un albero binomiale B_q, $q \leq h$, rimpiazzando ciascun nodo di B_q con un albero binomiale B_{h-q}. I figli del nodo rimpiazzato diventano figli della radice dell'albero rimpiazzante.
(c) Un albero binomiale B_h di profondità h contiene un nodo con h figli, che sono radici di alberi binomiali $B_0, \ldots, B_{h-1}$.

Problema 8.3 Dimostrare che il costo di ogni operazione `insert` in un heap binomiale è $O(1)$ ammortizzato, se non si effettua nessuna cancellazione.

Problema 8.4 Supponiamo di sostituire l'Invariante 8.1 vista per gli heap di Fibonacci con la seguente: "*Ogni nodo nella foresta può perdere al più due figli, a meno che non sia una radice*". Quali proprietà strutturali avrebbero gli alberi $\overline{B}_i$ della foresta mantenuti sotto questa invariante? Continuerebbe a valere la disuguaglianza $|\overline{B}_i| \geq F_{i+2}$? Il costo temporale ammortizzato delle varie operazioni peggiorerebbe?

Problema 8.5 Esibire una sequenza di operazioni che crei in un heap di Fibonacci un albero $\overline{B}_1$ arbitrariamente grande con struttura a catena, cioè con tutti i nodi interni di grado 1.

Problema 8.6 Mostrare che, scegliendo le priorità degli elementi opportunamente, è possibile fare in modo che una coda con priorità si comporti come una coda o come una pila (vedi il Paragrafo 3.2 del Capitolo 3).

8.5 Sommario

In questo capitolo abbiamo studiato come mantenere efficientemente il minimo (o il massimo) valore in una collezione di chiavi soggetta a operazioni di aggiornamento. Abbiamo definito le operazioni di interesse mediante il tipo di dato `CodaPriorita` e ne abbiamo mostrato quattro realizzazioni ottenute mediante raffinamenti successivi.

	DHeap	HeapBin.	HeapBin.Ril.	HeapFib.
findMin	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
insert	$O(\log_d n)$	$O(\log n)$	$O(1)$	$O(1)$
delete	$O(d \log_d n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
deleteMin	$O(d \log_d n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
increaseKey	$O(d \log_d n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log_d n)$	$O(\log n)$	$O(\log n)$	$O(1)$
merge	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$

Tabella 8.3 Tabella riassuntiva dei tempi per operazione ottenibili mediante le realizzazioni viste in questo capitolo. L'analisi per `DHeap` e `HeapBinomiale` è effettuata nel caso peggiore, mentre quella per `HeapBinomialeRilassato` e `HeapFibonacci` è ammortizzata.

La prima realizzazione (`DHeap`) mantiene la collezione di chiavi mediante un albero con grado d costante e profondità logaritmica in cui ogni nodo contiene un valore maggiore o uguale a quello dei figli. Con questa organizzazione dei dati, il minimo è sempre contenuto nella radice e tutte le operazioni di aggiornamento possono essere implementate in modo che richiedano tempo logaritmico nel caso peggiore.

La seconda realizzazione (`HeapBinomiale`) mantiene una foresta con $\log n$ alberi, piuttosto che un solo albero. Dopo ogni aggiornamento, come ad esempio l'inserimento o la cancellazione di una chiave, la foresta viene ristrutturata in modo da mantenere un numero logaritmico di alberi. Questa organizzazione consente di unire efficientemente due code con priorità in una, operazione non supportata dai `DHeap`. Data la particolare struttura degli alberi nella foresta (*alberi binomiali*) ogni ristrutturazione richiede tempo logaritmico nel caso peggiore.

Rinunciando a ristrutturare la foresta dopo ogni inserimento o fusione, e mantenendo un riferimento alla radice dell'albero che contiene il minimo, otteniamo il `HeapBinomialeRilassato`, in cui il tempo per `insert` e `merge` scende a costante ammortizzato, mentre il numero di alberi può crescere indefinitamente.

L'ultima realizzazione (`HeapFibonacci`), è fra le più efficienti che siano note. Essa permette di supportare anche l'operazione di decremento di una chiave in tempo costante ammortizzato. Come vedremo nel Capitolo 13, questo miglioramento ha conseguenze molto importanti per l'efficienza di alcuni algoritmi fondamentali che usano code con priorità. La classe `HeapFibonacci` è ottenuta a partire dalla classe `HeapBinomialeRilassato` mediante un ulteriore rilassamento sulla struttura degli alberi della foresta. Il nome `HeapFibonacci` deriva dal fatto che ogni albero con radice di grado h contiene almeno F_{h+2} nodi, dove F_{h+2} è l'$(h+2)$-esimo numero di Fibonacci.

Un quadro riassuntivo dei tempi per operazione ottenibili con le realizzazioni viste in questo capitolo è riportato in Tabella 8.3.

8.6 Note bibliografiche

La struttura dati heap e la sua applicazione alla realizzazione di code con priorità appaiono già nell'articolo di John Williams [8] del 1964 che descrive l'algoritmo di ordinamento `heapSort`. Gli heap binomiali sono stati inventati nel 1978 da Jean Vuillemin [7], mentre Mark Brown ne ha studiato le proprietà [1]. Gli heap di Fibonacci sono stati invece proposti da Michael Fredman e Robert Tarjan nel 1987 [4]. Come alternativa agli heap di Fibonacci, nel 1988 James Driscoll, Harold Gabow, Ruth Shrairman e Robert Tarjan [2] hanno introdotto un tipo diverso di heap chiamato "heap rilassato" che garantisce gli stessi tempi per operazione degli heap di Fibonacci. Tuttavia, i tempi ottenuti sono nel caso pessimo invece che ammortizzati.

Nel caso speciale di valori interi, è possibile ottenere code con priorità con prestazioni migliori. Ad esempio, utilizzando una struttura dati inventata da Peter Van Emde Boas [3] è possibile realizzare tutte le operazioni di una coda con priorità in tempo $O(\log \log C)$, dove C è il massimo valore intero di una chiave. Per una rassegna dei migliori risultati noti per code con priorità a valori interi, si può fare riferimento ai lavori di Rajeev Raman [5] e Mikkel Thorup [6].

Riferimenti bibliografici

[1] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[2] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[3] 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*, pages 75–84. IEEE Computer Society, 1975.

[4] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[5] Rajeev Raman. Priority Queues: Small, Monotone and Trans-dichotomous. Proceedings of the Fourth Annual European Symposium on Algorithms (ESA'96), 1996.

[6] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA

[7] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[8] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

9

Union-find

Union-find and the Ackermann inverse,
"Exercise left for reader", please, be terse,
wken I don't understand, it aii seems so grand
as we dance to tke aigorithms tango.

(Bridget Spitznagel)

In questo capitolo introdurremo le strutture dati *union-find*, che oltre ad avere importanza per le tecniche algoritmiche coinvolte, e per la loro analisi raffinata, verranno anche utilizzate nel seguito del libro, come ad esempio nel Capitolo 12. Supponiamo di avere una collezione S di n elementi distinti. Per fissare le idee, supponiamo che gli n elementi distinti siano i numeri interi da 1 a n. Inizialmente, tali elementi sono organizzati in n insiemi disgiunti: ogni insieme contiene esattamente un elemento. In maggiore dettaglio, avremo gli n insiemi disgiunti $\{1\}, \{2\}, \ldots, \{n\}$. Ogni insieme ha un nome: senza perdita di generalità, assumiamo che inizialmente il nome dell'insieme $\{i\}$ sia i, $1 \leq i \leq n$. Nel problema *union-find* desideriamo mantenere una collezione di insiemi disgiunti durante una sequenza di operazioni del seguente tipo:

`union`(A, B):	Unisci gli insiemi A e B in un unico insieme. Il nome dell'insieme ottenuto è A. Questa operazione distrugge i vecchi insiemi A e B.
`find`(x):	Dato un elemento, restituisci il nome dell'insieme che lo contiene.

Notiamo che su n elementi possono essere eseguite al più $(n-1)$ operazioni `union`: infatti, dopo $(n-1)$ operazioni `union`, otterremo un unico insieme contentente tutti gli n elementi.

Esempio 9.1 Supponiamo di avere 6 elementi ($n = 6$). Inizialmente abbiamo gli insiemi disgiunti: $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}$, e $\{6\}$. La Figura 9.1 mostra gli effetti di una sequenza di operazioni su questa configurazione iniziale. □

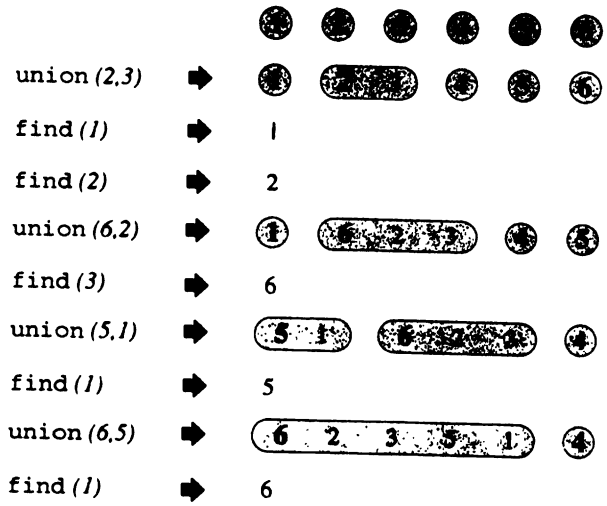


Figura 9.1 Effetti della sequenza di operazioni `union` e `find` definita nell'Esempio 9.1. Nella figura, il nome di un insieme viene indicato in grassetto.

Fino a questo punto abbiamo assunto che n fosse noto *a priori*. Definiamo adesso il problema *union-find* nel caso più generale, in cui il numero totale di elementi non è noto a priori. Inizialmente non ci sono né insiemi né elementi, ed i nuovi elementi sono introdotti dall'operazione:

`makeSet`(x): Crea un nuovo insieme $\{x\}$, contenente un nuovo elemento x. Il nome dell'insieme $\{x\}$ è x.

Diremo quindi che il problema *union-find* consiste nel mantenere una collezione di insiemi disgiunti durante una qualsiasi sequenza di operazioni `union`, `find` e `makeSet` a partire dall'insieme vuoto. Nel resto del capitolo, supporremo che vengono eseguite n operazioni `makeSet` ed m operazioni `find`. Dato che n `makeSet` creano esattamente n elementi, non serve specificare il numero di `union`, dato che sarà sicuramente limitato superiormente da $(n-1)$ (si veda il Problema 9.1). Il tipo di dato `UnionFind` è descritto schematicamente in Figura 9.2.

9.1 Approcci elementari al problema union-find

Prima di presentare gli algoritmi più efficienti noti per la risoluzione del problema union-find, descriveremo preliminarmente alcuni algoritmi elementari che ci saranno utili a scopo introduttivo. In particolare, nel Paragrafo 9.1.1 esamineremo algoritmi di tipo *QuickFind*, ovvero algoritmi che eseguono rapidamente le

tipo `UnionFind`:
dati:
una collezione di insiemi disgiunti di elementi *elem*; ogni insieme ha un nome *name*.
operazioni:
`makeSet`(*elem e*)
crea un nuovo insieme contenente unicamente l'elemento e; il nome dell'insieme $\{e\}$ è e.
`union`(*name a*, *name b*)
restituisce un nuovo insieme ottenuto dall'unione degli insiemi di nome a e b; il nome del nuovo insieme è a; i vecchi insiemi di nome a e b non sono più disponibili dopo l'operazione.
`find`(*elem e*) $\rightarrow$ *name*
restituisce il nome dell'insieme contenente l'elemento e.

Figura 9.2 Dati e operazioni di una struttura `UnionFind`.

operazioni `find`, a scapito dell'efficienza delle operazioni `union`. Nel Paragrafo 9.1.2 esamineremo invece algoritmi di tipo *QuickUnion*, che eseguono rapidamente le operazioni `union`, a scapito dell'efficienza delle operazioni `find`. Soluzioni in cui entrambe le operazioni di `union` e `find` sono eseguite efficientemente verranno invece presentate nei Paragrafi 9.2 e 9.3.

Nel resto del capitolo, rappresenteremo gli insiemi disgiunti tramite strutture ad albero così come sono state descritte nel Capitolo 3. In maggior dettaglio, rappresenteremo ogni insieme mediante un albero radicato: i nodi dell'albero corrisponderanno agli elementi dell'insieme, mentre la radice conterrà anche l'informazione relativa al nome dell'insieme. Gli archi dell'albero saranno semplicemente rappresentati con puntatori. In tale scenario, la collezione di insiemi disgiunti corrisponderà ad una foresta di alberi, ed avremo esattamente un albero per ciascun insieme.

9.1.1 Algoritmi di tipo QuickFind

In algoritmi di tipo *QuickFind* utilizzeremo alberi di altezza uno per rappresentare gli insiemi disgiunti. Gli elementi dell'insieme sono rappresentati dalle foglie dell'albero, mentre la radice di ogni albero conterrà semplicemente il nome dell'insieme. Denoteremo queste strutture dati come *alberi QuickFind*. L'implementazione delle operazioni `makeSet`, `union` e `find` con alberi QuickFind è definita in Figura 9.3 ed illustrata graficamente nella Figura 9.4.

Nel seguente teorema caratterizzeremo il tempo di esecuzione delle operazioni `makeSet`, `union` e `find` nella rappresentazione mediante alberi QuickFind.

classe `QuickFind` **implementa** `UnionFind`:
dati: $S(n) = O(n)$
una collezione di insiemi disgiunti di elementi *elem*; ogni insieme ha un nome *name*.

operazioni:
`makeSet`(*elem e*) $T(n) = O(1)$
crea un nuovo albero, composto da due nodi: una radice ed un unico figlio (foglia). Memorizza e sia nella foglia dell'albero che come nome nella radice.

`union`(*name a*, *name b*) $T(n) = O(n)$
considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Sostituisce tutti i puntatori dalle foglie di B alla radice di B con puntatori alla radice di A. Cancella la vecchia radice di B.

`find`(*elem e*) $\rightarrow$ *name* $T(n) = O(1)$
accede al nodo x corrispondente all'elemento e. Da tale nodo segue il puntatore al padre, che è la radice dell'albero, e restituisce il nome memorizzato in tale radice.

Figura 9.3 Realizzazione delle operazioni `makeSet`, `union` e `find` mediante alberi QuickFind.

Teorema 9.1 *Alberi QuickFind sono in grado di supportare operazioni* `makeSet` *e* `find` *in tempo costante. Una* `union` *può richiedere nel caso peggiore tempo* $O(n)$, *dove* n *è il numero di operazioni* `makeSet`. *L'occupazione di memoria è* $O(n)$.

Dimostrazione. Ogni operazione `makeSet` richiede la creazione di un albero con due nodi, e può essere eseguita in tempo $O(1)$. Per eseguire una `find`, è sufficiente seguire un puntatore da un nodo (foglia) al padre (radice), e restituire il valore memorizzato nella radice: anche questa operazione può essere eseguita in tempo $O(1)$. Il tempo richiesto da una `union` è proporzionale alla cardinalità dell'insieme B: nel caso peggiore, questa può essere $\Omega(n)$, dove n è il numero totale di elementi presenti (ovvero il numero di operazioni `makeSet` eseguite).

Per calcolare l'occupazione di memoria degli alberi QuickFind, osserviamo che un'operazione `makeSet` introduce esattamente due nodi ed un arco, mentre un'operazione `find` non altera la struttura degli alberi. Infine, un'operazione `union`(A, B) distrugge e crea lo stesso numero di archi, ed inoltre distrugge la vecchia radice dell'albero corrispondente all'insieme B. Ad ogni istante, il numero totale di nodi è pertanto $O(n)$, dove n è il numero totale di operazioni `makeSet` effettuate fino a quel momento, mentre il numero totale di archi è al più n. L'occupazione totale di memoria dalla struttura dati è pertanto $O(n)$. □

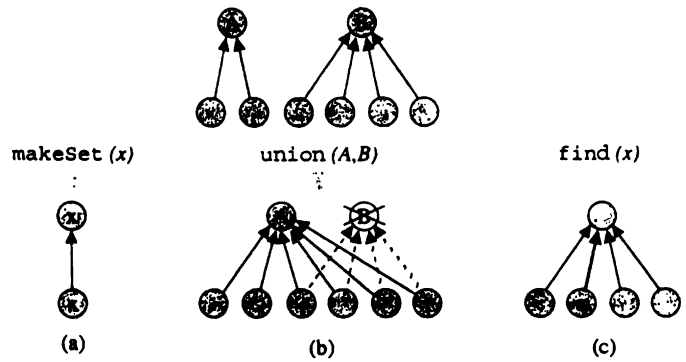


Figura 9.4 Esempio di `makeSet`, `union` e `find` su alberi QuickFind

9.1.2 Algoritmi di tipo QuickUnion

In algoritmi di tipo *QuickUnion* utilizzeremo ancora alberi per rappresentare gli insiemi disgiunti. A differenza degli alberi QuickFind, questa volta consentiremo agli alberi di avere un'altezza maggiore di uno. Similmente agli alberi QuickFind, il nome di ogni insieme sarà contenuto nella radice dell'albero corrispondente. Denoteremo queste strutture dati come *alberi QuickUnion*. L'implementazione di operazioni `makeSet`, `union` e `find` con alberi QuickUnion è definita in Figura 9.5 ed illustrata graficamente nella Figura 9.6. Osserviamo che, a differenza degli alberi QuickFind, anche le radici di alberi QuickUnion corrispondono ad elementi degli insiemi disgiunti. La radice di un albero QuickUnion memorizzerà quindi sia l'informazione relativa ad un elemento dell'insieme che il nome dell'insieme.

Nell'Esempio 9.2 è illustrato come sia possibile, con opportune sequenze di union, costruire alberi QuickUnion di altezza $O(n)$, dove n è il numero totale di elementi presenti (ovvero il numero di operazioni `makeSet` eseguite).

Esempio 9.2 Consideriamo la sequenza di operazioni `makeSet` e `union` generata nel modo seguente:

for $i = 1$ **to** n **do** `makeSet`(i)
for $i = n - 1$ **downto** 1 **do** `union`$(i, i + 1)$

L'albero QuickUnion costruito da tale sequenza ha altezza $\Omega(n)$, come può facilmente desumersi dall'esame della Figura 9.7. □

Nel seguente teorema caratterizzeremo il tempo di esecuzione delle operazioni `makeSet`, `union` e `find` su alberi QuickUnion.

Teorema 9.2 *Alberi QuickUnion sono in grado di supportare operazioni* `makeSet` *e* `union` *in tempo costante. Una* `find` *può richiedere nel caso peggiore tempo*

classe `QuickUnion` **implementa** `UnionFind`:
dati: $S(n) = O(n)$
una collezione di insiemi disgiunti di elementi *elem*; ogni insieme ha un nome *name*.
operazioni:
`makeSet`(*elem e*) $T(n) = O(1)$
crea un nuovo albero, composto da un unico nodo x. Memorizza e in tale nodo, sia come valore che come nome del nodo.
`union`(*name a*, *name b*) $T(n) = O(1)$
considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Rende la radice di B figlia della radice di A, introducendo un puntatore dalla radice di B alla radice di A.
`find`(*elem e*) $\rightarrow$ *name* $T(n) = O(n)$
accede al nodo x corrispondente all'elemento e. Partendo da tale nodo, segue ripetutamente i puntatori al padre fino a raggiungere la radice dell'albero. Restituisce il nome memorizzato in tale radice.

Figura 9.5 Realizzazione delle operazioni `makeSet`, `union` e `find` mediante alberi QuickUnion.

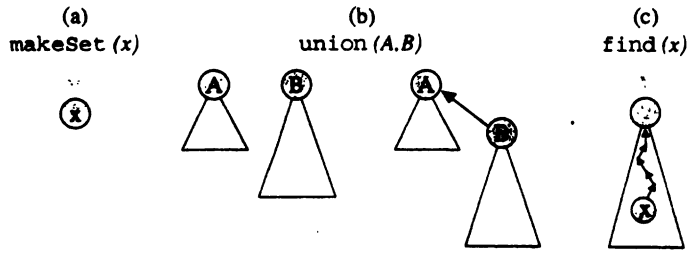


Figura 9.6 Esempio di `makeSet`, `union` e `find` su alberi QuickUnion

$O(n)$, *dove n è il numero di operazioni* `makeSet`. *L'occupazione di memoria è* $O(n)$.

Dimostrazione. Ogni operazione `makeSet` richiede la creazione di un albero con un solo nodo, e può essere eseguita in tempo $O(1)$. Per eseguire una `union`(A, B), è sufficiente inserire un puntatore dalla radice di B alla radice di A: anche questa operazione può essere semplicemente eseguita in tempo $O(1)$. Il tempo richiesto da una `find`(x) è proporzionale alla distanza dal nodo x alla radice dell'albero che lo contiene. Come evidenziato nell'Esempio 9.2, nel caso peggiore questa può essere $O(n)$, dove n è il numero totale di elementi presenti (ovvero il numero di operazioni `makeSet` eseguite).

Per calcolare l'occupazione di memoria degli alberi QuickUnion, è sufficiente osservare che i nodi possono essere introdotti solo da operazioni `makeSet` (un nodo per ogni `makeSet`), mentre gli archi possono essere introdotti solo da operazioni `union` (un arco per ogni `union`). Il numero totale di nodi è pertanto n (numero totale di operazioni `makeSet` effettuate), mentre il numero totale di archi è al più $(n - 1)$ (numero totale di operazioni `union` effettuate). □

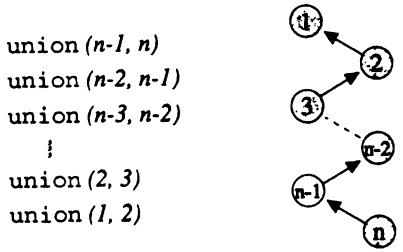


Figura 9.7 Sequenze di union possono costruire alberi QuickUnion di altezza $\Omega(n)$.

	`makeSet`	`union`	`find`
`QuickFind`	$O(1)$	$O(n)$	$O(1)$
`QuickUnion`	$O(1)$	$O(1)$	$O(n)$

Tabella 9.1 Tempi di esecuzione ottenibili con algoritmi di tipo QuickFind e di tipo QuickUnion.

La Tabella 9.1 riassume i tempi di esecuzione ottenibili con algoritmi di tipo QuickFind e QuickUnion. Nei prossimi paragrafi vedremo come sia possibile ottenere algoritmi più efficienti per la risoluzione del problema union-find.

9.2 Euristiche di bilanciamento nell'operazione union

In questo paragrafo vedremo algoritmi più efficienti degli algoritmi descritti in precedenza. In particolare, nel Paragrafo 9.2.1 vedremo come riuscire a ridurre, almeno in senso ammortizzato, il tempo di esecuzione delle `union` in algoritmi di tipo QuickFind, mentre nel Paragrafo 9.2.2 vedremo come riuscire a ridurre il tempo di esecuzione delle `find` in algoritmi di tipo QuickUnion.

9.2.1 Bilanciamento per algoritmi di tipo QuickFind

Se vogliamo migliorare le prestazioni degli alberi QuickFind descritte nel Paragrafo 9.1.1, dobbiamo concentrarci sull'operazione più costosa, che è indubbiamente

la union: in base all'analisi effettuata nel Teorema 9.1, infatti, la union presenta un tempo di esecuzione pari a $O(n)$ nel caso peggiore.

Osserviamo innanzitutto che l'operazione union viene implementata senza molta flessibilità: infatti, nel Paragrafo 9.1.1 abbiamo effettuato una union(A, B) spostando sempre tutti i nodi dell'albero corrispondente all'insieme B, in modo che la loro nuova radice sia la radice dell'insieme A. Questo può essere particolarmente oneroso, soprattutto in quei casi in cui l'insieme B ha cardinalità maggiore dell'insieme A (ovvero $|A| < |B|$). In tal caso sembrerebbe invece più efficiente spostare i nodi dell'albero corrispondente all'insieme A. Notiamo che così facendo rischieremmo di non garantire la proprietà che il nome del nuovo insieme $A \cup B$ sia A: per assicurare questo, è però sufficiente memorizzare A nella nuova radice.

Ci riferiremo a questo algoritmo come un *algoritmo di tipo QuickFind con bilanciamento sulle union*, e agli alberi ottenuti in questo modo come *alberi QuickFind con bilanciamento sulle union* oppure più semplicemente come *alberi QuickFind bilanciati*. Per ogni insieme disgiunto, utilizzeremo un campo *size* per memorizzarne la dimensione. Tale informazione sarà memorizzata nella radice dell'albero corrispondente. L'implementazione delle operazioni makeSet, union e find con alberi QuickFind bilanciati è definita in Figura 9.8 ed illustrata graficamente nella Figura 9.9.

La seguente proprietà è una conseguenza immediata del bilanciamento che abbiamo introdotto sulle union.

Lemma 9.1 *Ogni volta che una foglia di un albero QuickFind bilanciato acquista un nuovo padre a causa di un'operazione* union, *dopo la* union *tale foglia farà parte di un insieme che è grande almeno il doppio dell'insieme di cui faceva parte prima della* union.

Dimostrazione. Senza perdita di generalità assumiamo che l'operazione sia una union(A, B), con size$(A) \geq$ size(B). A causa di tale operazione, verranno spostate le foglie di B. Sia ρ una qualsiasi foglia di B prima di effettuare l'operazione. Prima della union(A, B), la foglia ρ si trovava in un insieme di cardinalità size(B). Dopo la union(A, B), la foglia ρ avrà una nuova radice, e si troverà in un insieme di cardinalità size(A) + size$(B) \geq$ size(B) + size$(B) = 2 \cdot$ size(B). □

Siamo ora in grado di dimostrare che il bilanciamento introdotto sulle union consente di ottenere prestazioni migliori di quelle ottenute nel Paragrafo 9.1.1, almeno in senso ammortizzato.

Teorema 9.3 *Alberi QuickFind con bilanciamento sulle union sono in grado di eseguire una sequenza di m operazioni* find, *n operazioni* makeSet *ed al più $(n-1)$ operazioni* union *in un tempo totale di $O(m + n \log n)$. L'occupazione di memoria è $O(n)$.*

Dimostrazione. Le limitazioni sull'occupazione di memoria e sul tempo richiesto dalle operazioni makeSet e find derivano direttamente dal Teorema 9.1. Questo

classe QuickFindBilanciato **implementa** UnionFind:
dati: $S(n) = O(n)$
una collezione di insiemi disgiunti di elementi *elem*; ogni insieme ha un nome *name*.
operazioni:
makeSet(*elem e*) $T(n) = O(1)$
crea un nuovo albero, composto da due nodi: una radice ed un unico figlio (foglia). Memorizza e sia nella radice che nella foglia dell'albero. Inizializza la cardinalità del nuovo insieme ad 1, assegnando il valore size$(x) = 1$ alla radice x.
union(*name a*, *name b*) $T_{am} = O(\log n)$
considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Se size$(A) \geq$ size(B), muovi tutti i puntatori dalle foglie di B alla radice di A, e cancella la vecchia radice di B. Altrimenti (size$(B) >$ size(A)) memorizza nella radice di B il nome A, muovi tutti i puntatori dalle foglie di A alla radice di B, e cancella la vecchia radice di A. In entrambi i casi assegna al nuovo insieme la somma delle cardinalità dei due insiemi originali (size(A) + size(B)).
find(*elem e*) $\rightarrow$ *name* $T(n) = O(1)$
accede al nodo x corrispondente all'elemento e. Da tale nodo segue il puntatore al padre, che è la radice dell'albero, e restituisce il nome memorizzato in tale radice.

Figura 9.8 Realizzazione delle operazioni makeSet, union e find mediante alberi QuickFind bilanciati.

implica un tempo totale di $O(m + n)$ per le find e le makeSet. Per dimostrare che il tempo totale delle union è $O(n \log n)$, utilizzeremo la tecnica dei crediti introdotta nel Paragrafo 2.7 del Capitolo 2. La politica di assegnazione dei crediti che useremo è la seguente:

Per ogni operazione makeSet *assegnamo $\lfloor \log_2 n \rfloor$ crediti alla nuova foglia creata.*

Il numero totale di crediti richiesti da tale assegnazione è chiaramente $O(n \log n)$. Dimostriamo ora che i crediti assegnati durante le operazioni makeSet sono sufficienti a pagare per tutto il lavoro richiesto dalle operazioni union. In particolare, assumiamo che ogni spostamento di una foglia in un nuovo insieme, causato da una operazione union, venga pagato con uno dei crediti assegnati alla foglia nel momento della sua creazione.

Sia ρ una generica foglia di un albero QuickFind con bilanciamento. All'inizio, immediatamente dopo l'operazione makeSet che ha creato ρ, ρ si troverà in un insieme di cardinalità 1 ed avrà esattamente $\lfloor \log_2 n \rfloor$ crediti. Ogni volta che la foglia ρ viene spostata, perde un credito per pagare tale spostamento. In base al

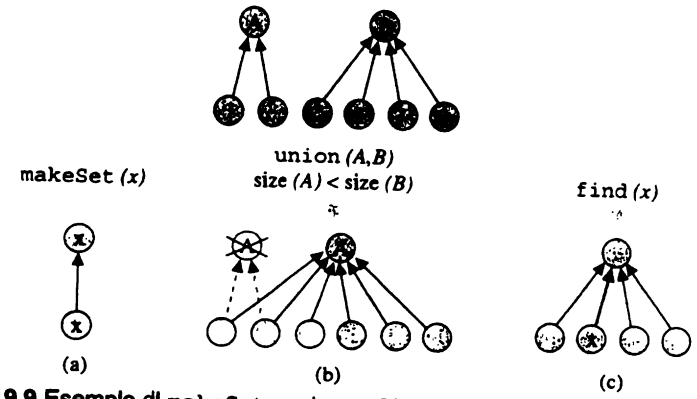


Figura 9.9 Esempio di `makeSet`, `union` e `find` su alberi QuickFind bilanciati.

Lemma 9.1, però, ρ si troverà in un insieme che è grande almeno il doppio dell'insieme di cui faceva parte prima dello spostamento. Questo implica che dopo k spostamenti, $k \geq 0$, e quindi dopo aver rilasciato k crediti, ρ si troverà in un insieme di cardinalità almeno 2^k ed avrà $(\lfloor \log_2 n \rfloor - k)$ crediti residui.

Dato che alla fine della sequenza di operazioni ρ può trovarsi in un insieme di cardinalità al più n, se ρ ha subito k spostamenti dovremo necessariamente avere $2^k \leq n$. In altri termini, ρ potrà subire al massimo $k \leq \lfloor \log_2 n \rfloor$ spostamenti, e quindi i crediti che le sono stati assegnati al momento della sua creazione sono sufficienti a pagare per tutti i suoi spostamenti causati dalle operazioni `union`.

Poiché abbiamo n foglie, questo dimostra che il tempo totale delle `union` è $O(n \log n)$. □

Concludiamo questo paragrafo osservando che, in base al Teorema 9.3, il tempo di esecuzione delle `union` nell'algoritmo QuickFind bilanciato è $O(\log n)$, se ammortizzato su sequenze di operazioni contenenti almeno $\Omega(n)$ `union`.

9.2.2 Bilanciamento per algoritmi di tipo QuickUnion

Vedremo ora come migliorare le prestazioni degli alberi QuickUnion descritte nel Paragrafo 9.1.2. Anche in questo caso, ci concentreremo sull'operazione più costosa, che è la `find`: in base all'analisi effettuata nel Teorema 9.2, infatti, la `find` ha un tempo di esecuzione pari a $O(n)$. La ragione di tale complessità risiede nel fatto che, per facilitare l'operazione `union`, tolleriamo che l'altezza di alberi QuickUnion possa crescere senza alcun controllo. Anche per gli algoritmi QuickUnion osserviamo che l'operazione `union` viene implementata senza molta flessibilità: infatti, nel Paragrafo 9.1.2 abbiamo effettuato una `union`(A, B) rendendo sempre la radice dell'insieme B figlia della radice dell'insieme A. Questo può sembrare particolarmente inefficiente in quei casi in cui l'insieme B ha un'altezza maggiore dell'insieme A. Potrebbe essere chiarificatore a tale scopo l'Esempio 9.2, utilizzato nel Paragrafo 9.1.2 per dimostrare che alberi QuickUnion possono avere nel caso peggiore altezza $\Omega(n)$: per costruire un albero di altezza $\Omega(n)$ rendevamo consistentemente la radice dell'albero più alto figlia della radice dell'albero più basso.

Union by rank. Per evitare che l'altezza dell'albero risultante cresca troppo, sembrerebbe più logico rendere consistentemente la radice dell'albero più basso figlia della radice dell'albero più alto. Ancora una volta, così facendo potremmo rischiare di non garantire la proprietà che il nome del nuovo insieme $A \cup B$ sia A: per assicurare questo, basterà però semplicemente memorizzare A nella radice del nuovo albero risultante.

Ci riferiremo a questo algoritmo come un *algoritmo di tipo QuickUnion con bilanciamento sulle union*, e agli alberi ottenuti in questo modo come *alberi QuickUnion con bilanciamento in altezza* oppure più semplicemente come *alberi QuickUnion bilanciati in altezza*. L'algoritmo utilizzerà per ogni radice x il valore rank(x), che esprime l'altezza dell'albero di cui x è radice. Per questo motivo chiameremo anche l'implementazione dell'operazione di `union` come *union by rank*. L'implementazione delle operazioni `makeSet`, `union` e `find` con alberi QuickUnion bilanciati in altezza è definita in Figura 9.10 ed illustrata graficamente nella Figura 9.11.

Per comprendere cosa guadagnamo dall'eseguire l'euristica di *union by rank*, cercheremo di caratterizzare l'altezza massima degli alberi QuickUnion bilanciati. Il seguente lemma ci fornisce un primo indizio in questa direzione.

Lemma 9.2 *Un albero QuickUnion bilanciato in altezza con radice x ha almeno $2^{\text{rank}(x)}$ nodi.*

Dimostrazione. Procediamo per induzione sul numero di operazioni effettuate. All'inizio, non ci sono alberi e quindi la base dell'induzione è banalmente verificata. Assumiamo che il lemma sia verificato prima di un'operazione, e dimostriamo che sarà valido anche dopo. Dato che le `find` non modificano alberi o rank, consideriamo solamente le operazioni `makeSet` e `union`.

Consideriamo prima un'operazione `makeSet`(x). Tale operazione non modifica alberi e rank preesistenti, ed introduce un albero di altezza 0, contenente solo il nodo x, con rank$(x) = 0$. Tale albero ha $2^{\text{rank}(x)} = 2^0 = 1$ nodo e pertanto verifica il lemma.

Consideriamo ora un'operazione `union`(A, B). Indichiamo con rank(A) e rank(B) il rank delle radici dei due alberi A e B prima della `union`(A, B), e con rank$(A \cup B)$ il rank della radice dell'albero risultante. Similmente, indichiamo con $|A|$ e $|B|$ il numero di nodi nei due alberi A e B prima della `union`(A, B), e con $|A \cup B|$ il numero di nodi nell'albero risultante. Osserviamo che avremo sempre $|A \cup B| = |A| + |B|$. Distinguiamo ora tre casi.

1. Se rank$(B) <$ rank(A), l'operazione `union` rende la radice dell'albero B figlia della radice dell'albero A. La radice dell'albero risultante avrà pertanto rank$(A \cup B) =$ rank(A), e l'albero risultante avrà $|A \cup B| = |A| +$

classe `QuickUnionBilanciato` **implementa** `UnionFind`:
dati: $S(n) = O(n)$
una collezione di insiemi disgiunti di elementi *elem*; ogni insieme ha un nome *name*.

operazioni:

`makeSet`(*elem e*) $T(n) = O(1)$
crea un nuovo albero, composto da un unico nodo x. Memorizza e sia come valore che come nome in tale nodo. Inizializza $\text{rank}(x) = 0$ (l'altezza del nuovo albero è 0), memorizzando nel nodo x anche tale valore di rank.

`union`(*name a*, *name b*) $T(n) = O(1)$
considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Confronta $\text{rank}(A)$ e $\text{rank}(B)$, distinguendo tre casi.

1. Se $\text{rank}(B) < \text{rank}(A)$, rende la radice dell'albero B figlia della radice dell'albero A.
2. Se $\text{rank}(A) < \text{rank}(B)$, rende la radice dell'albero A figlia della radice dell'albero B, e memorizza A come nome nella radice del nuovo albero.
3. Se $\text{rank}(A) = \text{rank}(B)$, rende la radice dell'albero B figlia della radice dell'albero A, ed aggiorna $\text{rank}(A) = \text{rank}(A) + 1$.

`find`(*elem e*) $\rightarrow$ *name* $T(n) = O(\log n)$
accede alla foglia x corrispondente all'elemento e. Partendo da tale nodo, segue ripetutamente i puntatori al padre fino a raggiungere la radice dell'albero. Restituisce il nome memorizzato in tale radice.

Figura 9.10 Realizzazione delle operazioni `makeSet`, `union` e `find` mediante alberi QuickUnion bilanciati in altezza.

$|B|$ nodi. Utilizzando l'ipotesi induttiva sugli insiemi A e B, il numero di nodi nell'albero risultante è pertanto

$$|A \cup B| = |A| + |B| \geq 2^{\text{rank}(A)} + 2^{\text{rank}(B)} > 2^{\text{rank}(A)} = 2^{\text{rank}(A \cup B)}$$

2. Se $\text{rank}(A) < \text{rank}(B)$, l'operazione `union` rende la radice dell'albero A figlia della radice dell'albero B, e memorizza A come nome nella radice del nuovo albero. La radice dell'albero risultante avrà pertanto $\text{rank}(A \cup B) = \text{rank}(B)$, e l'albero risultante avrà $|A \cup B| = |A| + |B|$ nodi. Utilizzando l'ipotesi induttiva sugli insiemi A e B, il numero di nodi nell'albero risultante è pertanto

$$|A \cup B| = |A| + |B| \geq 2^{\text{rank}(A)} + 2^{\text{rank}(B)} > 2^{\text{rank}(B)} = 2^{\text{rank}(A \cup B)}$$

3. Se $\text{rank}(A) = \text{rank}(B)$, l'operazione `union` rende la radice dell'albero B figlia della radice dell'albero A, ed aggiorna il rank di A a $\text{rank}(A) + 1$.

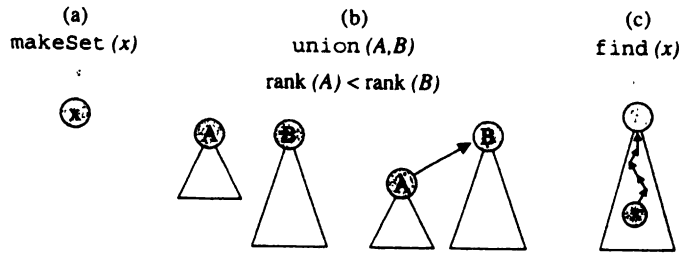


Figura 9.11 Esempio di `makeSet`, `union` e `find` su alberi QuickUnion bilanciati in altezza.

La radice dell'albero risultante avrà pertanto $\text{rank}(A \cup B) = \text{rank}(A) + 1$, e l'albero risultante avrà $|A \cup B| = |A| + |B|$ nodi. Utilizzando l'ipotesi induttiva sugli insiemi A e B, il numero di nodi nell'albero risultante è pertanto

$$|A \cup B| = |A| + |B| \geq 2^{\text{rank}(A)} + 2^{\text{rank}(B)} \geq$$
$$\geq 2 \cdot 2^{\text{rank}(A)} = 2^{\text{rank}(A)+1} = 2^{\text{rank}(A \cup B)}$$

In ognuno dei tre casi, il lemma sarà quindi verificato anche dopo l'operazione `union`(A, B). □

Il seguente corollario è una conseguenza immediata del Lemma 9.2.

Corollario 9.1 *Durante una sequenza di operazioni* `makeSet`, `union` *e* `find`, *l'altezza di un albero QuickUnion bilanciato è limitata superiormente da* $\lfloor \log_2 n \rfloor$, *dove n è il numero totale di* `makeSet`.

Dimostrazione. Sia T un qualsiasi albero QuickUnion bilanciato, e sia x la sua radice. Sia inoltre $\text{size}(x)$ il numero di nodi in T. Osserviamo che per definizione l'altezza di T è data da $\text{rank}(x)$, e che $\text{size}(x) \leq n$. Per il Lemma 9.2 abbiamo che $\text{size}(x) \geq 2^{\text{rank}(x)}$, ovvero $\text{rank}(x) \leq \lfloor \log_2(\text{size}(x)) \rfloor \leq \lfloor \log_2 n \rfloor$. □

Il Corollario 9.1 ci garantisce che l'euristica *union by rank* è in grado di mantenere alberi di tipo QuickUnion con altezza $O(\log n)$. Visto che l'altezza di un albero fornisce una delimitazione superiore al tempo di esecuzione di un'operazione `find`, otteniamo immediatamente il seguente teorema.

Teorema 9.4 *Alberi QuickUnion bilanciati in altezza sono in grado di supportare operazioni* `makeSet` *e* `union` *in tempo costante. Una* `find` *richiede nel caso peggiore tempo $O(\log n)$, dove n è il numero totale di operazioni* `makeSet` *eseguite nella sequenza. L'occupazione di memoria è $O(n)$.*

Di seguito, riassumiamo le principali proprietà del *rank* di alberi QuickUnion bilanciati in altezza, che sono verificate durante una qualsiasi sequenza di operazioni.

Proprietà 9.1 *Il rank di un generico nodo x assume inizialmente il valore 0 (i.e., immediatamente dopo l'operazione* `makeSet`(x) *che lo ha creato), ed aumenta nel tempo fintantoché x rimane radice. Non appena x cessa di essere radice, il suo rank rimane invariato.*

Proprietà 9.2 *Sia x un generico nodo (non radice) e $p(x)$ il suo nodo padre: allora rank$(p(x))$ > rank (x). In particolare, se seguiamo un cammino da un generico nodo x alla sua radice, otteniamo valori del rank monotoni crescenti.*

Proprietà 9.3 *Come conseguenza del Lemma 9.2, per ogni nodo x, abbiamo che rank$(x) \leq \lfloor \log_2 n \rfloor$, dove n è il numero totale di nodi presenti.*

Union by size. Concludiamo questo paragrafo osservando che si può ottenere un'altra forma di bilanciamento sull'algoritmo QuickUnion, basandosi questa volta sulla cardinalità degli alberi invece che sulla loro altezza. Tale algoritmo manterrà per ogni radice x il valore size(x), che esprime la cardinalità dell'albero di cui x è radice. Per questo motivo chiameremo questa implementazione dell'operazione di `union` come *union by size*.

Ci riferiremo a questo algoritmo come un *algoritmo di tipo QuickUnion con bilanciamento in cardinalità*, e agli alberi ottenuti in questo modo come *alberi QuickUnion con bilanciamento in cardinalità*. L'implementazione delle operazioni `makeSet`, `union` e `find` con alberi QuickUnion bilanciati in cardinalità è definita in Figura 9.10.

Seguendo un approccio molto simile a quello seguito nel caso di *union by rank*, è possibile dimostrare che anche le *union by size* mantengono alberi di altezza logaritmica nel numero dei nodi, e quindi sono in grado di supportare le operazioni `makeSet`, `union` e `find` esattamente negli stessi tempi del Teorema 9.4 (si veda il Problema 9.4). La Tabella 9.2 riassume i tempi di esecuzione ottenibili con algoritmi di tipo QuickFind e QuickUnion con e senza bilanciamento. Come vedremo nel prossimo paragrafo, questi non sono ancora gli algoritmi più efficienti per il problema union-find.

	`makeSet`	`union`	`find`
`QuickFind`	$O(1)$	$O(n)$	$O(1)$
`QuickFindBilanciato`	$O(1)$	$O(\log n)$ amm.	$O(1)$
`QuickUnion`	$O(1)$	$O(1)$	$O(n)$
`QuickUnionBilanciatoRank`	$O(1)$	$O(1)$	$O(\log n)$
`QuickUnionBilanciatoSize`	$O(1)$	$O(1)$	$O(\log n)$

Tabella 9.2 Tempi di esecuzione ottenibili mediante realizzazioni QuickFind e QuickUnion con e senza bilanciamento. Il tempo di esecuzione delle `union` mediante alberi QuickFind bilanciati è ammortizzato su sequenze contenenti almeno $\Omega(n)$ `union`.

classe `QuickUnionBilanciatoSize` **implementa** `UnionFind`:
dati: $S(n) = O(n)$
una collezione di insiemi disgiunti di elementi *elem*; ogni insieme ha un nome *name*.

operazioni:
`makeSet`(*elem e*) $T(n) = O(1)$
crea un nuovo albero, composto da un unico nodo x. Memorizza e sia come valore che come nome in tale nodo. Inizializza size$(x) = 1$ (la cardinalità del nuovo albero è 1), memorizzando nel nodo x anche tale valore di size.

`union`(*name a*, *name b*) $T(n) = O(1)$
considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Confronta size(A) e size(B), distinguendo tre casi.

1. Se size(B) < size(A), rende la radice dell'albero B figlia della radice dell'albero A.
2. Se size(A) < size(B), rende la radice dell'albero A figlia della radice dell'albero B, e memorizza il nome A nella radice del nuovo albero.
3. Se size(A) = size(B), rende la radice dell'albero B figlia della radice dell'albero A.

In ogni caso, memorizza nella radice del nuovo albero la nuova cardinalità size(A) + size(B).

`find`(*elem e*) → *name* $T(n) = O(\log n)$
accede alla foglia x corrispondente all'elemento e. Partendo da tale nodo, segue ripetutamente i puntatori al padre fino a raggiungere la radice dell'albero. Restituisce il nome memorizzato in tale radice.

Figura 9.12 Realizzazione delle operazioni `makeSet`, `union` e `find` mediante alberi QuickUnion bilanciati in cardinalità.

9.3 Euristiche di compressione nell'operazione find

Se vogliamo migliorare ulteriormente i tempi di esecuzione del Teorema 9.4, sembra indispensabile riuscire a ridurre ulteriormente l'altezza degli alberi union-find. Vedremo adesso tre euristiche che hanno proprio lo scopo di comprimere il cammino visitato durante un'operazione `find`. Per $\ell \geq 3$, siano $u_0, u_1, \ldots, u_{\ell-1}$ i nodi incontrati nel cammino esaminato da una `find`(x), dove $u_0 = x$, ed $u_{\ell-1}$ è la radice dell'albero contenente x. Notiamo che non viene preso in considerazione il caso $\ell \leq 2$, perché in tal caso il cammino della `find` contiene al più due nodi e quindi non serve effettuare alcuna forma di compressione.

- L'euristica di *path compression* [9] rende tutti i nodi u_i, $0 \leq i \leq \ell - 3$, figli della radice $u_{\ell-1}$ (notiamo che $u_{\ell-2}$ è già figlio della radice $u_{\ell-1}$).

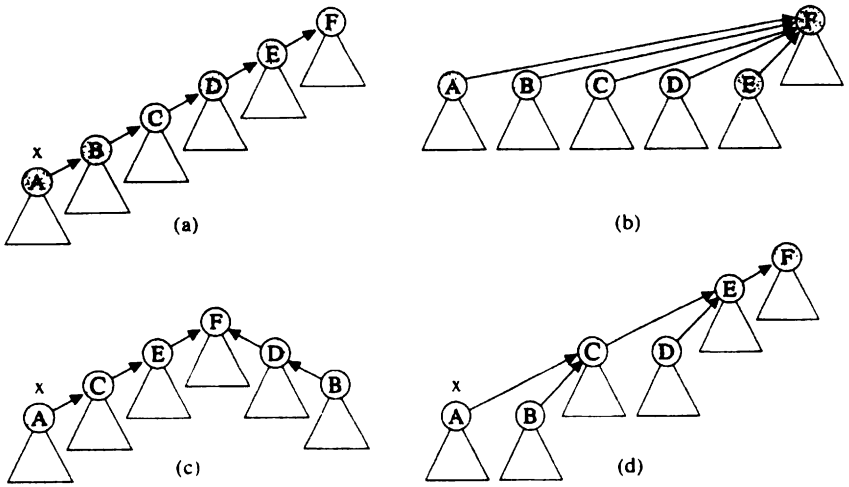


Figura 9.13 (a) L'albero prima di eseguire l'operazione find(x); (b) L'albero in figura (a) dopo aver eseguito find(x) con path compression; (c) L'albero in figura (a) dopo aver eseguito find(x) con path splitting; (d) L'albero in figura (a) dopo aver eseguito find(x) con path halving.

- L'euristica di *path splitting* [14, 15] rende il nodo u_i, $0 \leq i \leq \ell - 3$, figlio del suo nonno u_{i+2}.
- L'euristica di *path halving* [14, 15] rende il nodo u_{2i}, $0 \leq i \leq \lfloor \frac{\ell-1}{2} \rfloor - 1$, figlio del suo nonno u_{2i+2}.

La Figura 9.13 illustra ognuna delle tre euristiche al lavoro. Nel prossimo paragrafo dimostreremo, tramite un'analisi sofisticata, che tali euristiche sono in grado di fornire algoritmi più efficienti di quelli analizzati finora.

9.4 * Union-find con bilanciamento e compressione

Le euristiche introdotte nei paragrafi precedenti ci forniscono un totale di sei algoritmi, a seconda del tipo di bilanciamento sulle `union` utilizzato (*union by rank* o *union by size*) e del tipo di compressione dei cammini utilizzato sulle `find` (*path compression*, *path splitting* o *path halving*). E' possibile dimostrare che tutti e sei gli algoritmi hanno gli stessi tempi di esecuzione.

In questo paragrafo analizzeremo per semplicità solamente il tempo di esecuzione dell'algoritmo che utilizza *union by rank* combinata con *path compression*, e dimostreremo che l'euristica di compressione dei cammini consente effettivamente di migliorare le prestazioni ottenibili applicando solamente l'euristica di bilanciamento delle `union`.

9.4.1 Proprietà dei rank

Ricordiamo che entrambe le operazioni `makeSet` e `union` possono essere facilmente implementate in tempo costante, e quindi il cuore della nostra analisi sarà relativo alle operazioni `find`. La nozione cruciale che utilizzeremo nella nostra analisi sarà la nozione di *rank*, di cui ricordiamo la definizione. All'inizio, il *rank* di un nuovo nodo creato da un'operazione `makeSet` è 0. Il *rank* di un nodo viene incrementato di 1 solamente in caso di un'operazione `union` eseguita tra due insiemi con radici dello stesso *rank*.

Anche nel caso in cui oltre ad *union by rank* applichiamo la *path compression*, valgono tutte le considerazioni fatte sul *rank* nel Paragrafo 9.2.2. In particolare, anche in questo caso continuano a valere il Lemma 9.2 ed il Corollario 9.2, e le Proprietà 9.1, 9.2 e 9.3. C'è però una differenza importante. Nel Paragrafo 9.2.2 avevamo infatti osservato che rank(x) forniva esattamente l'altezza del nodo x nell'albero che lo conteneva. A causa della compressione dei cammini, l'altezza attuale di un generico nodo x potrebbe essere inferiore al valore stimato da rank(x), e quindi in questo caso rank(x) non fornisce esattamente l'altezza del nodo, ma solamente una sua delimitazione superiore (upper bound).

9.4.2 Un'analisi preliminare

In questo paragrafo dimostreremo che la tecnica di compressione dei cammini è in grado di migliorare sensibilmente le prestazioni ottenute dalle strutture dati descritte nei paragrafi precedenti. Intuitivamente, basta osservare che l'euristica di path compression abbassa la profondità dei cammini su cui agisce, e quindi è anche in grado di ridurre potenzialmente l'altezza degli alberi. Prima di tutto, dimostreremo che siamo in grado di riottenere almeno gli stessi bound del Teorema 9.3. La tecnica utilizzata nella dimostrazione ci sarà utile per raffinare ulteriormente nel seguito la nostra analisi.

Teorema 9.5 *Con le euristiche di union by rank e di path compression, una qualsiasi sequenza di n operazioni* `makeSet`, *m operazioni* `find` *ed al più $(n-1)$ operazioni* `union` *può essere realizzata in tempo totale $O(m + n \log n)$.*

Dimostrazione. Ciascuna operazione `makeSet` e `union` richiede tempo costante nel caso peggiore, mentre invece il costo di un'operazione `find`(x) è dato dal numero di nodi contenuti nel cammino da x alla radice.

Per dimostrare il teorema, stimeremo il costo delle operazioni `find` utilizzando una tecnica di ammortizzazione. In particolare, la nostra politica di ammortizzazione consisterà nell'assegnare i crediti nel modo seguente:

- Durante l'esecuzione di una `makeSet`(x), assegneremo 1 credito per eseguire il lavoro relativo all'operazione stessa, più $\lfloor \log_2 n \rfloor$ crediti al nodo x.
- Durante l'esecuzione di una `union`, assegneremo 1 credito per eseguire il lavoro relativo all'operazione stessa.
- Durante l'esecuzione di una `find`, assegneremo 2 crediti per eseguire il lavoro relativo all'operazione stessa.

	crediti allocati	crediti spesi	bilancio (surplus o debito)
makeSet	$1 + \lfloor \log_2 n \rfloor$	1	$\lfloor \log_2 n \rfloor$
union	1	1	0
find (ℓ nodi)	2	ℓ	$-(\ell - 2)$

Tabella 9.3 Politica di allocazione dei crediti per il Teorema 9.5.

La Tabella 9.3 riassume questa politica di allocazione dei crediti. Il numero totale di crediti assegnati durante una qualsiasi sequenza di n operazioni `makeSet`, m operazioni `find` ed al più $(n-1)$ operazioni `union` è quindi delimitato superiormente da

$$n(1 + \lfloor \log_2 n \rfloor) + (n-1) + 2m = O(m + n \log n)$$

Per dimostrare il teorema, basta quindi mostrare che i crediti allocati sono sufficienti a pagare per eseguire il lavoro richiesto dalle operazioni. Come illustrato nella Tabella 9.3, le uniche operazioni che non hanno crediti sufficienti a ripagare la loro esecuzione sono le `find` su cammini contenenti $\ell > 2$ nodi. Per pagare tali operazioni, utilizzeremo il surplus generato dalle `makeSet`. In maggiore dettaglio, sia π il cammino visitato dalla `find`, contenente ℓ nodi:

- I costi relativi alla visita della radice e del figlio della radice di π vengono pagati con i 2 crediti assegnati all'operazione `find`.
- I costi relativi ad ogni altro nodo x di π vengono pagati con i crediti "immagazzinati" in x (dall'operazione `makeSet`(x)).

Per concludere la dimostrazione, dobbiamo mostrare che i $\lfloor \log_2 n \rfloor$ crediti originariamente "immagazzinati" in eccesso nel nodo x dalla `makeSet`(x) sono sufficienti per ripagare tutto il lavoro causato dalle operazioni `find` che coinvolgono x. Infatti, ogni volta che addebitiamo un credito al nodo x a causa di una `find`, x non sarà né radice né figlio di radice, ed a causa della *path compression* acquisirà come padre una nuova radice. In altri termini, per la Proprietà 9.2 del rank, ogni volta che addebitiamo un credito ad un nodo x a causa di una `find`, il (nuovo) padre di x avrà rank strettamente maggiore del rank del vecchio padre di x. Come conseguenza della Proprietà 9.3 del rank (rank$\leq \lfloor \log_2 n \rfloor$), x potrà avere al più $\lfloor \log_2 n \rfloor$ crediti addebitati, e quindi i crediti originariamente "immagazzinati" in eccesso in x da `makeSet`(x) sono sufficienti per ripagare tutto il lavoro causato dalle operazioni `find` che coinvolgono x. □

9.4.3 Un'analisi più raffinata

In questo paragrafo raffineremo l'analisi del Teorema 9.5 per ottenere una stima più accurata dei tempi di esecuzione dell'algoritmo che applica le euristiche di union by rank e di path compression. Prima di far questo, abbiamo bisogno di

definire delle nuove funzioni. Definiamo innanzitutto la funzione $\log^{(i)} n$, per $i \geq 1$, nel modo seguente:

$$\begin{aligned}
\log^{(1)} n &= \log n \\
\log^{(2)} n &= \log(\log^{(1)} n) &&= \log\log n \\
\log^{(3)} n &= \log(\log^{(2)} n) &&= \log\log\log n \\
&\dots \\
\log^{(i)} n &= \log(\log^{(i-1)} n) &&= \underbrace{\log\log\dots\log}_{i \text{ volte}} n, \qquad \text{per } i \geq 2
\end{aligned}$$

Molto informalmente, la funzione $\log^*$ ci dice quante volte dobbiamo applicare ripetutamente la funzione log per rendere il risultato ≤ 1:

$$\log^* n = \left\{ \min i \mid \log^{(i)} n \leq 1 \right\}$$

La funzione $\log^*$ è una funzione dalla crescita estremamente lenta, come evidenziato dai suoi primi valori:

$$\begin{aligned}
\log^*(16) &= 3 \\
\log^*(65536) &= 4 \\
\log^*(2^{65536}) &= 5
\end{aligned}$$

Introduciamo adesso una funzione F dalla crescita estremamente veloce, definita nel modo seguente:

$$F(i) = \begin{cases} 2^{F(i-1)} & \text{se } i \geq 1 \\ 1 & \text{se } i = 0 \end{cases}$$

Notiamo che la funzione $F()$ è collegata alla funzione $\log^*()$:

$$\begin{array}{lll}
F(0) = 1 & \log^* x = 0, & 0 < x \leq 1 \\
F(1) = 2^1 = 2 & \log^* x = 1, & 1 < x \leq 2 \\
F(2) = 2^2 = 4 & \log^* x = 2, & 2 < x \leq 4 \\
F(3) = 2^4 = 16 & \log^* x = 3, & 4 < x \leq 16 \\
F(4) = 2^{16} = 65536 & \log^* x = 4, & 16 < x \leq 65536 \\
F(5) = 2^{65536} & \log^* x = 5, & 65536 < x \leq 2^{65536} \\
\dots & \dots &
\end{array}$$

Prima di analizzare ulteriormente il tempo di esecuzione dell'algoritmo che utilizza le euristiche di union by rank e di path compression, abbiamo bisogno del seguente lemma:

Lemma 9.3 *Durante l'esecuzione di una qualsiasi sequenza di* `makeSet`, `union` *e* `find`, *al più* $\frac{n}{2^r}$ *nodi possono avere rank uguale ad* r.

Dimostrazione. Quando un nodo v ottiene un valore di rank pari ad r per la prima volta, questo deve necessariamente avvenire dopo una operazione `union` e quindi v deve essere la radice di qualche albero. Per il Lemma 9.2, a questo punto v deve avere almeno 2^r discendenti. Osserviamo inoltre che, dopo successive operazioni di `union`, v potrebbe cessare di essere radice, potrebbe anche perdere qualcuno dei suoi discendenti a causa della path compression, ma comunque il valore del suo rank non potrà essere mai più modificato.

Per dimostrare il lemma, assumiamo che un generico nodo sia etichettato col valore r nel momento in cui la sua radice v aumenta per la prima volta il suo rank al valore r. Da questo punto in poi, tale nodo non potrà essere etichettato nuovamente col valore r, dato che le sue future radici dovranno sicuramente avere un rank strettamente superiore ad r, e quindi tale etichettatura è sicuramente univoca. Conseguentemente, per ogni radice che aumenta il suo rank ad r ci saranno almeno 2^r nodi che saranno etichettati col valore r. Visto che ci sono n nodi in totale, non sarà quindi mai possibile avere più di $\frac{n}{2^r}$ nodi di rank r. □

Una nozione cruciale per l'analisi dell'algoritmo con le euristiche di union by rank e di path compression è quella di partizionare ad ogni istante i nodi in *blocchi* $B(i)$ nel modo seguente:

Se un nodo v ha rank r, allora v appartiene al blocco $B(\log^* r)$.

Osserviamo che, essendo il rank di ogni nodo al più $\log_2 n$ per il Lemma 9.2, il numero totale di blocchi sarà al più $\log^*(\log n) = \log^* n - 1$. Conseguentemente, ci saranno al più $(\log^* n)$ blocchi, da $B(0)$ a $B(\log^* n - 1)$. In particolare avremo i seguenti blocchi:

Il blocco $B(0)$ contiene nodi di rank compreso in $[0, 1]$ ($[0, F(0)]$)
Il blocco $B(1)$ contiene nodi di rank compreso in $[2, 2]$ ($[F(0) + 1, F(1)]$)
Il blocco $B(2)$ contiene nodi di rank compreso in $[3, 4]$ ($[F(1) + 1, F(2)]$)
Il blocco $B(3)$ contiene nodi di rank compreso in $[5, 16]$ ($[F(2) + 1, F(3)]$)
Il blocco $B(4)$ contiene nodi di rank compreso in $[17, 65536]$ ($[F(3) + 1, F(4)]$)
...
Il blocco $B(i)$ contiene nodi di rank compreso in $[F(i - 1) + 1, F(i)]$

Siamo ora pronti ad analizzare il tempo di esecuzione dell'algoritmo con le euristiche di union by rank e di path compression.

Teorema 9.6 *Con le euristiche di union by rank e di path compression, una qualsiasi sequenza di n operazioni* `makeSet`, *m operazioni* `find` *ed al più $(n - 1)$ operazioni* `union` *può essere implementata in tempo totale $O((n + m)\log^* n)$.*

Dimostrazione. Visto che le operazioni `makeSet` e `union` richiedono tempo costante nel caso peggiore, la dimostrazione sarà ancora una volta centrata sull'analisi delle operazioni `find`. Procediamo a raffinare la politica di assegnazione dei crediti vista nel Teorema 9.5 nel modo seguente:

- Durante l'esecuzione di una `makeSet`(x), assegneremo 1 credito per eseguire il lavoro relativo all'operazione stessa, più 1 credito ad ogni blocco della partizione, per un totale di $1 + \log^* n$ crediti.

	crediti allocati	crediti spesi	bilancio (surplus o debito)
`makeSet`	$1 + \log^* n$	1	$\log^* n$
`union`	1	1	0
`find` (ℓ nodi)	$1 + \log^* n$	ℓ	$-(\ell - \log^* n - 1)$

Tabella 9.4 Politica di allocazione dei crediti per il Teorema 9.6.

- Durante l'esecuzione di una `union`, assegneremo 1 credito per eseguire il lavoro relativo all'operazione stessa.
- Durante l'esecuzione di una `find`, assegneremo $1 + \log^* n$ crediti per eseguire il lavoro relativo all'operazione stessa.

Osserviamo che il numero totale di crediti assegnati durante una qualsiasi sequenza di n operazioni `makeSet`, m operazioni `find` ed al più $(n - 1)$ operazioni `union` è delimitato superiormente da

$$n(1 + \log^* n) + (n - 1) + (m \log^* n) = O((n + m)\log^* n).$$

Per dimostrare il teorema, basta quindi mostrare che i crediti allocati sono sufficienti a pagare per eseguire tutto il lavoro richiesto dalle operazioni. Notiamo che, a differenza del Teorema 9.5, il surplus di crediti generato dall'operazione `makeSet` è inferiore, a fronte di un leggero aumento dei crediti assegnati all'operazione `find`, come è illustrato nella Tabella 9.4. Sia π il cammino visitato dalla `find`, contenente ℓ nodi. Raffiniamo l'analisi del Teorema 9.5 nel modo seguente:

(1) I costi relativi alla visita della radice, del figlio della radice e di tutti i nodi del cammino π che non sono nello stesso blocco del loro padre, vengono pagati con gli $(1 + \log^* n)$ crediti assegnati all'operazione `find`.
(2) I costi relativi ad ogni altro nodo x di π (ovvero ogni nodo che è nello stesso blocco del proprio padre) vengono pagati con i crediti immagazzinati nel blocco di appartenenza di x (da un'operazione `makeSet`).

Diremo che la radice, il figlio della radice e tutti i nodi che non sono nello stesso blocco del loro padre richiedono crediti di tipo (1), mentre tutti gli altri nodi del cammino visitato dalla `find` richiedono crediti di tipo (2). Sia x un generico nodo. Osserviamo che ogni volta che sposteremo x a causa di una path compression, per la Proprietà 9.2 del rank il (nuovo) padre di x avrà rank strettamente maggiore del rank del vecchio padre di x. Questo implica che, non appena x acquisirà un padre che è in un blocco diverso dal suo, da quel punto in poi il padre di x si troverà sempre in un blocco diverso da x. Pertanto x potrà richiedere inizialmente crediti di tipo (2), ma da un certo punto in poi richiederà escusivamente crediti di tipo (1).

Calcoliamo ora il numero totale di crediti di tipo (1) e di tipo (2) richiesti in totale da una sequenza di operazioni. Per analizzare i crediti di tipo (1), basta semplicemente osservare che ci sono al più $\log^* n$ blocchi distinti, e conseguentemente per ogni `find` possono esserci al più $(\log^* n - 1)$ nodi che si trovano in un blocco diverso da quello del proprio padre. Aggiungendo a questi nodi la radice ed il figlio della radice del cammino visitato dalla `find`, abbiamo un totale di al più

$$2 + (\log^* n - 1) = \log^* n + 1$$

nodi che richiedono crediti di tipo (1) per ogni operazione `find`. Gli $(1 + \log^* n)$ crediti assegnati ad ogni operazione `find` saranno quindi sufficienti a pagare per la visita di tali nodi.

Per concludere la dimostrazione, dobbiamo mostrare che i crediti allocati in eccesso durante le operazioni `makeSet` sono sufficienti per ripagare le richieste di crediti di tipo (2) per tutti gli altri nodi nei cammini delle `find`. Consideriamo il generico blocco $B(i)$, $0 \leq i \leq \log^* n - 1$. Ogni nodo che si trova nel blocco $B(i)$ potrà richiedere al più $F(i)$ crediti di tipo (2) durante la sequenza di operazioni, dato che ogni volta che richiederà un credito di tipo (2) il rank di suo padre dovrà necessariamente aumentare, e ci sono al più $F(i) - F(i-1) \leq F(i)$ possibili valori diversi del rank nel blocco $B(i)$. Il numero totale di crediti di tipo (2) richiesti da ogni nodo nel blocco $B(i)$, $0 \leq i \leq \log^* n - 1$, può essere pertanto al più $F(i)$.

Per avere una stima di quanti nodi si trovano in $B(i)$, osserviamo che per il Lemma 9.2 possono esserci al più $\frac{n}{2^r}$ nodi di rank r. Visto che $B(i)$ contiene nodi con rank compreso nell'intervallo $[F(i-1)+1, F(i)]$, il numero totale di nodi nel blocco $B(i)$, $0 \leq i \leq \log^* n - 1$, sarà dato da:

$$\sum_{r=F(i-1)+1}^{F(i)} \left(\frac{n}{2^r}\right) \leq n \frac{\left(\frac{1}{2}\right)^{F(i-1)+1} - \left(\frac{1}{2}\right)^{F(i)+1}}{1 - \left(\frac{1}{2}\right)} \leq$$

$$\leq n \frac{\left(\frac{1}{2}\right)^{F(i-1)+1}}{\frac{1}{2}} \leq \frac{n}{2^{F(i-1)}} = \frac{n}{F(i)}$$

Notiamo che questo è l'unico posto nella dimostrazione in cui utilizziamo la definizione della funzione F.

Riassumendo, per $0 \leq i \leq \log^* n - 1$, ci sono al più $\frac{n}{F(i)}$ nodi nel blocco $B(i)$, ed ogni nodo del blocco $B(i)$ potrà richiedere al più $F(i)$ crediti di tipo (2). Questo implica che il numero totale di crediti di tipo (2) richiesto da nodi in $B(i)$ è limitato superiormente da

$$\frac{n}{F(i)} F(i) \leq n$$

per $0 \leq i \leq \log^* n - 1$. Visto che il numero totale di crediti immagazzinati nel blocco $B(i)$ dalle n operazioni `makeSet` è esattamente n, i crediti immagazzinati nei blocchi dalle `makeSet` saranno del tutto sufficienti a pagare per la visita dei nodi che richiedono crediti di tipo (2). □

Concludiamo il capitolo osservando che i tempi di esecuzione riportati nel Teorema 9.6 non sono i migliori possibili. Tarjan e van Leeuwen [13] hanno infatti dimostrato che una qualsiasi combinazione di union by size o union by rank con path compression, splitting, o halving, è in grado di eseguire una sequenza di n operazioni `makeSet`, m operazioni `find` ed al più $(n-1)$ operazioni `union` in tempo totale $O(n + m\alpha(m+n, n))$. Qui α è una funzione inversa della funzione di Ackermann [1], e presenta una crescita ancora più lenta della funzione $\log^*$ introdotta in questo capitolo. E' stato infine dimostrato che questo è il miglior tempo ottenibile in vari modelli di calcolo, peraltro molto generali [5, 10, 13].

9.5 Problemi

Problema 9.1 Dimostrare che in una sequenza con n operazioni `makeSet` (e quindi n elementi) il numero di `union` può essere al più $(n-1)$.

Problema 9.2 In questo problema consideriamo gli algoritmi di tipo QuickUnion bilanciati in altezza, ovvero gli algoritmi che utilizzano *union by rank* e nessuna forma di compressione dei cammini. Dimostrare che esiste una sequenza di $(n-1)$ `union` in grado di costruire un albero la cui altezza è almeno $\Omega(\log n)$.

Problema 9.3 Sia T un albero costruito con *union by size* senza alcuna forma di compressione dei cammini come descritto nel Paragrafo 9.2.2. Dimostrare che se T ha altezza h, allora deve avere necessariamente almeno 2^h nodi.

Problema 9.4 Utilizzando i risultati del Problema 9.3, dimostrare che alberi QuickUnion bilanciati in cardinalità (ovvero utilizzando *union by size*), senza alcuna forma di compressione dei cammini, sono in grado di supportare operazioni `makeSet` e `union` in tempo costante, e `find` in tempo $O(\log n)$, dove n è il numero totale di operazioni `makeSet` eseguite nella sequenza.

Problema 9.5 Dato un nodo v di un albero union-find, quanti bit sono necessari per memorizzare il valore rank(x)? E per memorizzare size(x)?

Problema 9.6 Consideriamo le tre euristiche di compressione dei cammini descritte nel Paragrafo 9.3. Quanti passi sul cammino delle `find` sono necessari per implementare la *path compression*? E *path splitting* e *path halving*?

Problema 9.7 Considerare i sei algoritmi union-find ottenibili selezionando una tecnica di bilanciamento sulle `union` (*union by rank* oppure *union by size*) ed una tecnica di compressione dei cammini (*path compression*, *path splitting* oppure *path halving*). Quale dei sei algoritmi utilizzeresti in un'implementazione reale? Motivare la risposta.

Problema 9.8 Il Professor De Furbacchionis ha progettato una nuova variante di compressione dei cammini per problemi union-find:

Dopo aver eseguito una `find`(x), ogni nodo nel cammino della `find`, tranne x e la radice, diventa figlio di x, ed x diventa figlio della radice.

Il Professor De Furbacchionis sostiene che la sua variante è efficiente esattamente come *path compression*. Dimostrare che il Professor De Furbacchionis si sbaglia esibendo una sequenza di n `find` e $(n-1)$ `union` che richiede tempo $\Omega(n \log n)$ quando si usa la compressione del Professor De Furbacchionis.

Problema 9.9 Dimostrare che una sequenza di operazioni `union` e `find`, in cui tutte le `union` precedono le `find`, può essere eseguita in tempo proporzionale al numero di `union` e `find`.

Problema 9.10 Sia T un albero radicato, e siano x e y due nodi in T. Il *minimo antenato comune* di x e y in T è il nodo ρ tale che:

(a) ρ è antenato sia di x che di y
(b) tra tutti gli antenati di x ed y, ρ è quello a maggiore profondità, ovvero a distanza più grande dalla radice di T.

Il *problema del minimo antenato comune off-line* è definito nel modo seguente. Dato un albero T ed un insieme $C = \{\langle x, y\rangle\}$ di coppie di nodi in T, vorremmo trovare il minimo antenato comune di ogni coppia in C. Utilizzare strutture dati union-find per risolvere il problema del minimo antenato comune off-line. Qual è il tempo di esecuzione del vostro algoritmo?

9.6 Sommario

In questo capitolo abbiamo analizzato algoritmi efficienti per il problema union-find, che consiste nell'eseguire una sequenza di operazioni `makeSet`, `union` e `find`. Siamo partiti da strutture dati molto semplici, come gli alberi QuickUnion e QuickFind, che però non sembrano garantire prestazioni soddisfacenti. Abbiamo quindi introdotto opportune euristiche di bilanciamento come la *union by size* e la *union by rank*. Tali euristiche, applicate durante le operazioni `union`, ci hanno consentito di migliorare sensibilmente i tempi di esecuzione: abbiamo chiamato le strutture dati ottenute in questo modo alberi QuickUnion e QuickFind bilanciati. Infine, abbiamo presentato euristiche per la compressione dei cammini, come la *path compression*, la *path splitting* e la *path halving*: tali euristiche possono essere applicate durante le operazioni `find`. La combinazione di euristiche di bilanciamento e di compressione dei cammini ci hanno consentito di ottenere algoritmi molto veloci, come dimostrato nel Teorema 9.6. Sembra interessante notare che, nonostante le euristiche proposte siano molto semplici e molto intuitive, la loro analisi si è rilevata molto sofisticata.

9.7 Note bibliografiche

L'euristica di *union by size* è attribuita a McIlroy e Morris, come riportato in [2], mentre la *union by rank* è stata originariamente proposta da Tarjan e van Leeuwen in [13]. Per quanto riguarda invece le euristiche di compressione dei cammini, la *path compression* è stata suggerita da Hopcroft ed Ullman in [9], mentre la *path splitting* e la *path halving* sono state proposte da van Leeuwen e van der Weide in [14, 15]. Una rassegna degli algoritmi per il problema union-find è contenuta in [7]. I principali lower bound noti per il problema union-find sono di Fredman e Saks [5], La Poutré [10] e Tarjan e van Leeuwen [13].

Riferimenti bibliografici

[1] W. Ackermann, "Zum Hilbertshen Aufbau der reelen Zahlen", *Math. Ann.* 99 (1928), pp. 118–133.

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass. 1974.

[3] L. Banachowski, "A complement to Tarjan's result about the lower bound on the complexity of the set union problem", *Information Processing Letters* 11 (1980), pp. 59–65.

[4] M. J. Fischer, "Efficiency of equivalence algorithms", In *Complexity of computer computations*, R. E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 153–168.

[5] M. L. Fredman, M. E. Saks, "The cell probe complexity of dynamic data structures", *Proc. 21st Annual ACM Symposium on Theory of Computing*, 1989, pp. 345–354.

[6] H. N. Gabow, R. E. Tarjan, "A linear time algorithm for a special case of disjoint set union", *Journal of Computer and System Sciences* 30 (1985), pp. 209–221.

[7] Z. Galil, G. F. Italiano, "Data structures and algorithms for disjoint set union problems", *ACM Computing Surveys*, vol. 23, no. 3 (1991), 319–344.

[8] B. A. Galler, M. J. Fischer, "An improved equivalence algorithm", *Communications of the ACM* 7 (1964), pp. 301–303.

[9] J. E. Hopcroft, J. D. Ullman, "Set merging algorithms" *SIAM Journal of Computing* 2 (1973), pp. 294–303.

[10] J. A. La Poutré, "Lower bounds for the union–find and the split–find problem on pointer machines", *Proc. 22nd Annual ACM Symposium on Theory of Computing*, 1990, pp. 34–44.

[11] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm", *Journal of the Association of Computing Machinery* 22 (1975), pp. 215–225.

[12] R. E. Tarjan, "A class of algorithms which require non linear time to maintain disjoint sets", *Journal of. Computer and System Sciences* 18 (1979), pp. 110–127.

[13] R. E. Tarjan, J. van Leeuwen, "Worst-case analysis of set union algorithms", *Journal of the Association of Computing Machinery* 31 (1984), pp. 245–281.
[14] J. van Leeuwen, T. van der Weide, "Alternative path compression techniques" Technical Report RUU-CS-77-3, 1977, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands.
[15] T. van der Weide, *Data structures: an axiomatic approach and the use of binomial trees in developing and analyzing algorithms*. Mathematisch Centrum, Amsterdam, The Netherlands, 1980.

10

Tecniche algoritmiche

Divide et impera.
(Motto latino)

In questo capitolo illustreremo tre tecniche che sono fondamentali nella progettazione di algoritmi: la tecnica *divide et impera* (Paragrafo 10.1), la programmazione dinamica (Paragrafo 10.2), e il metodo goloso o *greedy* (Paragrafo 10.3). La tecnica *divide et impera*, che abbiamo già incontrato nei capitoli precedenti, consiste nel dividere l'istanza del problema in due o più sottoistanze (*divide*), risolvere ricorsivamente il problema sulle sottoistanze e poi ricombinare la soluzione dei sottoproblemi (*impera*) allo scopo di ottenere la soluzione globale del problema originario. Come dovrebbe essere evidente dagli esempi che abbiamo visto finora, questa è tipicamente una tecnica *top-down*: quando risolviamo un problema con la tecnica *divide et impera*, infatti, procediamo dall'alto verso il basso, affrontando immediatamente l'istanza del problema generale, e dividendola via via in istanze più piccole man mano che l'algoritmo procede nella sua esecuzione.

La programmazione dinamica è invece una tecnica che si basa su una filosofia di tipo *bottom-up*. Quando utilizziamo tale tecnica, infatti, consideriamo implicitamente dei sottoproblemi, definiti su opportune istanze di ingresso, e procediamo logicamente dai sottoproblemi più piccoli verso i sottoproblemi più grandi. Tale tecnica si applica tipicamente a problemi in cui si presentano sottoproblemi che non sono del tutto indipendenti, per cui uno stesso sottoproblema può apparire più volte durante la risoluzione del problema originario. La programmazione dinamica utilizza una tabella per memorizzare le soluzioni dei sottoproblemi incontrati: quando si incontrerà di nuovo lo stesso sottoproblema, sarà sufficiente esaminare l'elemento corrispondente della tabella di programmazione dinamica, piuttosto che risolverlo di nuovo.

La tecnica golosa o *greedy* invece si applica a problemi di ottimizzazione in cui ad ogni passo si deve compiere un insieme di scelte: il metodo goloso suggerisce di effettuare le scelte che appaiono più promettenti allo stato attuale delle cose, senza preoccuparsi affatto del futuro.

10.1 Tecnica *divide et impera*

La tecnica *divide et impera* è una tecnica di progettazione di algoritmi che abbiamo già incontrato varie volte nel corso di questo testo. In particolare abbiamo già introdotto tale tecnica fin dall'algoritmo di ricerca binaria illustrato nel Paragrafo 2.4.3 del Capitolo 2, e ne abbiamo viste innumerevoli altre applicazioni. Ricordiamo ad esempio il caso degli algoritmi `mergeSort`, `quickSort` discussi nel Capitolo 4 e l'algoritmo `select` presentato nel Capitolo 5. Vista la sua importanza, ricordiamo che abbiamo addirittura dimostrato un teorema generale per analizzare algoritmi di tipo *divide et impera*: il teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2).

Il principio su cui si basa la tecnica algoritmica *divide et impera* consiste nel dividere i dati di ingresso in due o più sottoinsiemi (*divide*), risolvere ricorsivamente il problema sui sottoinsiemi e poi ricombinare la soluzione dei sottoproblemi (*impera*) per ottenere la soluzione globale del problema originario. Ne consegue che la tecnica *divide et impera* sarà tanto più efficace quanto più sarà conveniente decomporre una instanza in ingresso in sottoistanze e ricombinare efficientemente le loro soluzioni. Ad esempio, nel caso del `mergeSort`, la tecnica *divide et impera* esplicita i due passi cruciali nel modo seguente:

(*Divide*) Dividi l'insieme da ordinare in due sottoinsiemi di cardinalità bilanciata, e ordinali ricorsivamente.
(*Impera*) Fondi due sottoinsiemi ordinati in un unico insieme ordinato.

Anche nel caso del `quickSort`, partizioniamo gli elementi in due sottoinsiemi che vengono ordinati ricorsivamente e che vengono poi ricombinati insieme. Differentemente dal `mergeSort`, tuttavia, in questo caso la parte più complessa è la fase di suddivisione (*divide*):

(*Divide*) Scegli un elemento x della sequenza (il *perno*), partiziona la sequenza in elementi $\leq x$ e in elementi $> x$, e ordina ricorsivamente le due sottosequenze.
(*Impera*) Concatena le sottosequenze ordinate.

Anche se `mergeSort` e `quickSort` sono entrambi esempi di applicazione della tecnica *divide et impera*, differiscono profondamente nel modo in cui partizionano il problema e ricombinano le soluzioni. Il `mergeSort` lavora con una partizione semplice, bilanciata sulla dimensione: la soluzione ricorsiva delle due parti produce due sottoinsiemi ordinati che sono "fusi" in un modo non banale nella fase di ricombinazione. Il `quickSort` lavora invece con una partizione più complicata, basata sui valori degli elementi, ma il passo di ricombinazione è semplice.

Questo ci fa capire che l'applicazione di tale tecnica non può essere completamente automatica, ma spesso richiede un sforzo creativo da parte del progettista di algoritmi nell'individuare tecniche efficienti di decomposizione del problema e di ricombinazione delle soluzioni. Nel resto del paragrafo, vedremo altri esempi di decomposizione e ricombinazione proprie della tecnica *divide et impera*.

10.1.1 Moltiplicazione di interi di grandezza arbitraria

Finora abbiamo sempre immaginato che moltiplicare o sommare due numeri fosse un'operazione eseguibile in tempo costante. Alla fine del Capitolo 1, abbiamo però osservato che se i numeri sono sufficientemente grandi, potrebbero non essere più rappresentabili in una sola parola di memoria di un calcolatore, che ha una lunghezza limitata. In tal caso anche eseguire operazioni elementari su tali numeri potrebbe non essere un'operazione primitiva che richiede tempo $O(1)$. Supponiamo di avere due numeri X ed Y, ciascuno contenente n cifre decimali:

$$X = x_{n-1}x_{n-2}\ldots x_1x_0 = \sum_{i=0}^{n-1} x_i \cdot 10^i, \qquad x_i \in \{0,1,\ldots,9\}$$

$$Y = y_{n-1}y_{n-2}\ldots y_1y_0 = \sum_{i=0}^{n-1} y_i \cdot 10^i, \qquad y_i \in \{0,1,\ldots,9\}$$

In quanto tempo possiamo moltiplicare i due numeri X ed Y? L'algoritmo di moltiplicazione banale, che ci hanno insegnato alla scuola elementare, richiede chiaramente $O(n^2)$ passi. Possiamo fare di meglio? Proviamo ad applicare la tecnica del *divide et impera*.

Per semplicità, supponiamo che n sia una potenza di 2, altrimenti basta considerare opportunamente la parte intera superiore e inferiore dei valori ottenuti nella nostra decomposizione. Un modo naturale di fare *divide* è quello di decomporre X ed Y a metà: le $\frac{n}{2}$ cifre più significative (X_1 ed Y_1) e le $\frac{n}{2}$ cifre meno significative (X_0 ed Y_0). Possiamo quindi scrivere:

$$X = X_1 \cdot 10^{\frac{n}{2}} + X_0$$

$$Y = Y_1 \cdot 10^{\frac{n}{2}} + Y_0$$

La moltiplicazione dei due numeri potrà allora essere calcolata come:

$$\begin{aligned} X \cdot Y &= (X_1 \cdot 10^{\frac{n}{2}} + X_0) \cdot (Y_1 \cdot 10^{\frac{n}{2}} + Y_0) \\ &= (X_1 \cdot Y_1) \cdot 10^n + (X_1 \cdot Y_0 + X_0 \cdot Y_1) \cdot 10^{\frac{n}{2}} + X_0 \cdot Y_0 \end{aligned} \tag{10.1}$$

Notiamo che moltiplicare un numero decimale per 10^k equivale ad eseguire uno shift delle sue cifre di k posizioni a sinistra, e che due numeri di k cifre decimali possono essere sommati in tempo totale $O(k)$. Di conseguenza, la moltiplicazione di due numeri con n cifre decimali può essere realizzata mediante quattro moltiplicazioni di numeri con $\frac{n}{2}$ cifre decimali più un tempo totale di $O(n)$ dovuto alle somme ed agli shift. Indicando con $T(n)$ il tempo richiesto per moltiplicare due numeri con n cifre decimali, otteniamo la relazione di ricorrenza:

$$T(n) = \begin{cases} 4T(n/2) + O(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases} \tag{10.2}$$

che in base al teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2) ammette come soluzione $T(n) = O(n^{\log_2 4}) = O(n^2)$. Ma questo è esattamente lo stesso tempo di esecuzione dell'algoritmo di moltiplicazione che ci hanno insegnato alla scuola elementare, e di conseguenza non abbiamo guadagnato nulla!

Da questa analisi preliminare emerge che, nell'applicare la tecnica *divide et impera*, dopo aver eseguito il passo di *divide*, e aver decomposto il problema in sottoproblemi di dimensione più piccola, non siamo riusciti a trarre vantaggio dalla fase di *impera* ricombinando più efficientemente le soluzioni di tali sottoproblemi. Se vogliamo ottenere un tempo di esecuzione migliore di $O(n^2)$, sembra evidente che dobbiamo riuscire a diminuire il numero di moltiplicazioni richieste tra numeri di $\frac{n}{2}$ cifre decimali: fintantoché il coefficiente moltiplicativo della relazione di ricorrenza (10.2) è 4, in base al Teorema 2.1, la soluzione di tale relazione di ricorrenza sarà infatti $O(n^2)$. Vediamo ora come effettuare il prodotto $X \cdot Y$ più efficientemente. A questo scopo, consideriamo

$$P_1 = (X_1 + X_0) \cdot (Y_1 + Y_0) = X_1 \cdot Y_1 + X_1 \cdot Y_0 + X_0 \cdot Y_1 + X_0 \cdot Y_0$$

e osserviamo che il termine $(X_1 \cdot Y_0 + X_0 \cdot Y_1)$ dell'equazione (10.1) può essere ottenuto come:

$$(X_1 \cdot Y_0 + X_0 \cdot Y_1) = P_1 - X_1 \cdot Y_1 - X_0 \cdot Y_0$$

Quindi se abbiamo a disposizione i tre prodotti

$$\begin{aligned} P_1 &= (X_1 + X_0) \cdot (Y_1 + Y_0) \\ P_2 &= (X_1 \cdot Y_1) \\ P_3 &= (X_0 \cdot Y_0) \end{aligned}$$

allora la moltiplicazione dei due numeri X e Y potrà essere calcolata come:

$$\begin{aligned} X \cdot Y &= (X_1 \cdot 10^{\frac{n}{2}} + X_0) \cdot (Y_1 \cdot 10^{\frac{n}{2}} + Y_0) \\ &= (X_1 \cdot Y_1) \cdot 10^n + (X_1 \cdot Y_0 + X_0 \cdot Y_1) \cdot 10^{\frac{n}{2}} + X_0 \cdot Y_0 \\ &= P_2 \cdot 10^n + (P_1 - P_2 - P_3) \cdot 10^{\frac{n}{2}} + P_3 \end{aligned}$$

Di conseguenza, la moltiplicazione di due numeri con n cifre decimali può essere realizzata mediante tre moltiplicazioni di numeri con $\frac{n}{2}$ cifre decimali più un tempo totale di $O(n)$ dovuto alle somme, alle sottrazioni ed agli shift. Indicando sempre con $T(n)$ il tempo richiesto per moltiplicare due numeri con n cifre decimali, questa volta otteniamo la relazione di ricorrenza:

$$T(n) = \begin{cases} 3T(n/2) + O(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

che in base al teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2) ammette come soluzione $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$. Questa volta, dopo aver eseguito il passo di *divide*, siamo riusciti anche a trarre vantaggio dalla fase di *impera*.

10.1.2 Moltiplicazione tra matrici

Siano A e B due matrici $n \times n$, e sia C il loro prodotto. Anche in questo caso supponiamo per semplicità che n sia una potenza di 2, altrimenti basta considerare opportunamente la parte intera superiore e inferiore dei valori ottenuti nella nostra decomposizione. L'algoritmo classico per calcolare il prodotto di due matrici richiede tempo $O(n^3)$:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

Raffinando ulteriormente l'approccio utilizzato nella moltiplicazione di interi a dimensioni elevate, è possibile ottenere un algoritmo asintoticamente più efficiente. Consideriamo ora la seguente decomposizione delle matrici A e B:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \qquad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

dove $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, $A_{2,2}$ $B_{1,1}$, $B_{1,2}$, $B_{2,1}$, e $B_{2,2}$ sono tutte matrici di dimensione $\frac{n}{2} \times \frac{n}{2}$. Se calcoliamo la matrice C nel modo seguente

$$C = \begin{pmatrix} A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} & A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} & A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2} \end{pmatrix}$$

eseguiremo un totale di quattro somme e di otto prodotti tra matrici di dimensione $\frac{n}{2} \times \frac{n}{2}$. Ricordiamo che due matrici $n \times n$ possono essere sommate in tempo $O(n^2)$. Indicando con $T(n)$ il tempo necessario a moltiplicare due matrici $n \times n$, otteniamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} 8T(n/2) + O(n^2) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases} \qquad (10.3)$$

La relazione (10.3), in base al teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2) ammette come soluzione $T(n) = O(n^{\log_2 8}) = O(n^3)$, ovvero lo stesso tempo di esecuzione dell'algoritmo banale di moltiplicazione tra matrici. Ancora una volta, abbiamo eseguito il passo di *divide*, ma non siamo riusciti a trarre vantaggio dalla fase di *impera*! In maniera analoga al prodotto tra numeri con n cifre, per migliorare il tempo di esecuzione, dobbiamo riuscire a diminuire il numero di moltiplicazioni richieste tra matrici $\frac{n}{2} \times \frac{n}{2}$. Definiamo le seguenti matrici:

$$\begin{aligned} M_1 &= (A_{2,1} + A_{2,2} - A_{1,1}) \cdot (B_{2,2} - B_{1,2} + B_{1,1}) \\ M_2 &= A_{1,1} \cdot B_{1,1} \\ M_3 &= A_{1,2} \cdot B_{2,1} \\ M_4 &= (A_{1,1} - A_{2,1}) \cdot (B_{2,2} - B_{1,1}) \\ M_5 &= (A_{2,1} + A_{2,2}) \cdot (B_{1,2} - B_{1,1}) \\ M_6 &= (A_{1,2} - A_{2,1} + A_{1,1} - A_{2,2}) \cdot B_{2,2} \\ M_7 &= A_{2,2} \cdot (B_{1,1} + B_{2,2} - B_{1,2} - B_{2,1}) \end{aligned}$$

A questo punto è possibile verificare che la matrice prodotto C verifica la

$$C = \begin{pmatrix} M_2 + M_3 & M_1 + M_2 + M_5 + M_6 \\ M_1 + M_2 + M_4 - M_7 & M_1 + M_2 + M_4 + M_5 \end{pmatrix}$$

È quindi possibile moltiplicare due matrici $n \times n$ eseguendo sette moltiplicazioni tra matrici $\frac{n}{2} \times \frac{n}{2}$ e ventiquattro somme e sottrazioni tra matrici $\frac{n}{2} \times \frac{n}{2}$. Indicando quindi con $T(n)$ il tempo richiesto per moltiplicare due matrici $n \times n$, otteniamo la relazione di ricorrenza:

$$T(n) = \begin{cases} 7T(n/2) + O(n^2) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

che in base al teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2) ammette come soluzione $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$. È quindi possibile moltiplicare due matrici $n \times n$ in meno di $O(n^3)$!

10.2 Programmazione dinamica

In questo paragrafo presenteremo una nuova tecnica algoritmica, la *programmazione dinamica*. In realtà, notiamo che abbiamo già introdotto in maniera informale questa tecnica nel Capitolo 1, e più precisamente nel Paragrafo 1.4. In quella sede, abbiamo infatti proposto l'algoritmo `fibonacci3`, un algoritmo iterativo per calcolare l'n-esimo numero di Fibonacci, che rivisitiamo nella Figura 10.1 per completezza.

```
    algoritmo fibonacci3(intero n) → intero
1.    F = array di (n + 1) interi
2.    F[0] ← 0
3.    F[1] ← 1
4.    for i = 2 to n do
5.        F[i] ← F[i − 1] + F[i − 2]
6.    return F[n]
```

Figura 10.1 Algoritmo `fibonacci3` per il calcolo dell'n-esimo numero di Fibonacci.

In pratica, l'algoritmo `fibonacci3` utilizza una tabella (array) F di dimensione $(n + 1)$. Ricordiamo che lo scopo della tabella è fondamentalmente quello di memorizzare le soluzioni dei sottoproblemi incontrati: quando si incontrerà di nuovo lo stesso problema, sarà sufficiente esaminare un elemento della tabella di programmazione dinamica, piuttosto che risolvere ancora una volta lo stesso sottoproblema. La tabella viene "programmata dinamicamente" (da cui il nome programmazione dinamica) mediante i seguenti passi:

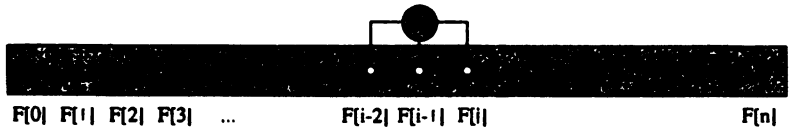


Figura 10.2 Tabella utilizzata dall'algoritmo `fibonacci3` per il calcolo dell'n-mo numero di Fibonacci.

(1) Identifichiamo innanzitutto dei sottoproblemi del problema originario: per $0 \leq i \leq n$, il sottoproblema i-esimo consiste nel calcolo dell'i-esimo numero di Fibonacci. Utilizziamo la tabella per memorizzare le soluzioni di tali sottoproblemi.
(2) Definiamo innanzitutto il valore iniziale di alcuni elementi della tabella: $F[0] = 0$ ed $F[1] = 1$.
(3) Al generico passo i-esimo, $i \geq 2$, avanziamo sulla tabella calcolando il valore dell'elemento i-esimo in base al valore degli elementi precedentemente calcolati: $F[i] \leftarrow F[i-1] + F[i-2]$.
(4) Restituiamo il valore memorizzato in un particolare elemento della tabella, ovvero $F[n]$.

La tabella utilizzata dall'algoritmo `fibonacci3` è illustrata graficamente in Figura 10.2. A questo punto facciamo alcune osservazioni. Innanzitutto possiamo affermare che la tecnica di programmazione dinamica precede in modo *bottom-up*, ovvero dal basso verso l'alto. In altre parole, utilizzando tale tecnica considereremo implicitamente dei sottoproblemi, definiti su opportune istanze di ingresso. Nel caso dei numeri di Fibonacci, ad esempio, i sottoproblemi sono definiti come il calcolo dell'i-esimo numero di Fibonacci, $i \geq 0$. Cominceremo a lavorare dai sottoproblemi più semplici, come ad esempio quelli corrispondenti a istanze di ingresso molto piccole ($i = 0$ ed $i = 1$ nel caso dei numeri di Fibonacci). Combineremo poi le soluzioni di questi sottoproblemi semplici, riempiendo ed utilizzando in maniera opportuna la tabella dei risultati intermedi, fino ad ottenere la soluzione di sottoproblemi definiti su istanze di ingresso più grandi (nel caso dei numeri di Fibonacci $F_i = F_{i-1} + F_{i-2}$). Ci fermeremo quando avremo raggiunto la soluzione al problema originario: nel caso dell'esempio di Fibonacci, questo equivale all'aver calcolato l'elemento di posizione n della tabella, corrispondente a F_n. Notiamo che questo modo di ottenere la soluzione di un problema è completamente diverso dalla tecnica *divide et impera* descritta nel Paragrafo 10.1, che invece precede in modo *top-down*, ovvero dall'alto verso il basso. Quando risolviamo un problema con la tecnica *divide et impera*, infatti, affrontiamo immediatamente l'istanza del problema generale, dividendolo via via in istanze più piccole man mano che l'algoritmo procede nella sua esecuzione.

In maniera del tutto generale, potremmo descrivere la tecnica di programmazione dinamica nel modo seguente:

(1) Identifichiamo dei sottoproblemi del problema originario, ed utilizziamo una tabella per memorizzare i risultati intermedi dei sottoproblemi.

(2) All'inizio, definiamo i valori iniziali di alcuni elementi della tabella, corrispondenti ai sottoproblemi più semplici.
(3) Al generico passo, avanziamo in modo opportuno sulla tabella calcolando il valore della soluzione di un sottoproblema (corrispondente ad un dato elemento della tabella) in base alla soluzione dei sottoproblemi precedentemente risolti (corrispondenti ad elementi della tabella precedentemente calcolati).
(4) Alla fine, restituiamo la soluzione del problema originario, che è stata memorizzata in un particolare elemento della tabella.

Vedremo ora due ulteriori esempi di utilizzo della tecnica di programmazione dinamica.

10.2.1 La distanza tra due stringhe di caratteri

Date due stringhe x ed y, desideriamo calcolare la "distanza" tra x ed y, misurata opportunamente in termini delle differenze tra le due stringhe. Ad esempio, potremmo essere interessati a trasformare la stringa x nella stringa y. Questo è un problema che nasce nelle correzioni ortografiche automatiche (*spell checking*) dei documenti: in tal caso, i correttori ortografici (*spell checker*) non riconoscono soltanto un vocabolo errato in un documento, ma propongono anche delle possibili correzioni o sostituzioni: la scelta di queste sostituzioni si basa sulla ricerca di tutte le pamle che hanno una distanza limitata da quella scritta in modo errato.

Cerchiamo di definire meglio cosa intendiamo per "distanza" tra due stringhe. Siano $X = x_1 \cdot x_2 \cdot \cdots \cdot x_m$ e $Y = y_1 \cdot y_2 \cdot \cdots \cdot y_n$ due stringhe di caratteri, di lunghezza rispettivamente m e n. Possiamo definire il costo della trasformazione di X in Y come il numero di cambiamenti che dobbiamo apportare alla stringa X per ottenere Y. I cambiamenti (od operazioni) che possiamo compiere mentre stiamo esaminando una stringa sono:

`inserisci`(a): Inserisci il carattere a nella posizione corrente della stringa.
`cancella`(a): Cancella il carattere a dalla posizione corrente della stringa.
`sostituisci`(a, b): Sostituisci il carattere a con il carattere b nella posizione corrente della stringa.

Se assumiamo che il costo di ognuna di queste operazioni sia 1, possiamo definire il *costo della trasformazione tra X e Y* come la somma di tutti i costi che abbiamo pagato per compiere le operazioni di trasformazione da X in Y. Potremmo definire quindi la *distanza tra due stringhe X ed Y* come il costo minimo di trasformazione di X in Y.

Esempio 10.1 Supponiamo di voler trasformare la stringa X = `RISOTTO` nella stringa Y = `PRESTO`. La soluzione più elementare è cancellare tutti i caratteri di X e poi aggiungere tutti quelli di Y, come illustrato in Tabella 10.1. In questo modo il costo totale per la trasformazione di X in Y è 13, dato da 7 cancellazioni

Azione	Costo	Stringa ottenuta
Cancello R	1	|*ISOTTO*
Cancello I	1	|*SOTTO*
Cancello S	1	|*OTTO*
Cancello O	1	|*TTO*
Cancello T	1	|*TO*
Cancello T	1	|*O*
Cancello O	1	|
Inserisco P	1	P|
Inserisco R	1	PR|
Inserisco E	1	PRE|
Inserisco S	1	PRES|
Inserisco T	1	PREST|
Inserisco O	1	PRESTO|

Tabella 10.1 Possibile trasformazione della stringa `RISOTTO` nella stringa `PRESTO`, il cui costo totale è pari a 13.

seguite da 6 inserimenti. Se desideriamo ottenere un costo totale inferiore a 13, potremmo analizzare la stringa `PRESTO`, carattere per carattere, e compiere delle opportune modifiche, come illustrato nella Tabella 10.2. Questa trasformazione riduce il costo da 13 a 4. È possibile verificare che non esiste una trasformazione da `RISOTTO` a `PRESTO` di costo inferiore a 4, e quindi la distanza tra `RISOTTO` e `PRESTO` è proprio 4. □

Date due stringhe X ed Y, desideriamo progettare un algoritmo per calcolare la minima distanza tra X e Y, ovvero il numero minimo di operazioni sufficienti a trasformare X in Y. Indichiamo con $\delta(X, Y)$ la distanza tra le stringhe X ed Y. Data una stringa $X = x_1 \cdot x_2 \cdot \cdots \cdot x_m$, per $0 \leq i \leq m$, definiamo il *prefisso di X fino al carattere i-esimo* come la stringa $X_i = x_1 \cdot x_2 \cdot \cdots \cdot x_i$ se $i \geq 1$, e come la stringa vuota $X_0 = \emptyset$ se $i = 0$. Anzichè risolvere il problema generale $\mathcal{P}$, ovvero trovare la distanza $\delta(X, Y)$ tra la stringa X e la stringa Y, proviamo a considerare i sottoproblemi $\mathcal{P}(i, j)$, ovvero trovare la distanza $\delta(X_i, Y_j)$ tra il prefisso X_i ed il prefisso Y_j. Notiamo che:

(1) Alcuni sottoproblemi $\mathcal{P}(i, j)$ sono particolarmente semplici. Ad esempio la soluzione del sottoproblema $\mathcal{P}(0, j)$ consiste nel partire dalla stringa vuota $X_0 = \emptyset$ e nell'inserire uno dopo l'altro i j caratteri di Y_j. La distanza $\delta(X_0, Y_j)$ è dunque data da j. In modo completamente analogo, la soluzione del sottoproblema $\mathcal{P}(i, 0)$ consiste nel partire dalla stringa X_i contenente i caratteri, e nel cancellare uno dopo l'altro gli i caratteri di X_i per ottenere la stringa vuota Y_0. La distanza $\delta(X_i, Y_0)$ è dunque data da i.

Azione	Costo	Stringa ottenuta
Inserisco P	1	P|*RISOTTO*
Mantengo R	0	PR|*ISOTTO*
Sostituisco I con E	1	PRE|*SOTTO*
Mantengo S	0	PRES|*OTTO*
Cancello O	1	PRES|*TTO*
Mantengo T	0	PREST|*TO*
Cancello T	1	PREST|*O*
Mantengo O	0	PRESTO|

Tabella 10.2 Possibile trasformazione della stringa RISOTTO nella stringa PRESTO, il cui costo totale è pari a 4.

(2) $\mathcal{P} = \mathcal{P}(m, n)$, ovvero $\delta(X, Y) = \delta(X_m, Y_n)$.

Queste proprietà ci suggeriscono di provare a progettare un algoritmo di programmazione dinamica per il problema della distanza tra stringhe. A tale proposito, osserviamo che finora abbiamo già individuato tre punti importanti.

1. *I sottoproblemi da risolvere* $\mathcal{P}(i, j)$. Questa scelta ci consiglia di utilizzare una tabella bidimensionale D, ovvero una matrice $m \times n$, per memorizzare i risultati intermedi. In particolare, sembra opportuno memorizzare in $D[i, j]$ la soluzione al problema $\mathcal{P}(i, j)$, ovvero la distanza minima $\delta(X_i, Y_j)$ tra il prefisso X_i ed il prefisso Y_j.
2. *I valori iniziali di alcuni elementi della tabella*, corrispondenti ai sottoproblemi più semplici. In particolare, per qualsiasi j, $0 \leq j \leq n$, avremo che $D[0, j] = j$, e per qualsiasi i, $0 \leq i \leq m$, avremo che $D[i, 0] = i$. Questo implica che la colonna e la riga 0 della matrice D sono facilmente calcolabili.
3. *Il punto in cui è memorizzata nella tabella D la soluzione del problema originale.* In particolare, la soluzione del problema $\mathcal{P} = \mathcal{P}(m, n)$ sarà disponibile nell'elemento $D[m, n]$.

L'unica tessera che sembra ancora mancare dal nostro mosaico è come realizzare il generico passo di avanzamento nella tabella di programmazione dinamica, ovvero come calcolare il valore della soluzione del sottoproblema $\mathcal{P}(i, j)$ in funzione della soluzione dei sottoproblemi precedentemente risolti. Notiamo che ci sono varie possibilità, a seconda del valore dei caratteri x_i ed y_j. In particolare, se $x_i = y_j$, allora il minimo costo per trasformare X_i in Y_j sarà dato dal minimo costo per trasformare X_{i-1} in Y_{j-1}. In tal caso, avremo dunque che $D[i, j] = D[i-1, j-1]$. Se invece $x_i \neq y_j$, dovremo distinguire in base all'ultima operazione utilizzata per trasformare il prefisso X_i nel prefisso Y_j in una sequenza ottima di operazioni. In maggior dettaglio, se l'ultima operazione è una:

`inserisci`(y_j): Allora il costo minimo per trasformare X_i in Y_j sarà induttivamente dato dal costo minimo per trasformare X_i in Y_{j-1}, più 1 per l'inserimento del carattere y_j. In tal caso, avremo dunque che

$$D[i, j] = D[i, j-1] + 1$$

`cancella`(x_i): Allora il costo minimo per trasformare X_i in Y_j sarà induttivamente dato dal costo minimo per trasformare X_{i-1} in Y_j, più 1 per la cancellazione del carattere x_i. In tal caso, avremo dunque che

$$D[i, j] = D[i-1, j] + 1$$

`sostituisci`(x_i, y_j): Allora il costo minimo per trasformare X_i in Y_j sarà induttivamente dato dal costo minimo per trasformare X_{i-1} in Y_{j-1}, più 1 per la sostituzione del carattere x_i in y_j. In tal caso, avremo dunque che

$$D[i, j] = D[i-1, j-1] + 1$$

Osserviamo che le precedenti relazioni di ricorrenza assumono che l'ultima operazione utilizzata per trasformare il prefisso X_i nel prefisso Y_j in una sequenza ottima di operazioni sia nota a priori. Questa ipotesi ovviamente non è verificata, dato che la sequenza ottima è proprio ciò che vogliamo calcolare. Tuttavia, sono possibili solo tre tipi di operazioni, inserimento, cancellazione o sostituzione dell'ultimo carattere, e la sequenza ottima deve necessariamente utilizzare una di queste tre operazioni. Quindi, per trovare la scelta effettuata dalla sequenza ottima, possiamo calcolare i tre valori relativi all'ipotesi di inserimento, cancellazione e sostituzione dell'ultimo carattere, e scegliere il valore migliore tra i tre. Queste considerazioni ci conducono alla seguente relazione, che specifica come calcolare il valore della soluzione ottima del sottoproblema $\mathcal{P}(i, j)$ in funzione della soluzione di alcuni dei sottoproblemi precedentemente risolti:

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{se } x_i = y_j \\ 1 + \min\{D[i, j-1], D[i-1, j], D[i-1, j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

Osserviamo che questa relazione ci fornisce proprio l'informazione che ci mancava per definire completamente il passo di avanzamento nella tabella di programmazione dinamica per il nostro problema. Prima di specificare l'algoritmo, evidenziamone le principali differenze con l'algoritmo `fibonacci3` del Capitolo 1.

- Nel caso dei numeri di Fibonacci, i sottoproblemi considerati generano una tabella unidimensionale, ovvero un array. Nel caso della distanza tra stringhe, i sottoproblemi sono caratterizzati da due indici, e quindi generano una tabella bidimensionale, ovvero una matrice.
- Nel caso dei numeri di Fibonacci, i valori da inizializzare nell'array sono soltanto due: $F[0]$ e $F[1]$. Nel caso della distanza tra stringhe, i valori da inizializzare nella matrice sono $(m + n + 1)$: per $0 \leq j \leq n$, inizializzeremo la riga 0 con $D[0, j] = j$, e per $0 \leq i \leq m$, inizializzeremo la colonna 0 con $D[i, 0] = i$.

```
    algoritmo distanzaStringhe(stringa X, stringa Y) → intero
1.    matrice D di (m + 1) × (n + 1) interi
2.    for i = 0 to m do D[i, 0] ← i
3.    for j = 1 to n do D[0, j] ← j
4.    for i = 1 to m do
5.       for j = 1 to n do
6.          if (x_i ≠ y_j) then
7.             D[i, j] ← 1 + min{D[i, j − 1], D[i − 1, j], D[i − 1, j − 1]}
8.          else D[i, j] ← D[i − 1, j − 1]
9.    return D[m, n]
```

Figura 10.3 Algoritmo di calcolo della distanza tra due stringhe.

- Nel passo generico, nel caso dei numeri di Fibonacci per calcolare il valore attuale ($F[i]$) è sufficiente disporre dei due valori precedenti ($F[i-1]$ e $F[i-2]$). Quindi l'array potrà essere riempito da sinistra a destra, secondo i valori crescenti dell'indice. Nel caso della distanza tra stringhe, per calcolare il valore attuale ($D[i,j]$) è sufficiente disporre dei tre valori $D[i-1,j-1]$, $D[i-1,j]$ e $D[i,j-1]$. Per assicurarci che tali valori siano disponibili quando servono, possiamo indifferentemente riempire la matrice D per righe oppure per colonne.
- Alla fine, la soluzione del problema dei numeri di Fibonacci si trova nell'ultimo elemento dell'array ($F[n]$). La soluzione del problema della distanza tra stringhe si trova all'incrocio tra l'ultima riga e l'ultima colonna della matrice ($D[m,n]$).

Siamo ora in grado di presentare l'algoritmo `distanzaStringhe`, illustrato nella Figura 10.3.

Esempio 10.2 Nella tabella 10.3 illustriamo la matrice di programmazione dinamica costruita dall'algoritmo `distanzaStringhe` sulle stringhe `RISOTTO` e `PRESTO`. Nella tabella viene indicata in grassetto una sequenza che consente di ottenere la distanza tra le due stringhe, da cui è possibile ricavare una sequenza di operazioni di costo minimo in grado di trasformare la stringa `RISOTTO` nella stringa `PRESTO`. Notiamo che alla riga 5, corrispondente alla prima `T` di `RISOTTO` abbiamo due possibilità: provenendo da $D[4,4]$ con un costo di 3, possiamo cancellare subito la `T` andando in $D[5,4]$ con un costo di 4, oppure tenere la `T` andando in $D[5,5]$ con un costo di 3; in questo secondo caso, quando andremo in $D[6,5]$ dovremo cancellare la seconda `T`. È facile verificare che la sequenza corrispondente a questa seconda scelta è esattamente quella già illustrata nella Tabella 10.2. □

Il seguente teorema caratterizza la complessità di spazio e di tempo dell'algoritmo `distanzaStringhe`.

Teorema 10.1 *L'algoritmo* `distanzaStringhe` *richiede un tempo di esecuzione $O(mn)$, dove X ed Y sono le due stringhe in ingresso, con $|X| = m$ e $|Y| = n$. L'occupazione di memoria è $O(mn)$.*

		P	R	E	S	T	O
	0	**1**	2	3	4	5	6
R	1	1	**1**	2	3	4	5
I	2	2	2	**2**	3	4	5
S	3	3	3	3	**2**	3	4
O	4	4	4	4	**3**	3	3
T	5	5	5	5	**4**	**3**	4
T	6	6	6	6	5	**4**	4
O	7	7	7	7	6	5	**4**

Tabella 10.3 Tabella di programmazione dinamica costruita dall'algoritmo `distanzaStringhe` sulle stringhe `RISOTTO` e `PRESTO`. In grassetto vengono indicate due sequenze di operazioni che consentono di ottenere la distanza tra le due stringhe.

Dimostrazione. Consideriamo l'algoritmo `distanzaStringhe` illustrato nella Figura 10.3. La fase di inizializzazione (righe 2–3) può essere chiaramente implementata in tempo $O(m+n)$, mentre invece il ciclo `for` esterno (righe 4–8) implica l'esame dell'intera matrice D, e quindi un tempo totale $O(mn)$. Per quanto riguarda l'occupazione di memoria, la matrice D ha dimensione $(m+1)\cdot(n+1)$ e quindi anche lo spazio totale richiesto dall'algoritmo è $O(mn)$. □

Concludiamo la nostra analisi sul problema della distanza tra due stringhe, osservando che lo spazio utilizzato dall'algoritmo può essere sensibilmente ridotto. Possiamo infatti fare un ragionamento molto simile a quello fatto nel caso dei numeri di Fibonacci nel Capitolo 1, in cui nel passare dall'algoritmo `fibonacci3` all'algoritmo `fibonacci4`, abbiamo osservato che per calcolare l'i-esimo numero di Fibonacci non serviva avere a disposizione tutto l'array, ma soltanto i due valori precedenti. Anche nel caso della distanza tra le stringhe, non è necessario mantenere tutta la matrice D: per calcolare un generico elemento $D[i,j]$, è sufficiente avere a disposizione la riga $(i-1)$, e la colonna $(j-1)$: in particolare, è possibile riempire la matrice D, riga per riga o colonna per colonna, mantenendo durante tutta l'esecuzione dell'algoritmo solamente una riga ed una colonna. Questa semplice osservazione è in grado di ridurre lo spazio richiesto dall'algoritmo `distanzaStringhe` da $O(mn)$ a $O(m+n)$.

10.2.2 Associatività dei prodotto tra matrici

Introduciamo il problema con un esempio concreto. Supponiamo di dover eseguire il prodotto di tre matrici

$$M = M_1 \cdot M_2 \cdot M_3$$

dove M_1 è una matrice 2×1000 (con 2 righe e 1000 colonne), M_2 è una matrice 1000×3 (con 1000 righe e 3 colonne), e M_3 è una matrice 3×100 (con 3 righe e 100 colonne). Notiamo che per la proprietà associativa delle matrici, ci sono due modi di calcolare il prodotto M delle tre matrici:

$$M = (M_1 \cdot M_2) \cdot M_3 \tag{10.4}$$

$$M = M_1 \cdot (M_2 \cdot M_3) \tag{10.5}$$

Nel metodo suggerito dalla (10.4) eseguiamo prima il prodotto della matrice M_1 per M_2, e poi moltiplichiamo il risultato per M_3. Nel metodo suggerito dalla (10.5), invece, eseguiamo prima il prodotto della matrice M_2 per M_3, e poi moltiplichiamo M_1 per il risultato ottenuto. Quali di questi due modi di calcolare M è il più conveniente? Assumiamo di utilizzare l'algoritmo più semplice per eseguire il prodotto tra due matrici, senza ricorrere al metodo sofisticato presentato nel Paragrafo 10.1. In altri termini, data la matrice $A = \{a_{i,k}\}$, di dimensione $\ell_1 \times \ell_2$, e la matrice $B = \{b_{k,j}\}$, di dimensione $\ell_2 \times \ell_3$, il generico elemento $c_{i,j}$ della matrice prodotto $C = A \cdot B$, di dimensione $\ell_1 \times \ell_3$, viene calcolato nel modo seguente:

$$c_{i,j} = \sum_{k=1}^{\ell_2} a_{i,k}\, b_{k,j} \tag{10.6}$$

Per calcolare il costo di questo prodotto, consideriamo come operazione elementare la moltiplicazione tra elementi delle matrici: il prodotto $C = A \cdot B$ richiede quindi un totale di $\ell_1 \cdot \ell_2 \cdot \ell_3$ operazioni elementari.

Consideriamo ora il costo di calcolare M nei due casi (10.4) e (10.5). Nel caso (10.4), il costo totale sarà dato da $(2 \cdot 1000 \cdot 3) + (2 \cdot 3 \cdot 100) = 6600$, mentre nel caso (10.5), il costo totale sarà dato da $(1000 \cdot 3 \cdot 100) + (2 \cdot 1000 \cdot 100) = 500000$. Perché il primo metodo costa molto meno? Il punto cruciale è che nel caso (10.4), la prima moltiplicazione $(M_1 \cdot M_2)$ paga un costo di $(2 \cdot 1000 \cdot 3) = 6000$ operazioni e produce una matrice molto piccola (2×3), mentre nel caso (10.4), la prima moltiplicazione $(M_2 \cdot M_3)$ non solo paga un costo più elevato ($(1000 \cdot 3 \cdot 100) = 300000$ operazioni) ma restituisce una matrice molto grande (1000×100). Questo fa sì che il primo metodo esegua soltanto 6600 operazioni elementari in totale al posto delle 500000 operazioni del secondo metodo.

Estendiamo ora il problema al caso più generale. Supponiamo di avere $n \geq 3$ matrici da moltiplicare

$$M = M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

dove, per $1 \leq i \leq n$, M_i è una matrice di dimensione $\ell_i \times \ell_{i+1}$. Qual è il modo migliore di associare le matrici $M_1, M_2, \ldots, M_n$ nel prodotto al fine di eseguire il numero minimo di operazioni?

Se vogliamo applicare la tecnica di programmazione dinamica a questo problema, dobbiamo per prima cosa identificare i sottoproblemi da risolvere. Sembrerebbe naturale considerare come sottoproblema $\mathcal{P}(i,j)$ quello della moltiplicazione delle $(j - i + 1)$ matrici

$$M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$$

per $i < j$. Dato che il problema originario considera

$$M = M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

avremo $\mathcal{P} = \mathcal{P}(1,n)$. Osserviamo inoltre che alcuni di questi sottoproblemi sono molto semplici da risolvere. In particolare, per $1 \leq i \leq n$, il problema $\mathcal{P}(i,i)$ è banale: per calcolare la matrice M_i non serve alcuna operazione, e quindi il costo ottimo in tal caso è dato da 0.

Ancora una volta, abbiamo già individuato gran parte degli ingredienti che sono necessari per applicare la tecnica di programmazione dinamica. In particolare:

1. *I sottoproblemi da risolvere* $\mathcal{P}(i,j)$. Anche in questo caso questa scelta ci consiglia di utilizzare una tabella bidimensionale, ovvero una matrice, per memorizzare i risultati intermedi. In particolare, per $i < j$ definiamo con $C[i,j]$ il costo ottimo di moltiplicare le matrici $M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$, ovvero la soluzione ottima per il problema $\mathcal{P}(i,j)$. Notiamo che, a differenza della matrice D del problema della distanza tra due stringhe, ora gli elementi $C[i,j]$ della matrice C sono significativi solo nel caso in cui $i \leq j$. In altre parole, C sarà una matrice triangolare superiore.
2. *I valori iniziali di alcuni elementi della tabella*, corrispondenti ai sottoproblemi più semplici. In particolare, per qualsiasi i, $1 \leq i \leq n$, avremo che $C[i,i] = 0$. Questo implica che la diagonale principale della matrice C è facilmente calcolabile.
3. *Il punto in cui è memorizzata nella tabella C la soluzione del problema originale*. In particolare, la soluzione del problema $\mathcal{P} = \mathcal{P}(1,n)$ sarà disponibile nell'elemento $C[1,n]$.

L'elemento ancora mancante è come realizzare il generico passo di avanzamento nella tabella di programmazione dinamica, ovvero come calcolare il valore della soluzione del sottoproblema $\mathcal{P}(i,j)$ in funzione della soluzione di sottoproblemi precedentemente risolti. Per definire questo, proviamo a seguire un ragionamento simile a quello effettuato nel caso della distanza tra due stringhe.

Assumiamo quindi che $i < j$. Sia r un intero tale che $i \leq r \leq j - 1$. Se qualcuno ci dicesse che il costo ottimo di moltiplicare $M_i \cdot M_{i+1} \cdot \ldots M_j$ si ottiene associando tutte le matrici $M_i \cdot \ldots \cdot M_r$ e tutte le matrici $M_{r+1} \cdot \ldots \cdot M_j$, allora il modo più efficiente di calcolare il prodotto $M_i \cdot M_{i+1} \cdot \ldots M_j$ si otterrebbe eseguendo $(M_i \cdot \ldots \cdot M_r) \cdot (M_{r+1} \cdot \ldots \cdot M_j)$. Questo implica che

$$C[i,j] = C[i,r] + C[r+1,j] + \ell_i \cdot \ell_{r+1} \cdot \ell_j$$

dove il terzo addendo rappresenta il costo di moltiplicare le due matrici $(M_i \cdot \ldots \cdot M_r)$ e $(M_{r+1} \cdot \ldots \cdot M_j)$, ottenute dalla suddivisione del problema. Esattamente come nel caso della distanza tra due stringhe, dato che nessuno ci garantisce che sia proprio r la migliore scelta possibile, dobbiamo provare tutti i possibili valori di r, $i \leq r \leq j - 1$, per ottenere il valore migliore:

$$C[i,j] = \min_{i \leq r \leq j-1} \{C[i,r] + C[r+1,j] + \ell_i \cdot \ell_{r+1} \cdot \ell_j\} \tag{10.7}$$

Questa era esattamente l'informazione che ci mancava per specificare completamente il passo di avanzamento nella tabella di programmazione dinamica per il nostro problema. Per calcolare il valore $C[i,j]$ sarà quindi sufficiente avere a disposizione tutti i valori $C[i,r]$ e $C[r+1,j]$, per $i \leq r \leq j-1$. Un possibile ordine di visita della matrice di programmazione dinamica sarà quindi per diagonali, cominciando dalla diagonale principale della matrice, e continuando per le diagonali della parte triangolare superiore. Più in dettaglio, costruiremo la

```
    algoritmo ordineMatrici(ℓ1, ℓ2, ..., ℓn+1) → intero
1.    matrice C di n × n interi
2.    for i = 1 to n do C[i, i] ← 0
3.    for d = 1 to (n − 1) do
4.      for i = 1 to (n − d) do
5.        j ← d + i
6.        C[i, j] ← +∞
7.        for r = i to (j − 1) do
8.          C[i, j] ← min{C[i, j], C[i, r] + C[r + 1, j] + ℓi · ℓr+1 · ℓj}
9.    return C[1, n]
```

Figura 10.4 Algoritmo di calcolo dell'ordine ottimo di moltiplicazione tra matrici.

matrice di programmazione dinamica diagonale dopo diagonale: la diagonale d, $1 \leq d \leq n-1$, contiene tutti gli elementi $C[i,j]$ tali che $(j-i) = d$. Questo è esattamente ciò che viene eseguito dallo pseudocodice in Figura 10.4, che calcola la relazione (10.7) alle righe 6–8.

Teorema 10.2 *L'algoritmo* `ordineMatrici` *richiede un tempo di esecuzione* $O(n^3)$, *dove* n *è il numero di matrici che si vogliono moltiplicare.*

Dimostrazione. Consideriamo lo pseudocodice in Figura 10.4, e sia d la diagonale che si sta esaminando, $1 \leq d \leq (n-1)$. Ci sono esattamente $(n-d)$ elementi che devono essere calcolati nella diagonale d, e per ognuno di questi elementi dobbiamo scegliere tra d diverse possibilità. Il tempo di esecuzione dell'algoritmo è pertanto proporzionale a

$$\sum_{d=1}^{n-1}(n-d)d = n\sum_{d=1}^{n-1} d - \sum_{d=1}^{n-1} d^2 = \frac{n^2(n-1)}{2} - \frac{n(n-1)(2n-1)}{6} = \frac{(n^3-n)}{6}$$

è quindi $O(n^3)$. □

Concludiamo questo paragrafo osservando che la tecnica di programmazione dinamica viene utilizzata spesso per risolvere problemi di ottimizzazione che verificano il principio di *sottostruttura ottima*: in una sequenza ottima di decisioni, anche ogni sottosequenza deve essere ottima. Nel caso dell'associatività del prodotto tra matrici, ad esempio, se il modo migliore di moltiplicare la sequenza di

matrici

$$(M_i \cdot M_{i+1} \cdot \cdots \cdot M_j)$$

richiede di associare tale prodotto come

$$(M_i \cdot M_{i+1} \cdot \cdots \cdot M_r) \cdot (M_{r+1} \cdot \cdots \cdot M_j)$$

il principio di sotostruttura ottima equivale a dire che anche $(M_i \cdot M_{i+1} \cdot \cdots \cdot M_r)$ e $(M_{r+1} \cdot \cdots \cdot M_j)$ devono essere calcolate in modo ottimo. Vederemo ulteriori applicazioni del principio di sottostruttura ottima e della tecnica di programmazione dinamica nel Capitolo 13 quando parleremo di cammini minimi in un grafo.

10.3 Tecnica golosa (o *greedy*)

La tecnica golosa o *greedy* viene tipicamente utilizzata per risolvere problemi di ottimizzazione, ovvero problemi per cui desideriamo trovare la migliore soluzione possibile. Alcuni problemi di ottimizzazione sono ad esempio trovare il percorso più breve per recarsi dalla città A alla città B, trovare l'ordine migliore in cui eseguire alcuni *job* in un computer, trovare il modo migliore di trasformare una stringa X in una stringa Y, oppure trovare l'ordine migliore in cui eseguire il prodotto di n matrici. Nella situazione più generale, in un problema di ottimizzazione avremo:

- Un insieme di candidati possibili (ad esempio, città, oppure *job* da eseguire).
- L'insieme dei candidati che sono già stati utilizzati.
- Una funzione `ammissibile` che verifica se un insieme di candidati fornisce una soluzione (anche se non necessariamente ottima) al nostro problema. Ad esempio: un cammino non necessariamente ottimo dalla città A alla città B, oppure una trasformazione non nessariamente di costo minimo dalla stringa X alla stringa Y.
- Una funzione `ottimo` che verifica se un insieme di candidati fornisce una soluzione ottima al nostro problema.
- Una funzione di selezione `seleziona` che indica quale dei candidati non ancora esaminati sia il più promettente.
- Una funzione obiettivo che fornisce il valore di una soluzione (ad esempio, la lunghezza del percorso, il tempo necessario per eseguire tutti i *job* nell'ordine stabilito).

Per risolvere il problema di ottimizzazione, dovremo trovare un insieme di candidati che goda delle seguenti proprietà:

(1) è innanzitutto una soluzione del problema, e
(2) ottimizza (i.e., minimizza o massimizza) il valore della funzione obiettivo.

Descriviamo ora l'approccio generalmente seguito dalla tecnica golosa nel risolvere problemi di ottimizzazione, che è illustrato nello pseudocodice della Figura 10.5. All'inizio, l'insieme dei candidati selezionati S è vuoto. Al generico

passo, cerchiamo di aggiungere a questo insieme il miglior candidato x tra quelli non ancora considerati; osserviamo che x ci viene segnalato dalla funzione `seleziona`. Se questa aggiunta rende l'insieme $S \cup \{x\}$ non ammissibile, allora scartiamo completamente x e non lo considereremo più in futuro. Altrimenti, inseriamo permanentemente x nell'insieme dei candidati selezionati. Ogni volta che inseriamo un nuovo candidato in questo insieme, verificheremo se abbiamo raggiunto la soluzione ottima.

```
    algoritmo paradigmaGreedy(insieme di candidati C) → soluzione
1.    S ← ∅
2.    while ( (not ottimo(S)) and (C ≠ ∅) ) do
3.       x ← seleziona(C)
4.       C ← C − {x}
5.       if ( ammissibile(S ∪ {x}) ) then S ← S ∪ {x}
6.    if ( ottimo(S) ) then return S
7.    else errore non ho trovato soluzioni
```

Figura 10.5 Algoritmo goloso generico.

Osserviamo che alla riga 3 l'algoritmo goloso sceglie il candidato che allo stato attuale delle cose sembra il più promettente, senza preoccuparsi affatto del futuro. L'algoritmo non può avere ripensamenti: se un candidato è stato selezionato rimane per sempre nella soluzione, e se invece è stato scartato, non verrà mai più preso in considerazione. Questo giustifica il fatto che questa tecnica sia appunto chiamata golosa, o *greedy* in inglese. Nei prossimi paragrafi, vederemo alcune applicazioni della tecnica golosa.

10.3.1 Il distributore automatico di resto

I distributori automatici, nel dare il resto ai clienti, tipicamente utilizzano il minor numero possibile di monete. Ad esempio, se una bibita in un distributore automatico costa 37 centesimi di Euro, e noi introduciamo 1 Euro per il pagamento, il resto sarà molto probabilmente composto da una moneta da 50 centesimi, una da 10 centesimi, una da 2 centesimi ed una da 1 centesimo, per un totale di 63 centesimi di Euro. La ragione di questo comportamento potrebbe ad esempio essere un consumo uniforme delle tipologie di monete contenute nel serbatoio del distributore, onde evitare che esso rimanga privo di monete di un certo tipo. Supponiamo quindi che il distributore automatico abbia a disposizione, nel suo serbatoio, un certo numero di monete, ad esempio da 1, 2, 5, 10, 20 e 50 centesimi di Euro, e che debba erogare un resto R ad un cliente. Come fa il distributore a calcolare quali monete dare come resto? Possiamo formulare questo problema nel paradigma goloso nel modo seguente:

- L'insieme di candidati possibili è dato da un insieme finito di monete, ad esempio da 1, 2, 5, 10, 20 e 50 centesimi di Euro, contenute nel serbatoio del distributore.
- La funzione `ammissibile` restituisce vero se il valore delle monete nell'insieme scelto non è superiore al resto che il distributore automatico deve restituire al cliente.
- La funzione `ottimo` restituisce vero se il valore delle monete nell'insieme scelto è esattamente uguale al resto che il distributore automatico deve restituire al cliente.
- La funzione `seleziona` sceglie la moneta di valore più grande rimasta tra quelle ancora da considerare.
- La funzione obiettivo restituisce il numero di monete presenti nella soluzione.
- La funzione `valore` restituisce il valore totale delle monete nell'insieme considerato.


```
    algoritmo distribuisciResto(resto R) → soluzione
1.    C ← monete contenute nel serbatoio del distributore
2.    S ← ∅
3.    while ( ( valore(S) ≠ R) and (C ≠ ∅) ) do
4.       x ← moneta di valore più elevato in C
5.       C ← C − {x}
6.       if ( valore(S ∪ {x}) ≤ R ) then S ← S ∪ {x}
7.    if ( valore(S) = R ) then return S come resto esatto
8.    else return S come resto parziale
```

Figura 10.6 Algoritmo goloso per la distribuzione automatica del resto.

Ponendo quindi nell'algoritmo goloso generico di Figura 10.5:

`ammissibile`(S)	←	(`valore`$(S) \leq R$)
`ottimo`(S)	←	(`valore`$(S) = R$)
`seleziona`(C)	←	scegli la moneta di valore più grande in C

otteniamo lo pseudocodice per realizzare il distributore automatico di resto illustrato in Figura 10.6.

Esempio 10.3 Consideriamo un distributore di resto contenente nel suo serbatoio tre monete, due da 50 centesimi, ed una da 20 centesimi ($C = \{50, 50, 20\}$), e assumiamo che debba distribuire un resto di 55 centesimi ($R = 55$). La Tabella 10.4 illustra l'esecuzione dell'algoritmo `distribuisciResto` di Figura 10.6 su tali dati. In particolare, la tabella mostra lo stato delle principali variabili immediatamente dopo l'esecuzione dell'i-esima iterazione del ciclo **while** alle righe 3–6. Alla fine dell'esecuzione, l'algoritmo restituirà 50 centesimi di resto. □

iterazione	C	x	S	R
0	$\{50, 50, 20\}$		$\emptyset$	55
1	$\{50, 20\}$	50	50	55
2	$\{20\}$	50	50	55
3	$\emptyset$	20	50	55

Tabella 10.4 Risultati delle iterazioni dell'algoritmo `distribuisciResto` sui dati dell'Esempio 10.3.

10.3.2 Problemi di sequenziamento

Ci siamo mai chiesti perché i messaggi di posta elettronica più brevi impiegano meno tempo ad essere recapitati, mentre invece i messaggi di posta elettronica più voluminosi (magari con allegati di grandi dimensioni) impiegano più tempo?

Assumiamo di avere un server (ad esempio un server di posta elettronica, un server Web, una CPU, una stampante, o un impiegato dell'ufficio postale) che deve servire n clienti (messaggi di posta elettronica, richieste HTTP, *job* che richiedono l'attenzione della CPU, documenti in coda di stampa, o clienti che desiderano pagare dei bollettini postali). Supponiamo che il tempo di servizio di ogni cliente sia noto a priori: per $1 \leq i \leq n$, il servizio richiesto dal cliente i richiede esattamente t_i secondi.

Per $1 \leq i \leq n$, definiamo $T(i)$ come il tempo di attesa del cliente i, ovvero il tempo totale in cui il cliente i rimane nel sistema. In un tale contesto, tipicamente si desidera calcolare un ordine in cui servire le richieste dei clienti tale da minimizzare il tempo totale in cui i clienti rimangono nel sistema:

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

Visto che il numero n di clienti è fissato, questo equivale a minimizzare il tempo di servizio medio del generico cliente:

$$T_{avg} = \frac{T}{n} = \frac{1}{n} \sum_{i=1}^{n} T(i)$$

Esempio 10.4 Supponiamo che nella coda di un server di posta elettronica sono presenti tre messaggi che attendono di essere recapitati ai loro destinatari. I tempi richiesti per servire i tre messaggi sono rispettivamente:

$$t_1 = 50 \text{ msec}, \quad t_2 = 100 \text{ msec}, \quad t_3 = 3 \text{ msec}$$

In totale sono possibili sei diversi ordini di recapito dei messaggi. Quali di questi è preferibile? Innanzitutto osserviamo che per il server di posta elettronica non

Ordine	T			
1 2 3	50 + (50 + 100) + (50 + 100 + 3)	msec	=	353 msec
1 3 2	50 + (50 + 3) + (50 + 3 + 100)	msec	=	256 msec
2 1 3	100 + (100 + 50) + (100 + 50 + 3)	msec	=	403 msec
2 3 1	100 + (100 + 3) + (100 + 3 + 50)	msec	=	356 msec
3 1 2	3 + (3 + 50) + (3 + 50 + 100)	msec	=	209 msec
3 2 1	3 + (3 + 100) + (3 + 100 + 50)	msec	=	259 msec

Tabella 10.5 Tempi totali di servizio in funzione dell'ordine di consegna dei messaggi.

cambia nulla, perché durante la consegna dei tre messaggi sarà sempre occupato per un totale di 153 msec, indipendentemente dall'ordine in cui i messaggi verranno recapitati. Però l'ordine di consegna può avere un effetto non trascurabile sui tempi di attesa dei messaggi. Infatti, come illustrato nella Tabella 10.5, se scegliamo di recapitare prima il messaggio 2, poi 1 e poi 3, il tempo di attesa del messaggio 2 sarà 100 msec, del messaggio 1 sarà 150 msec, e del messaggio 3 sarà 153 msec. In questo caso il tempo totale di attesa sarà quindi pari a 403 msec, ed il tempo medio di attesa per ogni messaggio sarà pari a $\frac{403}{3} = 134.33$ msec. Se invece scegliamo di recapitare prima il messaggio 3, poi 1 e poi 2, il tempo di attesa del messaggio 3 sarà 3 msec, del messaggio 1 sarà 53 msec, e del messaggio 3 sarà 153 msec. In quest'ultimo caso quindi il tempo totale di attesa sarà pari a 209 msec, per un tempo medio di attesa per ogni messaggio pari a $\frac{209}{3} = 69.67$ msec.

In questo esempio, anche se per il server di posta elettronica non cambia nulla, l'ordine di servizio ha un impatto cruciale sui tempi medi di servizio. In particolare, se diamo priorità più elevata ai messaggi più brevi siamo in grado di ridurre all'incirca del 50% il tempo medio di servizio! □

Consideriamo ora come risolvere il problema di calcolare il sequenziamento ottimo nel caso generale. Supponiamo che il nostro algoritmo generi l'ordine di servizio in maniera incrementale, ovvero ha già deciso di sequenziare i clienti i_1, $i_2, \ldots, i_m$. Se adesso decide di servire il cliente j, allora il tempo totale di servizio diventerà

$$t_{i_1} + t_{i_2} + \ldots + t_{i_m} + t_j$$

Osserviamo che se vogliamo minimizzare l'incremento nel tempo totale di servizio, sarà sufficiente minimizzare t_j. Questo ci suggerisce un semplice algoritmo goloso: tra i clienti rimasti, è sempre conveniente selezionare quello che richiede il servizio più breve. Nell'esempio 10.4, l'algoritmo goloso trova correttamente la soluzione 3, 1, 2. Il tempo di esecuzione di questo algoritmo sarà $O(n \log n)$, ovvero il tempo richiesto semplicemente per ordinare i tempi di servizio degli n clienti.

Concludiamo questo paragrafo osservando che non sempre gli algoritmi golosi sono in grado di trovare soluzioni ottime, come illutrato ad esempio nel Problema 10.8. Progetteremo ancora altri algoritmi con il metodo goloso quando studieremo problemi su grafi, ed in particolare nei Capitoli 12 e 13.

10.4 Problemi

Problema 10.1 Sia data una sequenza S di n oggetti $x_1, x_2, \ldots, x_n$, non necessariamente distinti. Diciamo che un oggetto x è un *elemento di maggioranza* in S se appare almeno $\lceil n/2 \rceil$ volte in S. Progettare un algoritmo efficiente che sia in grado di stabilire se S contiene un elemento di maggioranza; ed in caso affermativo lo restituisca. Il tempo di esecuzione dell'algoritmo dovrebbe essere lineare in n.

Problema 10.2 Sia dato un array A di n interi. Progettare un algoritmo *divide et impera* che sia in grado di restituire il massimo ed il minimo in A in tempo $O(n)$.

Problema 10.3 Siano date due stringhe A e B, rispettivamente di m e n caratteri. Diciamo che A è una *sottosequenza* di B se tutti i caratteri di A appaiono nello stesso ordine anche in B, anche se possono essere separati da altri caratteri. Ad esempio la stringa `serafico` è una sottosequenza della stringa `supercalifragilistichespiralidoso` — l'occorrenza di `serafico` è evidenziata in grassetto in **SupERcAliFragIlistiChespiralidOso**. Progettare un algoritmo efficiente per calcolare, date due stringhe A e B, se A è una sottosequenza di B. Analizzare il tempo di esecuzione.

Problema 10.4 Siano date due stringhe X e Y. Una sequenza Z è una *sottosequenza comune* di X e Y, se è contemporaneamente sottosequenza di X e di Y (vedi il Problema 10.3). Progettare un algoritmo efficiente che riceve in ingresso due stringhe X e Y, rispettivamente di m e n caratteri, e restituisce la sottosequenza comune di X e Y di lunghezza massima. Ad esempio, `maria` è la sottosequenza di lunghezza massima comune a `marinaio` e a `malaria`. Analizzare il tempo di esecuzione del proprio algoritmo.

Problema 10.5 Il Dott. Copiatutto ha nel suo PC n file $F_1, F_2, \ldots, F_n$, tali che il file F_i occupa $w(F_i)$ Megabyte, e tali che

$$\sum_{i=1}^{n} w(F_i) = 1300 \text{ Megabyte}$$

Il Dott. Copiatutto vuole trasferire gli n file $F_1, F_2, \ldots, F_n$ su due CD-ROM, ciascuno avente 650 Megabyte di capacità, in modo tale che nessun file venga spezzato tra i due CD-ROM. Progettare un algoritmo efficiente per aiutare il Dott. Copiatutto a verificare se questo è possibile, ed eventualmente a individuare come dividere i file tra i due CD-ROM.

Problema 10.6 Consideriamo l'algoritmo goloso per la distribuzione automatica del resto descritto in Figura 10.6. Supponiamo che il distributore automatico contenga almeno una moneta di ogni tipologia. Quale delle seguenti due affermazioni è vera?

(1) Se esiste una soluzione, l'algoritmo goloso la trova sempre.
(2) Esiste almeno un caso in cui una soluzione esiste, ma l'algoritmo goloso non è in grado di trovarla.

Motivare in dettaglio le proprie risposte.

Problema 10.7 Consideriamo l'algoritmo goloso per il sequenziamento dei lavori descritto nel Paragrafo 10.3. Quale delle seguenti due affermazioni è vera?

(1) L'algoritmo goloso trova sempre la soluzione ottima.
(2) Esiste almeno un caso in cui l'algoritmo goloso non è in grado di trovare la soluzione ottima.

Motivare in dettaglio le proprie risposte.

Problema 10.8 Consideriamo il seguente problema. La stanza di un museo contiene una collezione di n opere d'arte $X = \{x_1, x_2, \ldots, x_n\}$ tali che, per $1 \leq i \leq n$, l'opera d'arte x_i pesa $p(x_i)$ kilogrammi e ha un valore di $v(x_i)$ Euro. Un ladro che si trova nella stanza del museo è in grado di trasportare al massimo un peso totale pari a P, ed è intenzionato a selezionare un sottoinsieme $Y \subseteq X$ delle opere d'arte oggetti tale che

$$\sum_{x_i \in Y} p(x_i) \leq P$$

così da massimizzare il loro valore:

$$\max_{Y \subseteq X} \left\{ \sum_{x_i \in Y} v(x_i) \right\}$$

(1) Progettare un algoritmo goloso per questo problema.
(2) Esiste almeno un caso in cui l'algoritmo goloso progettato in (1) non è in grado di trovare la soluzione ottima? Motivare in dettaglio la propria risposta.

10.5 Sommario

In questo capitolo abbiamo presentato alcune importanti tecniche algoritmiche, e in particolare la tecnica *divide et impera*, la programmazione dinamica, e il metodo goloso o *greedy*.

La tecnica *divide et impera* consiste nel dividere l'istanza del problema in ingresso in due o più sottoistanze (*divide*), risolvere ricorsivamente il problema sulle sottoistanze e poi ricombinare la soluzione dei sottoproblemi (*impera*) allo scopo di ottenere la soluzione globale del problema originario. Come dovrebbe essere evidente dagli esempi che abbiamo visto finora, è tipicamente una tecnica *top-down*: quando risolviamo un problema con la tecnica *divide et impera*, infatti, procediamo dall'alto verso il basso, affrontando immediatamente l'istanza del problema generale, e dividendola via via in istanze più piccole man mano che l'algoritmo procede nella sua esecuzione. Abbiamo esaminato varie modalità di decomposizione in sottoproblemi, e di ricombinazione delle soluzioni. Le due fasi sono intimamente legate tra di loro, e dipendono anche dalla struttura del problema che si vuole risolvere: dalla loro efficacia dipende in gran parte l'efficienza dell'algoritmo ottenuto.

La programmazione dinamica è invece una tecnica che si basa su una filosofia di tipo *bottom-up*. Quando utilizziamo tale tecnica, infatti, consideriamo implicitamente dei sottoproblemi, definiti su opportune sottoistanze del problema, e procediamo logicamente dai sottoproblemi più semplici verso i sottoproblemi più complessi. Tale tecnica si applica tipicamente a problemi in cui si presentanto sottoproblemi che non sono del tutto indipendenti, per cui uno stesso sottoproblema può apparire più volte durante la risoluzione del problema originario. La programmazione dinamica utilizza una tabella per memorizzare le soluzioni dei sottoproblemi incontrati: quando si incontrerà di nuovo lo stesso sottoproblema, sarà sufficiente esaminare l'elemento corrispondente della tabella di programmazione dinamica, piuttosto che risolverlo di nuovo. Ulteriori applicazioni della tecnica di programmazione dinamica saranno illustrate nel Capitolo 13 (cammini minimi).

La tecnica golosa invece si applica a problemi di ottimizzazione in cui ad ogni passo si devono compiere un insieme di scelte: il metodo goloso suggerisce di effettuare le scelte che appaiono più promettenti per il presente, senza preoccuparsi affatto del futuro. Abbiamo visto che la tecnica golosa non è sempre in grado di produrre algoritmi efficienti per i problemi a cui si applica. Ulteriori applicazioni della tecnica golosa saranno illustrate nel Capitolo 12 (minimi alberi ricoprenti) e nel Capitolo 13 (cammini minimi).

10.6 Note bibliografiche

L'algoritmo $O(n^{1.59})$ per moltiplicare due interi con n cifre decimali non è il più efficiente algoritmo noto per questo problema. Schönage e Strassen [8] hanno proposto un algoritmo che richiede tempo $O(n \log n \log \log n)$. Algoritmi efficienti per moltiplicare interi sono stati utilizzati per calcolare π fino a oltre 10 milioni di cifre decimali [7]. L'algoritmo che calcola il prodotto di due matrici $n \times n$ in tempo $O(n^{2.81})$ è stato proposto da Strassen [9]. L'algoritmo attualmente più veloce per il calcolo del prodotto di due matrici $n \times n$ è stato proposto da Coppersmith e Winograd [3] e ha tempo di esecuzione $O(n^{\omega})$, dove $\omega < 2.376$.

La tecnica di programmazione dinamica è illustrata in vari libri, come ad esempio [1, 2]. L'algoritmo che calcola la distanza tra stringhe è di Wagner e Fischer [10], mentre l'algoritmo relativo all'associatività del prodotto tra matrici è descritto da Godbole in [4]. Un algoritmo più efficiente per quest'ultimo problema è stata proposto da Hu e Shing [5, 6].

Riferimenti bibliografici

[1] R.E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[2] R.E. Bellman and S.E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ, 1962.

[3] D. Coppersmith, S. Winograd, "Matrix multiplication via arithmetic progression", *Journal of Symbolic Computation*, 9, 1990, 251–280.

[4] S. Godbole, "On efficient computation of matrix chain products", *IEEE Trans. on Computers*, C-22 (9), 1973, 864–866.

[5] T.C. Hu and M.R. Shing, "Computation of matrix chain products, Part I", *SIAM Journal on Computing*, 11 (2), 1982, 362–373.

[6] T.C. Hu and M.R. Shing, "Computation of matrix chain products, Part II", *SIAM Journal on Computing*, 13 (2), 1984, 228–251.

[7] Y. Kanada, Y. Tamura, S. Yoshino and Y. Ushiro, "Calculation of π to 10,013,395 decimal places based on the Gauss-Legendre Algorithm and Gauss Arctangent relation", Tech. Report 84-01, Computer Centre, University of Tokyo, 1983.

[8] A. Schönage and V. Strassen, "Schnelle Multiplikation grosser Zahlen", *Computing*, 7, 1971, 281–292.

[9] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik* 13, 1969, 354–356.

[10] R.A. Wagner and M.J. Fischer, "The string-to-string correction problem", *Journal of the Association of Computing Machinery*, 21 (1), 1974, 168–173.

11

Grafi e visite di grafi

Andare in profondità significa pure riuscire a scoprire la legge che unifica fatti apparentemente sconnessi e incomponibili, come appare manifesto quando l'analisi ha quella veste astratta e universale che le danno le formule algebriche. Poiché quella veste commune rende comparabili fra loro anche concetti che a prima vista potevano apparir privi d'ogni intima relazione. E così nella confusione del superficiale e del vario, la mente può discernere l'identico, il costante, l'essenziale, il certo.

(Carlo Cattaneo – Psicologia delle menti associate.)

Un grafo è una nozione matematica astratta, usata per rappresentare l'idea di "connessione" o di "relazione" tra coppie di oggetti. In particolare,

Definizione 11.1 *Un grafo $G = (V, E)$ consiste in:*

- *un insieme V di "vertici" (o "nodi"), e*
- *un insieme E di coppie di vertici, detti "archi" (o "spigoli"): ogni arco connette due vertici.*

I vertici rappresentano oggetti e gli archi rappresentano relazioni tra questi oggetti. Notiamo che in questa definizione astratta non è importante cosa sia rappresentato dai vertici, dagli archi o complessivamente dal grafo: questo aspetto dipende in generale dal problema o dall'applicazione che si vuole modellare.

Esistono due tipi di grafi: grafi orientati e grafi non orientati. In un *grafo non orientato* (oppure *non diretto*) gli archi non hanno un'orientazione: ciò accade quando la relazione tra una coppia di vertici esemplificata da un arco è simmetrica. Un esempio di grafo non orientato è illustrato nella Figura 11.1 (b).

Esempio 11.1 Supponiamo di disegnare un grafo con un vertice per ogni persona che vive in Italia, ed un arco che connette due persone che si sono strette la mano almeno una volta nella loro vita. Se ti ho stretto la mano, anche tu evidentemente mi hai stretto la mano, e quindi ogni arco che esemplifica la relazione delle "strette di mano" è inerentemente simmetrico. Sorprendentemente, i grafi delle strette di mano hanno un "diametro" molto piccolo: possiamo connettere ogni coppia di persone con una breve catena di strette di mano! □

L'altra tipologia di grafi è nota come *grafi orientati* (oppure *diretti*). Tali grafi rappresentano relazioni *orientate* tra coppie di oggetti: ogni arco è orientato da uno dei due vertici verso l'altro vertice, e si disegna con un freccia che ne indica la direzione. Un esempio di grafo orientato è illustrato nella Figura 11.1 (c).

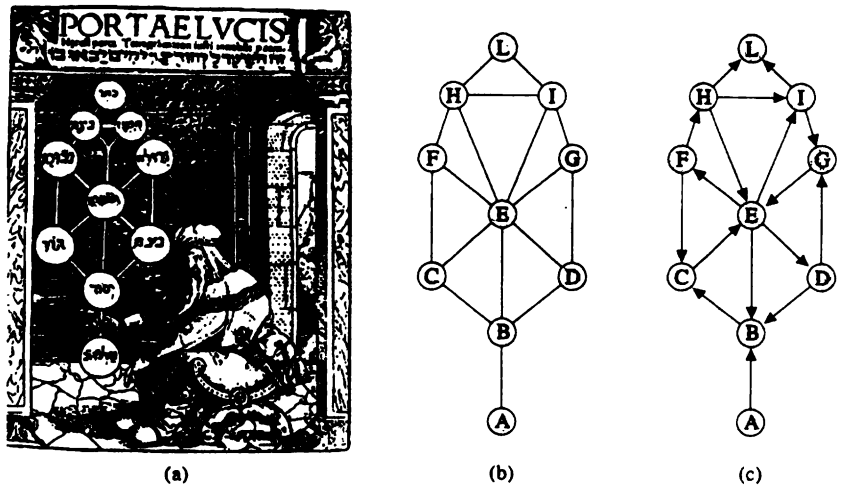


Figura 11.1 (a) Illustrazione del 1516 di Paolo Riccl (anche noto come Paulus Israelita) raffigurante il grafo delle relazioni tra le dieci forze creative (i Sefirot) della dottrina mistica ebraica; (b) grafo non orientato derivato dal grafo dei Sefirot; (c) grafo orientato derivato dal grafo dei Sefirot: si noti che in questo caso le relazioni tra vertici non sono simmetriche.

Esempio 11.2 Come ulteriore esempio di grafo orientato, potremmo disegnare un grafo che ha ancora come vertici tutte le persone che vivono in Italia, ma questa volta con un arco (orientato) tra la persona A e la persona B se A ha spedito un messaggio di posta elettronica a B. Notiamo che in tal caso la relazione non è necessariamente simmetrica: io potrei averti inviato una posta elettronica, ma tu potresti non avermi mai spedito alcuna posta elettronica. Un altro esempio di grafo orientato è il grafo del Web: esiste un vertice per ogni pagina Web (statica), ed esiste un arco dal vertice x al vertice y se la pagina Web x contiene un *link* alla pagina Web y. Anche questa è una relazione non simmetrica: nella mia pagina Web personale potrei inserire un link alla pagina Web della mia squadra di calcio preferita, ma non è assolutamente detto che la pagina Web della mia squadra di calcio preferita contenga un link alla mia pagina Web personale! □

Denoteremo un grafo come $G = (V, E)$, intendendo che V è l'insieme dei vertici, ed $E \subseteq V \times V$ è l'insieme degli archi. Utilizzeremo inoltre n per indicare il numero di vertici, ed m per indicare il numero di archi. A volte indicheremo

anche con $V(G)$ l'insieme dei vertici del grafo G, e con $E(G)$ l'insieme degli archi di G. Prima di addentrarci ulteriormente nel mondo dei grafi, introdurremo alcune nozioni, definizioni ed ulteriore terminologia di base.

11.1 Definizioni preliminari sui grafi

Molte delle definizioni di base sui grafi sono comuni a grafi orientati e non orientati. A volte però, nel caso dei grafi orientati, si hanno piccole differenze di notazione. Nelle seguenti definizioni, faremo quindi riferimento sia a grafi orientati che a grafi non orientati, a meno che non sia specificato esplicitamente.

Adiacenza ed incidenza. Se (x, y) è l'arco di un grafo G, diciamo che (x, y) è *incidente* sui vertici x e y. Se inoltre il grafo G è orientato, diciamo che l'arco (x, y) *esce* dal vertice x ed *entra* nel vertice y. Nel caso di un grafo non orientato, diremo che x è *adiacente* a y, y è *adiacente* a x, ovvero x ed y sono *adiacenti*. Nel caso di un grafo orientato, diremo che y è *adiacente* a x. Dato un vertice v, chiameremo i vertici adiacenti a v anche *vicini* di v.

Grado di un vertice. Il *grado* di un vertice v in un grafo è dato dal numero di archi incidenti su v. Denoteremo il grado del vertice v con la notazione $\delta(v)$. Ad esempio, nel grafo di Figura 11.1 (b), il vertice H ha grado 4, e quindi $\delta(H) = 4$. Osserviamo che, se sommiamo tutti i gradi dei vertici di un grafo non orientato, conteremo ogni arco (x, y) esattamente due volte: una volta quando consideriamo il grado di x ed una volta quando consideriamo il grado di y. Quindi:

$$\sum_{v \in G} \delta(v) = 2m$$

Se il grafo è orientato, parleremo di *grado in uscita di* v, intendendo con questo il numero di archi che escono da v, e di *grado in entrata di* v, intendendo con questo il numero di archi che entrano in v. Denoteremo il grado in entrata ed il grado in uscita di un vertice v rispettivamente con la notazione $\delta_{in}(v)$ e $\delta_{out}(v)$. Nel caso di un grafo orientato, il grado di un vertice sarà dato dal suo grado in entrata sommato al grado in uscita: $\delta(v) = \delta_{in}(v) + \delta_{out}(v)$. Ad esempio, il vertice E nel grafo orientato di Figura 11.1 (c) ha grado in entrata $\delta_{in}(E) = 3$, grado in uscita $\delta_{out}(E) = 4$, e quindi grado totale $\delta(E) = 7$. Osserviamo che nel caso di un grafo orientato:

$$\sum_{v \in G} \delta_{in}(v) = \sum_{v \in G} \delta_{out}(v) = m$$

e quindi anche nel caso di grafi orientati otteniamo la relazione

$$\sum_{v \in G} \delta(v) = \sum_{v \in G} \delta_{in}(v) + \sum_{v \in G} \delta_{out}(v) = 2m$$

Cammini e cicli. Un *cammino* in un grafo G da un vertice x ad un vertice y è dato da una sequenza di vertici $\langle v_0, v_1, \ldots, v_k \rangle$ con $v_0 = x$ e $v_k = y$, tale che, per $1 \leq i \leq k$, l'arco (v_{i-1}, v_i) appartiene a G. In particolare, in tal caso diremo che il cammino è di *lunghezza* k. Diremo che il cammino *contiene* (ovvero *passa per*) i vertici $v_0, v_1, \ldots, v_k$ e gli archi $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Diremo che il cammino è *semplice* se tutti i suoi vertici sono distinti. Nel caso di grafi orientati, indicheremo un cammino dal vertice x al vertice y con la notazione $x \rightsquigarrow y$.

Esempio 11.3 Nel grafo non orientato di Figura 11.1 (b), esiste un cammino semplice di lunghezza 4 tra il vertice A ed il vertice L, contenente i vertici A, B, E, I, L e gli archi (A, B), (B, E), (E, I) ed (I, L). □

Esempio 11.4 Nel grafo orientato di Figura 11.1 (c), esiste un cammino di lunghezza 8 tra il vertice A ed il vertice L, contenente i vertici A, B, C, E, F, H, E, I, L e gli archi (A, B), (B, C), (C, E), (E, F), (F, H), (H, E), (E, I), ed (I, L). Tale cammino $A \rightsquigarrow L$ non è un cammino semplice perchè passa due volte per lo stesso vertice E. □

Se esiste un cammino da un vertice x ad un vertice y di G, diremo anche che y è *raggiungibile da* x, ovvero che y è un *discendente* di x e che x è un *antenato* di y. Un cammino $\langle v_0, v_1, \ldots, v_k \rangle$ tale che $v_0 = v_k$ e $k \geq 1$ è un *ciclo*. Il ciclo è *semplice* se i vertici $v_0, v_1, \ldots, v_{k-1}$ sono tutti distinti. Un *grafo diretto aciclico* è un grafo orientato non contenente cicli.

Esempio 11.5 Nel grafo non orientato di Figura 11.1 (b), esiste un ciclo semplice contenente i vertici B, C, E, D, B e gli archi (B, C), (C, E), (E, D), (D, B). Invece il ciclo nel grafo orientato di Figura 11.1 (c), contenente i vertici E, F, H, E, I, G, E e gli archi (E, F), (F, H), (H, E), (E, I), (I, G), e (G, E) non è un ciclo semplice perché passa più volte per lo stesso vertice E. □

Connettività e connettività forte. Un grafo non orientato $G = (V, E)$ si dice *connesso* se esiste un cammino tra ogni coppia di vertici in G. Ad esempio, il grafo di Figura 11.1 (b) è connesso. È semplice dimostrare che ogni grafo per essere connesso deve avere almeno $(n - 1)$ archi (si veda il Problema 11.1). Un grafo orientato $G = (V, E)$ si dice *fortemente connesso* se esiste un cammino (orientato) tra ogni coppia di vertici in G. Il grafo di Figura 11.1 (c) non è fortemente connesso, perchè ad esempio non esiste un cammino dal vertice B al vertice A. Diremo inoltre che un vertice x è fortemente connesso ad un vertice y se esiste un cammino (orientato) da x ad y ed un cammino (orientato) da y ad x. Notiamo che i due cammini possono anche avere archi o vertici in comune, o possono essere anche cammini degeneri, come quelli contenenti un solo vertice: come caso particolare, un vertice è fortemente connesso a se stesso. Utilizzeremo la notazione $x \Leftrightarrow y$ per indicare che il vertice x è fortemente connesso a y.

Sottografi. Diciamo che un grafo $G' = (V', E')$ è un *sottografo* di un grafo $G = (V, E)$ se $V' \subseteq V$ ed $E' \subseteq E$. In questo caso utilizzeremo la notazione

tipo `Grafo`:
dati:
un insieme di vertici (di tipo *vertice*) e un insieme di archi (di tipo *arco*).

operazioni:

`numVertici()` → *intero*
restituisce il numero di vertici presenti nel grafo.

`numArchi()` → *intero*
restituisce il numero di archi presenti nel grafo.

`grado(`*vertice v*`)` → *intero*
restituisce il numero di archi incidenti sul vertice v.

`archiIncidenti(`*vertice v*`)` → $\langle$*arco, arco, ..., arco*$\rangle$
restituisce, uno dopo l'altro, gli archi incidenti sul vertice v.

`estremi(`*arco e*`)` → $\langle$*vertice, vertice*$\rangle$
restituisce gli estremi x e y dell'arco $e = (x, y)$.

`opposto(`*vertice x, arco e*`)` → *vertice*
restituisce y, l'estremo dell'arco $e = (x, y)$ diverso da x.

`sonoAdiacenti(`*vertice x, vertice y*`)` → *booleano*
restituisce `true` se esiste l'arco (x, y), e `false` altrimenti

`aggiungiVertice(`*vertice v*`)`
inserisce un nuovo vertice v.

`aggiungiArco(`*vertice x, vertice y*`)`
inserisce un nuovo arco tra i vertici x e y.

`rimuoviVertice(`*vertice v*`)`
cancella il vertice v e tutti gli archi ad esso incidenti.

`rimuoviArco(`*arco e*`)`
cancella l'arco e.

Figura 11.2 Possibili dati e operazioni su grafi non orientati.

$G' \subseteq G$. Dati un grafo $G = (V, E)$ ed un insieme di vertici $V' \subseteq V$, il sottografo di G indotto da V' è definito come il grafo $G' = (V', E')$ dove $E' = \{(x, y) \in E : x, y \in V'\}$.

Come possiamo notare da queste definizioni preliminari, i grafi hanno una struttura molto ricca e complessa. Non sorprende quindi che sia possibile definire un tipo di dato grafo in vari modi, soprattutto in funzione delle operazioni che intendiamo supportare su di esso. A titolo puramente esemplificativo, descriviamo schematicamente in Figura 11.2 un possibile tipo di dato per grafi non orientati.

Concludiamo questo paragrafo con ulteriori esempi di grafi.

Esempio 11.6 Gli alberi, visti più volte in questo libro, ed in particolare nel Ca-

pitolo 3, sono in realtà casi particolari di grafi. Infatti, gli alberi non orientati possono essere definiti come grafi non orientati, connessi ed aciclici. Anche in questo caso non è difficile dimostrare cbe ogni albero non orientato ha esattamente $(n-1)$ archi (si veda il Problema 11.3). Nel Capitolo 6, abbiamo visto gli alberi di ricerca che rappresentano esempi di alberi non orientati. Nel Capitolo 4, abbiamo invece usato una notazione per gli alberi di decisione, in cui connettevamo due "nodi" tramite un arco diretto verso il basso per indicare l'esito di un confronto. Questo è un esempio di albero (e quindi di grafo) orientato. □

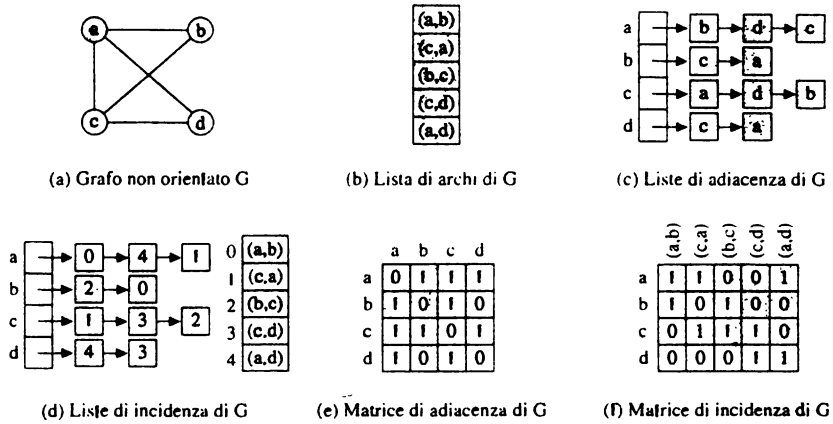
Esempio 11.7 Consideriamo ora il grafo delle rotte aeree, i cui vertici corrispondono ad aeroporti, e gli archi a voli. A differenza degli esempi precedenti, possiamo aggiungere altre informazioni al grafo: ad esempio, il tempo di percorrenza, il costo o il codice della linea aerea per ogni tratta. Un tipico algoritmo su grafi in questo esempio potrebbe trovare il cammino di percorso minimo o di costo minimo da una data città A ad un'altra città B, ad esempio consentendo o meno un certo numero di scali intermedi tra A e B. Notiamo che un tale grafo potrebbe essere orientato o meno, a seconda del tipo di informazioni che vogliamo associare agli archi: se associamo ad ogni arco il suo tempo di percorrenza, ad esempio, l'arco da x ad y sarà necessariamente distinto dall'arco simmetrico da y ad x. □

Da questi esempi introduttivi appare evidente che molti problemi possono essere formulati come problemi su grafi. Questa modellazione ha il pregio di nascondere i dettagli non necessari e di rendere evidente la struttura del problema. Se però vogliamo risolvere problemi, progettare algoritmi e scrivere programmi su grafi, dobbiamo per prima cosa decidere come rappresentare e memorizzare un grafo. Questo sarà l'argomento del prossimo paragrafo.

11.2 Strutture dati per rappresentare grafi

Osserviamo che, per essere efficacemente interpretata da un programma, una rappresentazione di un grafo deve essere vicina alla sua definizione astratta. In questo paragrafo esamineremo alcuni dei principali metodi di rappresentazione di un grafo.

Rappresentazione di grafi con lista di archi. La cosa più ovvia da fare sembrerebbe "ispirarsi" alla definizione astratta di grafo, ottenendone in qualche modo una sua rappresentazione elementare nel linguaggio di programmazione utilizzato. In altri termini, possiamo definire una qualche struttura per ogni vertice, che rappresenta l'informazione che vogliamo associare ad un vertice, ed un'altra struttura per ogni arco, contenente puntatori ai due vertici estremi dell'arco. In pratica una tale rappresentazione avrà una lista o un array di strutture (come ad esempio le `struct` in C o C++), dove utilizzermo una struttura per ogni vertice ed una per ogni arco. Faremo riferimento a questo modo di rappresentare un grafo



(a) Grafo non orientato G (b) Lista di archi di G (c) Liste di adiacenza di G

(d) Liste di incidenza di G (e) Matrice di adiacenza di G (f) Matrice di incidenza di G

Figura 11.3 Strutture dati per rappresentare grafi non orientati.

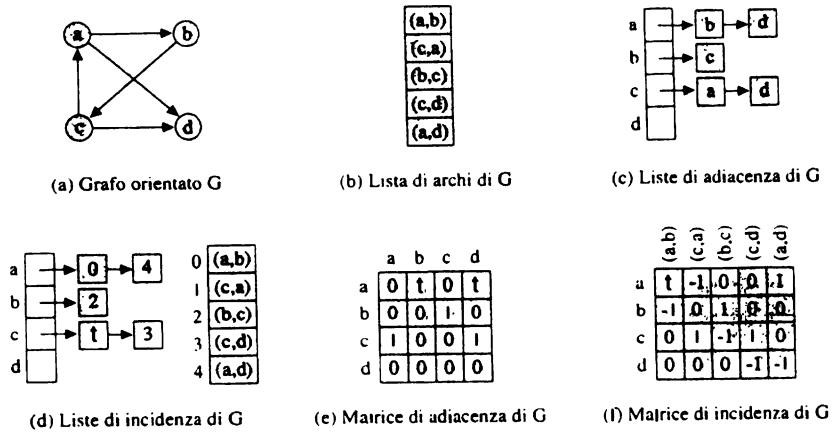
come rappresentazione con *lista di archi*. Lo spazio totale usato da questa rappresentazione è $O(m+n)$, visto che serve una quantità costante di spazio (una struttura) per ogni vertice e per ogni arco.

La Figura 11.3 (b) mostra come tale rappresentazione memorizza gli archi del grafo non orientato in Figura 11.3 (a). In modo del tutto simile, la Figura 11.4 (b) illustra come tale rappresentazione memorizza gli archi del grafo orientato in Figura 11.4 (a).

Anche se questa rappresentazione è molto diffusa, ci sono alcune operazioni che non sembrano facilmente realizzabili in modo efficiente. Come può vedersi dalla Tabella 11.1, che illustra i tempi di esecuzione della rappresentazione di un grafo non orientato rappresentato con lista di archi, il punto di debolezza sembra infatti che molte delle operazioni e delle tipologie di accessi agli archi del gra-

Operazione	Tempo di esecuzione
`grado`(v)	$O(m)$
`archiIncidenti`(v)	$O(m)$
`sonoAdiacenti`(x,y)	$O(m)$
`aggiungiVertice`(v)	$O(1)$
`aggiungiArco`(x,y)	$O(1)$
`rimuoviVertice`(v)	$O(m)$
`rimuoviArco`(e)	$O(m)$

Tabella 11.1 Tempi di esecuzione delle operazioni su grafi non orientati rappresentati con lista di archi.



(a) Grafo orientato G (b) Lista di archi di G (c) Liste di adiacenza di G

(d) Liste di incidenza di G (e) Matrice di adiacenza di G (f) Matrice di incidenza di G

Figura 11.4 Strutture dati per rappresentare grafi orientati.

fo richiedono l'esame dell'intera lista di archi, e questo richiede un tempo che è $O(m)$ nel caso peggiore. Inoltre, a meno che l'insieme degli archi non sia organizzato in maniera particolarmente sofisticata, sembra anche difficile accedere efficientemente ad un particolare arco.

Rappresentazione di grafi con liste di adiacenza. Una rappresentazione più utilizzata della precedente è quella con *liste di adiacenza*. In tale rappresentazione, ogni vertice v ha una lista contenente i suoi vertici adiacenti, ovvero tutti i vertici u per cui esiste un arco (v, u). Notiamo che gli archi non appaiono esplicitamente in questa rappresentazione, ma sono codificati nella nozione di adiacenza. La Figura 11.3 (c) mostra una rappresentazione con liste di adiacenza del grafo non orientato in Figura 11.3 (a). In modo del tutto simile, la Figura 11.4 (c) illustra una rappresentazione con liste di adiacenza del grafo orientato in Figura 11.4 (a).

Notiamo che in questa rappresentazione diventa molto più semplice trovare gli archi incidenti su un particolare vertice v: per far questo è sufficiente scandire la lista di adiacenza di v. Osserviamo anche che ci sono n liste di adiacenza, una per ogni vertice. Nel caso di grafi non orientati la lunghezza totale delle liste di adiacenza è esattamente $2m$, visto che ogni arco (x, y) è rappresentato due volte, una nella lista di adiacenza di x e l'altra nella lista di adiacenza di y. Nel caso di grafi orientati la lunghezza totale delle liste di adiacenza è invece m, visto che ogni arco (x, y) è rappresentato una sola volta, ovvero nella lista di adiacenza di x. In entrambi i casi, lo spazio richiesto dalla rappresentazione con liste di adiacenza è ancora $O(m + n)$.

La rappresentazione con lista di adiacenza è efficiente in molte applicazioni in cui si richiede di visitare un grafo esaminando gli archi incidenti sui suoi vertici. D'altro canto, l'operazione che viene realizzata con meno efficienza è verificare se una particolare coppia di vertici, x ed y, sia connessa da un arco. Infatti, questo

Operazione	Tempo di esecuzione
`grado`(v)	$O(\delta(v))$
`archiIncidenti`(v)	$O(\delta(v))$
`sonoAdiacenti`(x, y)	$O(\min\{\delta(x), \delta(y)\})$
`aggiungiVertice`(v)	$O(1)$
`aggiungiArco`(x, y)	$O(1)$
`rimuoviVertice`(v)	$O(m)$
`rimuoviArco`$(e = (x, y))$	$O(\delta(x) + \delta(y))$

Tabella 11.2 Tempi di esecuzione delle operazioni su grafi non orientati rappresentati mediante liste di adiacenza.

richiede di scandire un'intera lista di adiacenza. Ad esempio, su un grafo non orientato, se scandiamo le liste di adiacenza di x ed y in parallelo, un passo alla volta, possiamo realizzare tale operazione in un tempo che è $O(\min\{\delta(x), \delta(y)\})$ nel caso peggiore, dove $\delta(v)$ è il grado del vertice v. Si potrebbe eseguire questa verifica più efficientemente ordinando le liste di adiacenza e memorizzandole in una struttura ad accesso diretto (come ad esempio un array) così da poter utilizzare l'algoritmo di ricerca binaria: tuttavia, così facendo, l'operazione di inserimento di un nuovo arco (x, y) richiederebbe tempo $O(\delta(x) + \delta(y))$ invece che $O(1)$. Un altro potenziale punto di debolezza della rappresentazione con liste di adiacenza è che un generico arco è rappresentato due volte, ed in due liste di adiacenza diverse. Se ad esempio ad ogni arco deve essere associata dell'informazione, potrebbe essere complicato verificare che le due copie dello stesso arco siano sempre consistenti. In sintesi, i tempi di esecuzione della rappresentazione di un grafo non orientato mediante liste di adiacenza sono illustrati nella Tabella 11.2.

Rappresentazione di grafi con liste di incidenza. La rappresentazione con liste di incidenza combina le caratteristiche ed i vantaggi della rappresentazione con lista di archi descritta all'inizio del paragrafo e della rappresentazione con liste di adiacenza. In particolare, in aggiunta alla rappresentazione con lista di archi, basata su strutture per ogni arco e per ogni vertice, memorizziamo per ogni vertice v una lista di puntatori agli archi incidenti a v. La quantità di spazio richiesta è leggermente superiore a quella delle precedenti rappresentazioni, ma è ancora $O(m + n)$ nel caso peggiore. Nella Figura 11.3 (d) viene visualizzata una rappresentazione con liste di incidenza del grafo non orientato in Figura 11.3 (a). In modo del tutto analogo, la Figura 11.4 (d) mostra una rappresentazione con liste di incidenza del grafo orientato in Figura 11.4 (a). I tempi di esecuzione delle principali operazioni su grafi non orientati rappresentati mediante liste di incidenza sono illustrati nella Tabella 11.3.

Rappresentazione di grafi con matrici di adiacenza. In alcune applicazioni potremmo voler privilegiare una rappresentazione del grafo che richieda più

Operazione	Tempo di esecuzione
grado(v)	$O(\delta(v))$
archiIncidenti(v)	$O(\delta(v))$
sonoAdiacenti(x, y)	$O(\min\{\delta(x), \delta(y)\})$
aggiungiVertice(v)	$O(1)$
aggiungiArco(x, y)	$O(1)$
rimuoviVertice(v)	$O(m)$
rimuoviArco($e = (x, y)$)	$O(\delta(x) + \delta(y))$

Tabella 11.3 Tempi di esecuzione delle operazioni su grafi non orientati rappresentati mediante liste di incidenza.

spazio, ma per cui la verifica della presenza di un arco possa essere effettuata più velocemente. In questi casi, possiamo utilizzare una rappresentazione del grafo tramite *matrice di adiacenza*. Per comodità, assumiamo che i vertici del grafo che dobbiamo rappresentare corrispondano ai numeri interi da 1 ad n. Se non è così osserviamo che è sempre possibile definire una corrispondenza biunivoca tra gli n vertici di un grafo e gli interi da 1 ad n. Sia quindi M una matrice, di dimensione $n \times n$, le cui righe e colonne sono "indicizzate" dai vertici del grafo. La matrice di adiacenza M è definita nel modo seguente:

$$M[u, v] = \begin{cases} 1 & \text{se } (u, v) \text{ è un arco di } G \\ 0 & \text{altrimenti} \end{cases}$$

La Figura 11.3 (e) mostra una rappresentazione con matrice di adiacenza del grafo non orientato in Figura 11.3 (a). In modo del tutto analogo, la Figura 11.4 (e) illustra una rappresentazione con matrice di adiacenza del grafo orientato in Figura 11.4 (a).

Osserviamo che nel caso di grafi non orientati, la matrice di adiacenza è simmetrica, dato che $M[x, y] = M[y, x]$. Un punto di forza della rappresentazione con matrici di adiacenza è che possiamo verificare la presenza di un arco (x, y) in tempo costante, controllando semplicemente il valore memorizzato in $M[x, y]$. D'altro canto, però, trovare i vicini di un vertice v diventa un'operazione costosa: dobbiamo infatti esaminare tutte le posizioni $M[v, \cdot]$ della matrice, ovvero scandirne un'intera riga, anche nel caso in cui ci sono pochi vicini di v. Questa operazione richiede quindi tempo $O(n)$. Osserviamo che per memorizzare informazioni ausiliarie, come ad esempio il costo o la lunghezza di un arco, possiamo sempre utilizzare altre matrici oltre alla matrice di adiacenza. I tempi di esecuzione delle principali operazioni su grafi non orientati rappresentati mediante matrici di adiacenza sono illustrati nella Tabella 11.4. Osserviamo che i tempi relativi alle operazioni aggiungiVertice(v) e rimuoviVertice(v) tengono in conto anche il tempo necessario alla riallocazione della matrice di adiacenza.

La rappresentazione mediante matrice di adiacenza è particolarmente utile in

Operazione	Tempo di esecuzione
grado(v)	$O(n)$
archiIncidenti(v)	$O(n)$
sonoAdiacenti(x, y)	$O(1)$
aggiungiVertice(v)	$O(n^2)$
aggiungiArco(x, y)	$O(1)$
rimuoviVertice(v)	$O(n^2)$
rimuoviArco(e)	$O(1)$

Tabella 11.4 Tempi di esecuzione delle operazioni su grafi non orientati rappresentati mediante matrici di adiacenza.

calcoli algebrici. Ad esempio, la matrice di adiacenza codifica tutti i cammini di lunghezza 1 nel grafo (ovvero gli archi). Eseguendo il prodotto

$$M^2 = M \cdot M$$

otterremo invece informazione sui cammini di lunghezza 2. Infatti, per la definizione stessa di moltiplicazione di due matrici:

$$M^2[u, v] = \sum_{x \in V} (M[u, x] \cdot M[x, v])$$

Quindi $M^2[u, v]$ avrà valore diverso da zero se e soltanto se esisterà almeno un vertice y tale che $M[u, y] = M[y, v] = 1$, ovvero se e solo se esiste almeno un cammino tra i vertici u e v contenente due archi. Procedendo per induzione, si può dimostrare facilmente che per la potenza k-esima di M avremo che $M^k[u, v] \neq 0$ se e soltanto se i due vertici u e v sono connessi da almeno un cammino composto da k archi.

Rappresentazione di grafi con matrici di incidenza. La *matrice di incidenza* è una matrice di dimensione $n \times m$: le righe della matrice sono "indicizzate" dai vertici mentre le colonne sono "indicizzate" dagli archi. Similmente al caso della matrice di adiacenza, la matrice di incidenza avrà un valore uguale ad 1 quando arco e vertice corrispondenti sono incidenti. Di conseguenza, ogni colonna ha esattamente due 1: in particolare, la colonna corrispondente all'arco (x, y) avrà valore uguale ad 1 solamente nelle due righe corrispondenti ad x e ad y. Nel caso di un grafo orientato, possiamo usare una matrice simile, in cui ogni colonna ha esattamente un valore $+1$ ed un valore -1 per distinguere tra il vertice sorgente ed il vertice destinazione dell'arco. Osserviamo che la matrice di incidenza non è necessariamente una matrice quadrata, e quindi neanche simmetrica.

Nella Figura 11.3 (f) viene visualizzata una rappresentazione con matrice di incidenza del grafo non orientato in Figura 11.3 (a). In modo del tutto analogo,

la Figura 11.4 (f) mostra una rappresentazione con matrice di incidenza del grafo orientato in Figura 11.4 (a). I tempi di esecuzione delle principali operazioni su grafi non orientati rappresentati mediante matrici di incidenza sono illustrati nella Tabella 11.5.

Operazione	Tempo di esecuzione
`grado`(v)	$O(m)$
`archiIncidenti`(v)	$O(m)$
`sonoAdiacenti`(x, y)	$O(m)$
`aggiungiVertice`(v)	$O(nm)$
`aggiungiArco`(x, y)	$O(nm)$
`rimuoviVertice`(v)	$O(nm)$
`rimuoviArco`(e)	$O(n)$

Tabella 11.5 Tempi di esecuzione delle operazioni su grafi non orientati rappresentati mediante matrici di incidenza.

11.3 Visite di grafi

L'attraversamento o visita di un grafo è un problema di base che si presenta, anche come sottoproblema, in molte applicazioni. Intuitivamente, ci aspettiamo che la visita di un grafo G ci consenta di esaminare i vertici e gli archi di G in un modo sistematico. Proponiamo adesso un particolare algoritmo per la visita di un grafo non orientato, che chiamiamo `visitaGenerica`, e di cui presentiamo lo pseudocodice in Figura 11.5.

Un primo problema che dobbiamo affrontare in un algoritmo di visita è come verificare se un vertice non sia già stato visitato: questo allo scopo di evitare di dover rivisitare più volte, e inutilmente, lo stesso vertice. Per risolvere questo problema, possiamo semplicemente associare ad ogni vertice un bit di "marcatura": tale bit avrà valore 1 se il vertice è già stato visitato dall'algoritmo (i.e., appartiene già all'albero T), ed avrà valore 0 altrimenti. All'inizio, tutti i bit di marcatura saranno inizializzati a 0 (riga 1 dell'algoritmo `visitaGenerica` di Figura 11.5): quando l'algoritmo visiterà un vertice per la prima volta, cambierà in 1 il valore del bit di marcatura (riga 10).

Una visita del grafo G esaminerà i vertici di G in un determinato ordine. Durante la sua esecuzione, l'algoritmo genererà un albero T contenente i vertici visitati fino a quel punto, e manterrà un insieme $F \subseteq T$ di vertici che costituiscono la "frangia" di T, definita nel modo seguente. Se un vertice v appartiene a $T - F$, allora anche tutti gli archi di G incidenti su v sono già stati esaminati dall'algoritmo. In tal caso diremo che il vertice v è stato "chiuso". Altrimenti, un vertice $v \in T$ appartiene alla frangia F se esistono ancora degli archi incidenti su v che non sono stati esaminati dall'algoritmo: in tal caso diremo che il vertice è ancora "aperto". Notiamo che questo partiziona i vertici del grafo in tre insiemi: i


```
    algoritmo visitaGenerica(vertice s) → albero
1.    rendi tutti i vertici non marcati
2.    T ← albero formato da un solo nodo s
3.    F ← insieme vuoto di vertici
4.    marca il vertice s e aggiungi s a F
5.    while (F ≠ ∅) do
6.        estrai un qualsiasi vertice u da F
7.        visita il vertice u
8.        for each ( arco (u,v) in G ) do
9.            if ( v non è ancora marcato ) then
10.               marca il vertice v e aggiungi v a F
11.               rendi u padre di v in T
12.           else eventualmente rendi u nuovo padre di v in T
13.   return T
```

Figura 11.5 Algoritmo di visita generica in un grafo non orientato G. La riga 12 è eseguita opzionalmente, in base alle modalità di visita che si vuole realizzare; ne vedremo l'utilità nei Capitoli 12 e 13.

vertici $v \in T - F$ "chiusi" dall'algoritmo, i vertici $v \in F$ "aperti" della frangia, ed i vertici $v \in G - T$ non ancora visitati dall'algoritmo.

Inizialmente, T ed F conterranno unicamente il vertice di partenza s (rigbe 2–4): s è l'unico vertice "aperto", nessun vertice è stato ancora "chiuso" dall'algoritmo, e tutti gli altri vertici diversi da s non sono ancora stati visitati. Nel generico passo, l'algoritmo `visitaGenerica` selezionerà un qualsiasi vertice "aperto" dalla frangia di T, $u \in F$ (riga 6), e lo chiuderà esaminando tutti gli archi (u, v) incidenti su u (riga 8). Se l'altro estremo v dell'arco (u, v) non è ancora stato incontrato dall'algoritmo (riga 9), allora il vertice v può essere inserito nell'albero T e farà potenzialmente parte della frangia F di T (righe 10–11). Se invece l'altro estremo v dell'arco (u, v) è già stato incontrato dall'algoritmo, allora il vertice v farà già parte dell'albero T: eventualmente potremo soltanto aggiornare l'albero T modificando l'arco che entra in v (riga 12). Quest'ultimo passo, qui non completamente specificato, ci sarà molto utile per ottenere vari algoritmi di visita, che vedremo nei paragrafi successivi, come casi particolari dell'algoritmo `visitaGenerica`.

Osserviamo infine che le operazioni eseguite al passo *visita il vertice u* (riga 7) dipendono dall'algoritmo che sta utilizzando la visita, esattamente come gli algoritmi di visita su alberi, descritti nel Paragrafo 3.3 del Capitolo 3, in generale comportano operazioni sui nodi (ad esempio stampe) che non sono specificate nell'algoritmo di visita. Analizziamo ora le prestazioni dell'algoritmo `visitaGenerica`, in funzione della rappresentazione scelta per il grafo G.

Teorema 11.1 *Sia $G = (V, E)$ un grafo con m archi ed n vertici. L'esecuzione dell'algoritmo* `visitaGenerica` *su G richiede i seguenti tempi:*

- *Se il grafo è rappresentato mediante lista di archi, il tempo di esecuzione è* $O(mn)$.
- *Se il grafo è rappresentato mediante liste di adiacenza, o liste di incidenza, il tempo di esecuzione è* $O(m+n)$.
- *Se il grafo è rappresentato mediante matrice di adiacenza, il tempo di esecuzione è* $O(n^2)$.

Dimostrazione. Supponiamo che tutte le operazioni di marcatura, di inizializzazione delle strutture dati, di inserimento e di eliminazione nell'insieme F che realizza la frangia dell'albero T possano essere implementate in tempo costante. Osserviamo inoltre che ogni vertice può essere inserito nell'insieme F al più una volta. Infatti, quando un vertice viene inserito in F viene marcato (righe 4 e 10), e vertici già marcati non vengono più considerati per un possibile inserimento in F (riga 9). Le operazioni sull'insieme F e la gestione del sistema di marcatura richiederà quindi tempo totale $O(n)$. Oltre a queste operazioni, l'algoritmo dovrà esplorare gli archi incidenti su un dato vertice. Osserviamo che tale esplorazione deve essere eseguita esattamente n volte, una per ogni volta che estraiamo un vertice u dalla frangia F.

Il tempo di esecuzione sarà quindi $O(n)$ più il tempo richiesto per esplorare gli archi incidenti su tutti i vertici nella rappresentazione scelta per il grafo. Indichiamo con $\tau(v)$ il tempo necessario per generare tutti i vicini di un vertice v, ovvero per eseguire l'operazione `archiIncidenti`(v), in funzione della rappresentazione scelta per il grafo. Possiamo quindi affermare che il tempo richiesto dall'algoritmo `visitaGenerica` sarà

$$O\left(n + \sum_{v \in G} \tau(v)\right)$$

- Nel caso di rappresentazione del grafo con lista di archi, come può desumersi dalla Tabella 11.1, l'operazione `archiIncidenti`(v) richiede l'esame di tutti gli archi del grafo. Quindi, utilizzando tale rappresentazione, si avrà $\tau(v) = O(m)$ per ogni vertice v. Di conseguenza, avremo un tempo totale di

$$O\left(n + \sum_{v \in G} \tau(v)\right) = O\left(n + \sum_{v \in G} m\right) = O(n + n \cdot m) = O(mn)$$

per l'algoritmo.
- Nel caso di rappresentazione con liste di adiacenza, o liste di incidenza, tutti i vicini del vertice v possono essere generati in tempo $\tau(v) = O(\delta(v))$, dove $\delta(v)$ è il grado del vertice v, come può desumersi dalle Tabelle 11.2 e 11.3. Sommando su tutti i vertici v, e ricordando che $\sum_{v \in G} \delta(v) = 2m$, otteniamo

$$O\left(n + \sum_{v \in G} \tau(v)\right) = O\left(n + \sum_{v \in G} \delta(v)\right) = O(n + 2m) = O(m + n)$$

Quindi in questa rappresentazione il costo totale dell'algoritmo sarà $O(m+n)$.

- Infine, nel caso di rappresentazione del grafo mediante matrice di adiacenza, l'operazione `archiIncidenti`(v) richiede la scansione di un'intera riga della matrice, e quindi un tempo $\tau(v) = O(n)$ per ogni vertice v, come può desumersi dalla Tabella 11.4. Quindi, con questa rappresentazione, il tempo totale dell'algoritmo sarà

$$O\left(n + \sum_{v \in G} \tau(v)\right) = O\left(n + \sum_{v \in G} n\right) = O(n + n^2) = O(n^2)$$

□

Il Teorema 11.1 ci suggerisce che la rappresentazione più efficiente da utilizzare per algoritmi di visita in un grafo è quella con liste di adiacenza (o di incidenza). Nel resto del capitolo, assumeremo quindi di rappresentare un grafo mediante liste di adiacenza.

Concludiamo questo paragrafo osservando che nella nostra analisi dell'algoritmo `visitaGenerica`, non abbiamo prestato molta attenzione all'ordine in cui abbiamo inserito e cancellato gli archi incidenti su T nell'insieme F. Ordini diversi di esame dei vertici in F producono visite diverse e conseguentemente alberi diversi. In particolare, se l'insieme F è gestito con una struttura dati coda, così come definita nel Capitolo 3, dove gli inserimenti avvengono ad un estremo di F e le cancellazioni all'altro estremo, otterremo un algoritmo di visita in ampiezza (*"breadth first search"*). In tal caso, vedremo che nella visita ogni vertice v sarà sempre esaminato il più vicino possibile al vertice di partenza s. In altri termini, una tale visita genera cammini minimi, nel senso del numero minimo di archi, da s agli altri vertici del grafo. Se invece l'insieme F è gestito con una struttura dati pila, così come definita nel Capitolo 3, dove gli inserimenti e le cancellazioni avvengono allo stesso estremo di F, otterremo un un algoritmo di visita in profondità (*"depth first search"*). Riesamineremo con maggiore attenzione le visite in ampiezza ed in profondità nei prossimi paragrafi, esaminando anche le modalità di costruzione dell'albero T e le sue proprietà.

11.3.1 Visita in ampiezza

Sia $G = (V, E)$ un grafo non orientato. Una visita in ampiezza di G, a partire da un vertice assegnato s, esamina i vertici di G in un ordine ben definito, generando un albero T che chiameremo *albero BFS* (da *breadth-first search*). Intuitivamente, la visita procede per ampiezza sul grafo a partire dai vertici incontrati: nell'albero BFS generato dalla visita, ogni vertice si trova il più vicino possibile alla radice s dell'albero T.

Come già abbiamo anticipato, un algoritmo di visita in ampiezza può essere ottenuto dal generico algoritmo di visita illustrato nella Figura 11.5, gestendo l'insieme di vertici F appartenenti alla frangia dell'albero T mediante una struttura dati coda, in cui i vertici sono inseriti da un estremo (operazione enqueue) e rimossi dall'altro (operazione dequeue). Osserviamo che la coda F registra implicitamente l'ordine in cui i vertici sono esaminati dall'algoritmo: i vertici in

testa a F sono stati esaminati per primi, mentre i vertici in coda ad F sono stati esaminati per ultimi. Nel generico passo, l'algoritmo seleziona il vertice in testa alla coda, ovvero il vertice che si trova da più tempo nella frangia F dell'albero T. Lo pseudocodice può quindi essere ottenuto direttamente dall'algoritmo `visitaGenerica` della Figura 11.5, come illustrato in Figura 11.7. In questo caso non abbiamo bisogno della riga 12 perché il padre di un nodo u nell'abero BFS può essere deciso nel momento in cui u viene incontrato per la prima volta (corrispondente all'istante della sua marcatura). Se vi sono più archi incidenti su un vertice v, possiamo esaminarli in un ordine qualsiasi: il più semplice è probabilmente l'ordine dato dalla lista di adiacenza di v. Un esempio di visita in ampiezza di un grafo non orientato è mostrato in Figura 11.6.

Proprietà della visita in ampiezza. Elenchiamo ora alcune semplici proprietà dell'algoritmo di visita BFS. Prima di tutto, esattamente come nel caso dell'algoritmo `visitaGenerica`, di cui l'algoritmo `visitaBFS` è una specializzazione, ogni vertice può essere marcato al più una volta, aggiunto alla coda al più una volta, e quindi cancellato dalla coda al più una volta. Dato che il tempo per esaminare un vertice è ancora proporzionale alla lunghezza della sua lista di adiacenza, il tempo totale dell'algoritmo di BFS su un grafo memorizzato con liste di adiacenza sarà $O(m + n)$, esattamente come dimostrato nell'analisi del Teorema 11.1.

Dati due vertici x e y in un grafo non orientato $G = (V, E)$, indichiamo con $d(x, y)$ la lunghezza di un cammino col minimo numero di archi tra x e y in G. Sia inoltre s un vertice in G, e T l'albero BFS prodotto a partire da s dall'algoritmo `visitaBFS` in Figura 11.7. Dato un vertice v in G, utilizziamo inoltre la notazione $\ell(v)$ per indicare il livello (profondità) di v in T. Se $v \notin T$, allora $\ell(v) = +\infty$. Gli alberi BFS godono di proprietà molto interessanti. In particolare, il Lemma 11.1 caratterizza la composizione della coda F durante l'esecuzione dell'algoritmo `visitaBFS` illustrato in Figura 11.7.

Lemma 11.1 *Sia $G = (V, E)$ un grafo non orientato. Se durante l'esecuzione dell'algoritmo* `visitaBFS` *la coda* F *contiene i vertici $\{v_1, v_2, \ldots, v_p\}$, con il vertice v_1 in testa alla coda, allora $\ell(v_i) \leq \ell(v_{i+1})$ per $1 \leq i \leq p - 1$, e $\ell(v_p) \leq \ell(v_1) + 1$.*

Dimostrazione. Dimostreremo il lemma per induzione sul numero di operazioni effettuate sulla coda F. All'inizio (riga 5 di Figura 11.7), la coda F contiene solamente il vertice s e dunque il lemma è banalmente verificato. Dimostriamo ora il passo induttivo. Sia $\{v_1, v_2, \ldots, v_p\}$, con $p \geq 1$, il contenuto della coda F prima di effettuare un'operazione. Per l'ipotesi induttiva, avremo:

$$\ell(v_i) \leq \ell(v_{i+1}) \quad \text{per ogni } i,\ 1 \leq i \leq p-1 \tag{11.1}$$

$$\ell(v_p) \leq \ell(v_1) + 1 \tag{11.2}$$

Abbiamo quindi due possibilità:

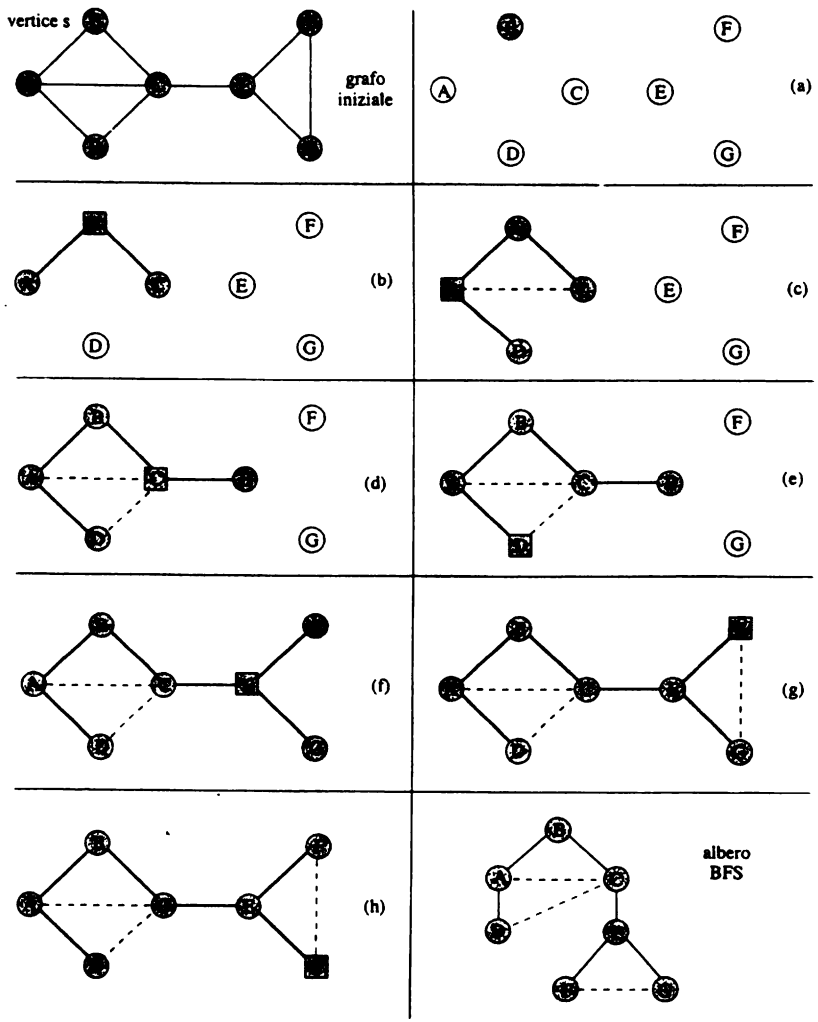


Figura 11.6 Visita in ampiezza di un grafo non orientato: assumiamo che i vicini di ogni vertice siano presi in considerazione secondo l'ordine lessicografico.

```
    algoritmo visitaBFS(vertice s) → albero
1.    rendi tutti i vertici non marcati
2.    T ← albero formato da un solo nodo s
3.    Coda F
4.    marca il vertice s
5.    F.enqueue(s)
6.    while ( not F.isempty() ) do
7.      u ← F.dequeue()
8.      for each ( arco (u, v) in G ) do
9.        if ( v non è ancora marcato ) then
10.         F.enqueue(v)
11.         marca il vertice v
12.         rendi u padre di v in T
13.   return T
```

Figura 11.7 Algorifmo di vlsifa in ampiezza in un grafo non orienfato G.

- Se l'operazione su F è una dequeue (riga 7 di Figura 11.7), o la coda rimane vuota (e il passo induttivo è banalmente verificato) oppure dopo la dequeue la coda contiene i vertici $\{v_2, \ldots, v_p\}$: in questo caso, dalle disuguaglianze (11.1) e (11.2) segue che $\ell(v_i) \leq \ell(v_{i+1})$ per $2 \leq i \leq p-1$ e $\ell(v_p) \leq \ell(v_1) + 1 \leq \ell(v_2) + 1$, e quindi il passo induttivo sarà ancora verificato dopo la dequeue (con v_2 in testa alla coda).
- Se l'operazione su F è una enqueue, sia v_{p+1} il vertice aggiunto in coda ad F alla riga 10 di Figura 11.7. Questo è possibile soltanto se alla riga 7 di Figura 11.7 abbiamo appena estratto dalla coda F un vertice v_0, per cui esiste l'arco (v_0, v_{p+1}). Quindi, per l'ipotesi induttiva, avremo $\ell(v_i) \leq \ell(v_{i+1})$ per $0 \leq i \leq p-1$, $\ell(v_p) \leq \ell(v_1) + 1$ ed $\ell(v_p) \leq \ell(v_0) + 1$. Inoltre, poché v_0 viene reso padre di v_{p+1} in T, per definizione di livello ℓ risulta $\ell(v_{p+1}) = \ell(v_0) + 1$. Avremo quindi $\ell(v_i) \leq \ell(v_{i+1})$ per $1 \leq i \leq p$, e $\ell(v_{p+1}) = \ell(v_0) + 1 \leq \ell(v_1) + 1$, e di conseguenza il passo induttivo sarà verificato anche dopo l'operazione enqueue.

□

Il Lemma 11.1 ci assicura quindi che, ad ogni istante, i vertici contenuti nella coda F possono appartenere al più a due livelli adiacenti dell'albero BFS. Il Lemma 11.2 ci garantisce invece che in un albero BFS i nodi vengono generati al livello più vicino possibile alla radice.

Lemma 11.2 *Siano dati un grafo non orientato e connesso $G = (V, E)$, un vertice s in G, ed un albero T prodotto dall'algoritmo* visitaBFS(s). *Per ogni vertice v, risulta $\ell(v) = d(s, v)$.*

Dimostrazione. Sia v un qualsiasi vertice di G. Dato che T contiene archi di G, il cammino in T da s a v deve necessariamente corrispondere ad un cammino da

s a v nel grafo G. Questo implica che $\ell(v) \geq d(s, v)$. Per dimostrare il lemma, sarà quindi sufficiente dimostrare che $\ell(v) \leq d(s, v)$. A tal scopo, per $k \geq 0$ definiamo l'insieme dei vertici a distanza k da s:

$$V_k = \{w \in V \ : \ d(s, w) = k\}$$

Per completare il lemma, dimostriamo che, ad un certo istante, per qualsiasi vertice $v \in V_k$, avremo che:

(1) v viene inserito nella coda F
(2) $\ell(v) \leq k$.

La dimostrazione procede per induzione su k. La base dell'induzione è banale, dato che per $k = 0$, $V_0 = \{s\}$, $d(s, s) = 0$, e il vertice s viene inserito nella coda F alla riga 5 con $\ell(s) = 0$. Supponiamo ora che l'ipotesi induttiva sia vera per $(k-1)$ e dimostriamola per k.

Sia v un generico vertice in V_k. Dato che $d(s, v) = k$, deve esistere almeno un vertice w tale che $d(s, w) = k - 1$ e $(w, v) \in E$. Sia quindi

$$U_{k-1} = \{w \in V_{k-1} \ : \ (w, v) \in E\}$$

e sia u il primo vertice di U_{k-1} ad essere inserito nella coda F. Per definizione di coda, u sarà anche il primo vertice di U_{k-1} ad essere estratto dalla coda F alla riga 7. Questo implica che, nell'istante in cui u è estratto dalla coda, il vertice v non è ancora marcato, e quindi sarà inserito nella coda F alla riga 10. Inoltre, per l'ipotesi induttiva, $\ell(u) \leq k - 1$: di conseguenza, visto che v viene reso figlio di u alla riga 12, avremo

$$\ell(v) = \ell(u) + 1 \leq (k-1) + 1 = k$$

□

Lemma 11.3 *Sia dato un grafo non orientato e connesso $G = (V, E)$ ed un albero BFS di G. Sia (x, y) un arco di G: allora x ed y appartengono allo stesso livello oppure a livelli consecutivi dell'albero BFS.*

Dimostrazione. Supponiamo per contraddizione che esista un arco (x, y) di G tale che $\ell(x) < \ell(y) - 1$. La presenza dell'arco (x, y) implica che

$$d(s, y) \leq d(s, x) + 1 \tag{11.3}$$

Per il Lemma 11.2, abbiamo che $d(s, x) = \ell(x)$ e $d(s, y) = \ell(y)$, e conseguentemente $d(s, x) = \ell(x) < \ell(y) - 1 = d(s, y) - 1$, ovvero

$$d(s, y) > d(s, x) + 1 \tag{11.4}$$

La diseguaglianza (11.4) è chiaramente in contraddizione con la (11.3): conseguentemente, non può esistere nessun arco (x, y) di G tale che $\ell(x) < \ell(y) - 1$.

□

La seguente proprietà è una conseguenza immediata del Lemma 11.3.

```
algoritmo preordine(nodo v)
    visita il nodo v
    for each ( figlio w di v ) do
        preordine(w)
```

Figura 11.8 Algoritmo di visita in preordine di un albero a partire dalla sua radice v.

Proprietà 11.1 *Ogni arco di un grafo non orientato G su cui si effettua la visita in ampiezza può essere classificato in tre gruppi rispetto all'albero BFS prodotto:*

1. *archi dell'albero BFS;*
2. *archi tra vertici allo stesso livello dell'albero BFS;*
3. *archi tra livelli adiacenti dell'albero BFS.*

11.3.2 Visita in profondità

Sia $G = (V, E)$ un grafo non orientato. Una visita in profondità di G, a partire da un vertice assegnato s, esamina i vertici di G generando un albero T che chiameremo *albero DFS* (da *depth-first search*). Intuitivamente, la visita procede in profondità sul grafo a partire dai vertici incontrati. Osserviamo che la visita in profondità di un grafo è sostanzialmente analoga alla visita in preordine di un albero incontrata nel Paragrafo 3.3.3 del Capitolo 3, ed in particolare definita su alberi binari nello pseudo-codice della Figura 3.14. Qui lo estendiamo per completezza al caso di alberi di grado qualsiasi, come illustrato nella Figura 11.8.

Per trasformare la visita in preordine su un albero in un algoritmo di visita su grafi, è sufficiente sostituire al termine "figlio" il termine "vicino". Per evitare cicli infiniti, dobbiamo riuscire a garantire che visiteremo ogni vertice soltanto una volta. Esattamente come nei casi precedenti, possiamo semplicemente utilizzare un sistema di marcatura. L'algoritmo `visitaDFSRicorsiva` è illustrato in Figura 11.9, e un esempio di visita in profondità di un grafo non orientato è mostrato in Figura 11.10.

Esattamente come nel caso della visita in ampiezza, possiamo classificare gli archi del grafo G, rispetto all'albero DFS generato. In particolare, il seguente lemma esclude completamente la possibilità di avere un arco tra vertici x e y che si trovano in differenti sottoalberi dell'albero DFS.

Lemma 11.4 *Siano dati un grafo non orientato $G = (V, E)$ ed un albero DFS di G. Sia (v, w) un arco di G. Allora*

1. *(v, w) è un arco dell'albero DFS; oppure*
2. *v e w sono antenato e discendente nell'albero DFS.*


```
procedura visitaDFSRicorsiva(vertice v, albero T)
    marca e visita il vertice v
    for each ( arco (v, w) ) do
        if ( w non è marcato ) then
            aggiungi l'arco (v, w) all'albero T
            visitaDFSRicorsiva(w, T)

algoritmo visitaDFS(vertice s) → albero
    T ← albero vuoto
    visitaDFSRicorsiva(s, T)
    return T
```

Figura 11.9 Algoritmo ricorsivo per la visita in profondità di un grafo a partire da un vertice v.

Dimostrazione. Supponiamo per assurdo che esista un arco (v, w) che non appartiene all'albero DFS e tale che v e w non sono l'uno antenato o discendente dell'altro. Senza perdita di generalità, supponiamo inoltre di utilizzare la visita in profondità ricorsiva illustrata in Figura 11.9, e che in tale visita abbiamo visitato il vertice v prima di w. Sotto tali ipotesi, l'unico motivo per cui l'arco (v, w) non può essere stato inserito come arco dell'albero DFS è che il vertice w sia già stato visitato prima di esaminare l'arco (v, w), ovvero durante una delle chiamate ricorsive generate a partire da v. In questo caso, però, v sarebbe un antenato di w, chiaramente una contraddizione con l'ipotesi che i vertici v e w non sono l'uno antenato dell'altro. □

Per simmetria rispetto alla Proprietà 11.1, riformuliamo il Lemma 11.4 nel modo seguente:

Proprietà 11.2 *Ogni arco di un grafo non orientato G su cui si effettua la visita in profondità può essere classificato in due gruppi rispetto all'albero DFS prodotto:*

1. *archi dell'albero DFS;*
2. *archi tra un antenato ed un discendente dell'albero DFS.*

Illustriamo ora, a titolo puramente esemplificativo, come utilizzare la classificazione degli archi rispetto all'albero DFS illustrata dalla Proprietà 11.2 per inferire altre proprietà del grafo, apparentemente non correlate con la visita in profondità.

Lemma 11.5 *Sia $G = (V, E)$ un grafo non orientato, con m archi ed n vertici, e sia k un qualsiasi intero, $0 < k < n$. Se non esiste in G alcun cammino con k o più archi, allora G ha al più $O(kn)$ archi.*

Dimostrazione. Sia T l'albero DFS generato da una visita in profondità di G. Visto che gli archi in T sono al più $(n - 1)$, per avere una stima asintotica del

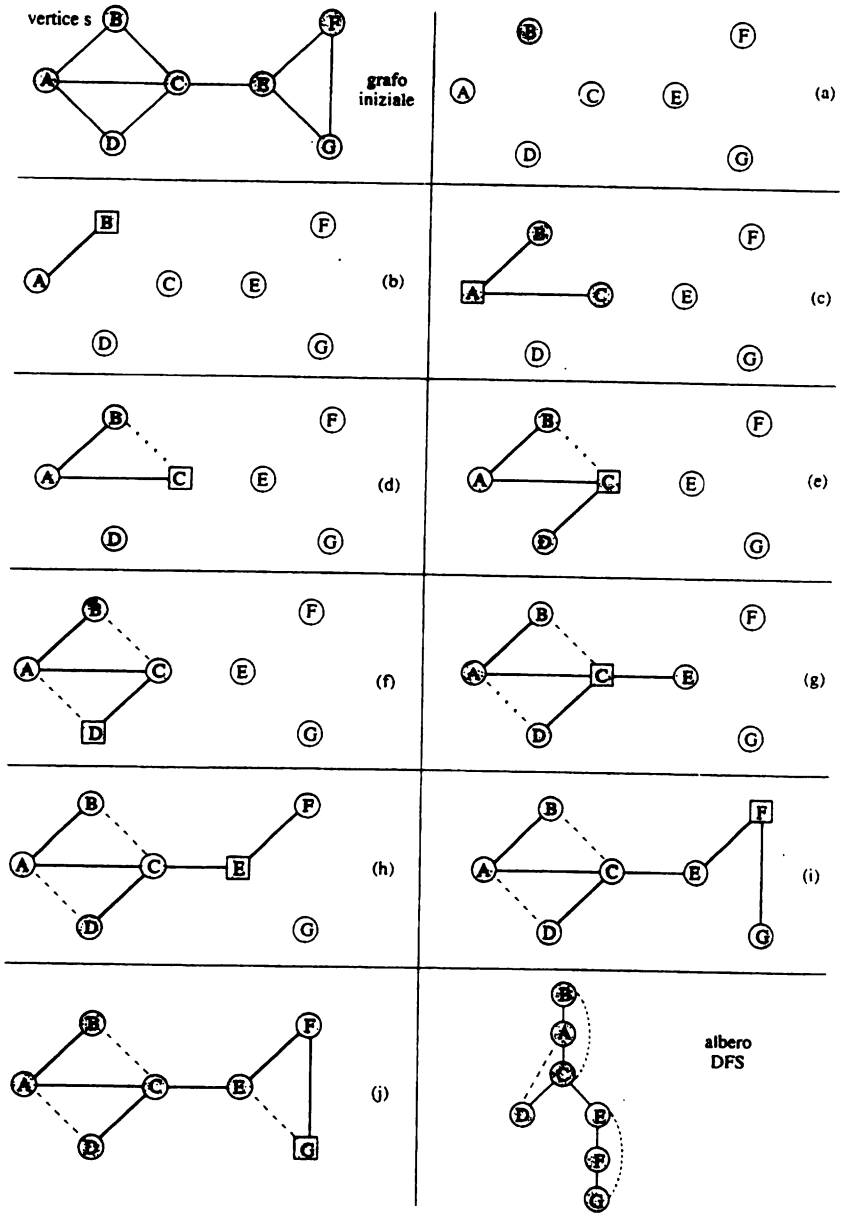


Figura 11.10 Visita in profondità di un grafo non orientato: assumiamo che i vicini di ogni vertice siano presi in considerazione secondo l'ordine lessicografico.

numero di archi in G sarà sufficiente delimitare superiormente il numero di archi che non fanno parte dell'albero DFS, ovvero gli archi in $G - T$.

Dato che G non ha cammini con k o più archi, allora anche ogni cammino dalla radice ad un qualsiasi vertice in T sarà di lunghezza strettamente minore di k. In base alla Proprietà 11.2, gli estremi di ogni arco di $G - T$ devono necessariamene essere antenato e discendente nell'albero T. Sia x un qualsiasi vertice del grafo: osserviamo che x si trova al più a distanza k dalla radice dell'albero, e quindi ha al più $(k - 1)$ antenati in T. Questo implica che il numero totale di archi in $G - T$ è al più $O(kn)$, e conclude la dimostrazione. □

Concludiamo il paragrafo osservando che si può derivare un algoritmo iterativo di visita in profondità, che è in sostanza molto simile all'algoritmo iterativo per la visita in ampiezza presentato in Figura 11.7. La differenza principale con una visita in ampiezza è nella struttura dati utilizzata per gestire la frangia F dell'albero T. In particolare, a differenza di una visita in ampiezza, nella visita in profondità F viene gestita mediante una struttura dati pila, in cui inserimenti e cancellazioni avvengono allo stesso estremo. In altri termini, una visita in ampiezza riparte sempre dal vertice inserito per primo nell'insieme F, (ovvero in *testa* ad F), mentre una visita in profondità riparte sempre dall'ultimo vertice inserito in F (ovvero in *coda* ad F). L'implementazione di questo algoritmo può essere ottenuta modificando opportunamente l'algoritmo `visitaGenerica` di Figura 11.5, ed è lasciata come utile esercizio al lettore (Problema 11.6).

11.3.3 Visite in ampiezza ed in profondità su grafi orientati

Anche se sono state definite su grafi non orientati, le visite in ampiezza ed in profondità possono essere definite anche su grafi orientati. L'unica differenza è nell'operazione di visita dei vicini di un vertice v: in tal caso, vogliamo infatti considerare solo gli archi (v, w) uscenti da v (ed ovviamente ignorare gli archi entranti in v). Anche per grafi orientati, gli alberi BFS e DFS godono di molte proprietà interessanti.

Dato un albero BFS, e due livelli ℓ_1 ed ℓ_2, diremo che il livello ℓ_1 è *precedente* al livello ℓ_2 se ℓ_1 è più vicino alla radice dell'albero. In tal caso diremo anche che il livello ℓ_2 è *successivo* al livello ℓ_1. Possiamo ora classificare gli archi in quattro classi rispetto ad un albero BFS ottenuto su un grafo orientato.

Proprietà 11.3 *Ogni arco di un grafo orientato G su cui si effettua una visita in ampiezza può essere classificato in quattro gruppi rispetto all'albero BFS prodotto:*

1. *archi dell'albero BFS;*
2. *archi i cui estremi sono nello stesso livello BFS;*
3. *archi che vanno da un livello al livello immediatamente successivo;*
4. *archi che vanno da un livello ad un qualsiasi livello precedente.*

Per quanto riguarda la visita in profondità su grafi orientati, possiamo classificare gli archi in quattro classi rispetto ad un albero DFS.

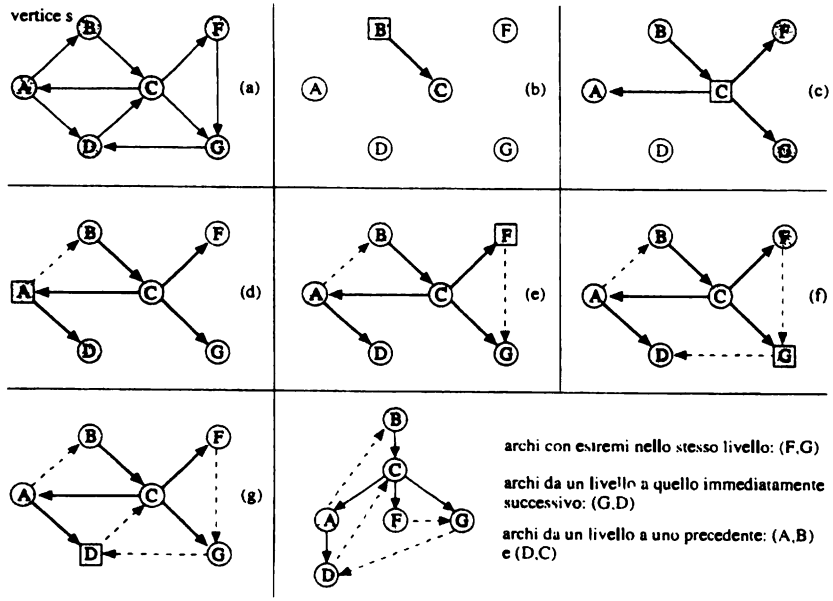


Figura 11.11 Visita in ampiezza di un grafo orientato e corrispondente classificazione degli archi. Assumiamo che i vicini di ogni vertice siano presi in considerazione secondo l'ordine lessicografico.

Proprietà 11.4 *Ogni arco di un grafo orientato G su cui si effettua una visita in profondità può essere classificato in quattro gruppi rispetto all'albero DFS prodotto:*

1. *archi dell'albero DFS;*
2. *archi diretti da un discendente ad un antenato nell'albero DFS* (archi all'indietro)*;*
3. *archi diretti da un antenato ad un discendente, che non siano archi dell'albero DFS* (archi in avanti)*;*
4. *archi che vanno da un vertice x ad un vertice y appartenente ad un sottoalbero visitato precedentemente ad x* (archi trasversali a sinistra)*.*

Notiamo che *non* sono possibili archi della seguente tipologia:

- archi che vanno da un vertice x ad un vertice y, dove il vertice x appartiene ad un sottoalbero che non contiene y e che è stato tutto visitato prima di y (*archi trasversali a destra*).

Le classificazioni degli archi del grafo rispetto agli alberi prodotti, soprattutto DFS, sono importanti per il progetto di molti algoritmi su grafi, come ad esempio

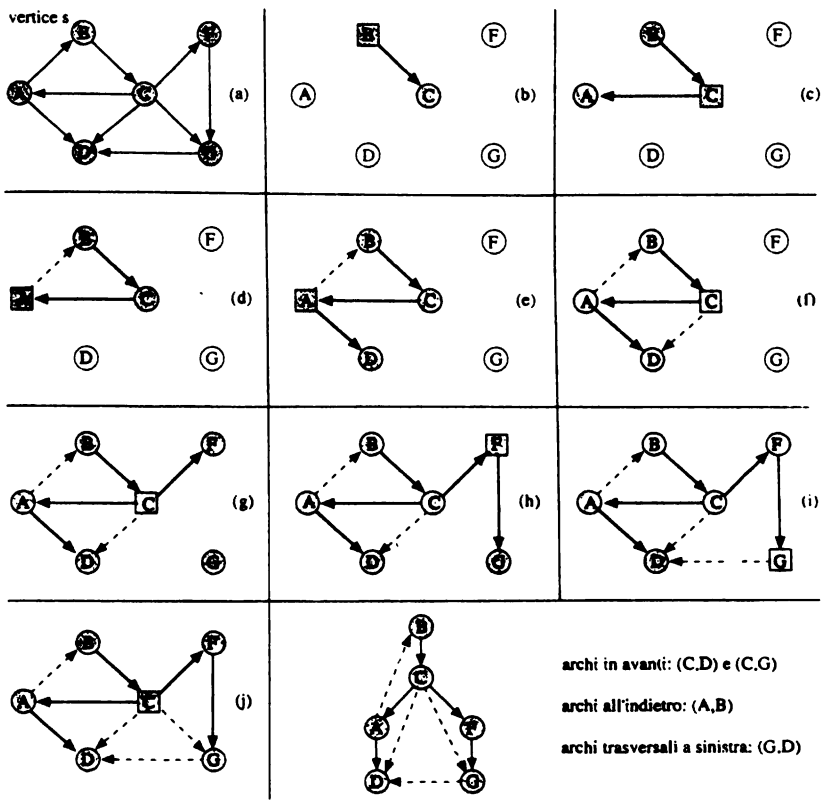


Figura 11.12 Visita in profondità di un grafo orientato e corrispondente classificazione degli archi. Assumiamo che i vicini di ogni vertice siano presi in considerazione secondo l'ordine lessicografico.

il calcolo delle componenti fortemente connesse o il calcolo delle componenti biconnesse di un grafo. Vedremo maggiori dettagli su questo nel Paragrafo 11.5.

Esempi di visita in ampiezza e in profondità di grafi orientati, i corrispondenti alberi BFS e DFS, e la conseguente classificazione degli archi sono mostrati nelle Figure 11.11 e 11.12, rispettivamente.

11.4 Componenti connesse di un grafo non orientato

Come primo esempio di applicazione degli algoritmi di visita su grafi, consideriamo un problema molto semplice: dato un grafo non orientato $G = (V, E)$, vogliamo verificare se G è connesso. Per risolvere questo problema, potremmo partire da un generico vertice s, ed esplorare il grafo. Se, partendo dal vertice s, riusciamo a raggiungere tutti i vertici del grafo G, allora G è connesso. Altrimenti, G non è connesso. Prima di approfondire i dettagli di questo approccio, approfondiamo la terminologia introdotta nel Paragrafo 11.1. Osserviamo innanzitutto che sussiste la seguente proprietà.

Proprietà 11.5 *La relazione di raggiungibilità tra vertici in grafi non orientati è una relazione di equivalenza.*

Dimostrazione. È sufficiente dimostrare che la relazione di raggiungibilità verifica le seguenti proprietà.

(a) *Riflessività*: per tutti i vertici $v \in G$, v è "raggiungibile da" v. Basta osservare che ogni vertice è per definizione raggiungibile da se stesso con un cammino degenere di lunghezza 0.
(b) *Simmetria*: se v è "raggiungibile da" u, allora u è "raggiungibile da" v. Infatti, è sufficiente utilizzare esattamente lo stesso cammino, nella direzione opposta.
(c) *Transitività*: se v è "raggiungibile da" u, e w è "raggiungibile da" v, allora w è "raggiungibile da" u. Infatti, concatenando il cammino da u a v ed il cammino da v a w si ottiene un cammino da u a w.

□

Siamo ora in grado di introdurre la nozione di componente connessa di un grafo.

Definizione 11.2 *Sia dato un grafo $G = (V, E)$ non orientato. Le* componenti connesse *di G sono le classi di equivalenza dei suoi vertici rispetto alla relazione "raggiungibile da".*

È facile verificare che un grafo non orientato è connesso se e solo se ha una sola componente connessa. Seguendo la terminologia della Definizione 11.2, possiamo affermare che la visita di un grafo, secondo gli algoritmi descritti nel Paragrafo 11.3, ci restituisce proprio una componente connessa del grafo. Più in dettaglio, l'albero T restituito dall'algoritmo di visita conterrà tutti e soli i vertici della stessa componente connessa del vertice di partenza s.

Per verificare se un grafo G è connesso, è sufficiente quindi eseguire una visita di G utilizzando ad esempio l'algoritmo `visitaGenerica` di Figura 11.5: se l'albero restituito contiene esattamente n nodi, allora siamo riusciti a raggiungere tutti i vertici, ed il grafo è connesso. Come vedremo meglio nel Capitolo 12, in tal caso l'albero T è un *albero ricoprente* di G. Se invece l'albero restituito contiene meno di n nodi, il grafo originario non è connesso. In questo caso, con un po' più di attenzione, anziché una sola componente connessa, possiamo trovare tutte le componenti connesse del grafo G, facendo partire una nuova visita da un vertice non ancora visitato (si veda il Problema 11.7). Utilizzando l'algoritmo di visita presentato in Figura 11.5, è facile verificare se un dato grafo è connesso come illustrato in Figura 11.13.


```
algoritmo connessoGrafo(grafo G) → booleano
1.   scegli arbitrariamente un vertice s in G
2.   T ← visitaGenerica(s)
3.   if ( T ha n nodi ) then
4.      return connesso
5.   else return non connesso
```

Figura 11.13 Algoritmo che verifica se un grafo G è connesso.

Il tempo di esecuzione dell'algoritmo `connessoGrafo` di Figura 11.13 è dato dal tempo richiesto dall'algoritmo `visitaGenerica`, che abbiamo visto essere $O(m + n)$, più la verifica sul numero di vertici nell'albero T: quest'ultima può essere facilmente eseguita in tempo $O(n)$, utilizzando uno degli algoritmi di visita di alberi descritti nel Capitolo 3.

11.5 Componenti fortemente connesse di un grafo orientato

Ricordiamo dal Paragrafo 11.1 che un grafo orientato G è fortemente connesso se esiste un cammino (orientato) tra ogni coppia di vertici in G. Dimostriamo ora che la relazione di connettività forte è una relazione di equivalenza.

Proprietà 11.6 *La relazione di connettività forte è una relazione di equivalenza.*

Dimostrazione. Ricordando dal Paragrafo 11.1 che la notazione $x \Leftrightarrow y$ indica che il vertice x è fortemente connesso a y, è sufficiente verificare che la relazione gode delle seguenti proprietà.

(a) *Riflessività*: per tutti i vertici $v \in G$, $v \Leftrightarrow v$. Infatti, ogni vertice è per definizione fortemente connesso a se stesso.
(b) *Simmetria*: se $u \Leftrightarrow v$, allora $v \Leftrightarrow u$. Se $u \Leftrightarrow v$, allora esistono un cammino $u \rightsquigarrow v$ ed un cammino $v \rightsquigarrow u$. Per verificare la proprietà di simmetria, è sufficiente utilizzare esattamente gli stessi cammini.
(c) *Transitività*: se $u \Leftrightarrow v$ e $v \Leftrightarrow w$, allora $u \Leftrightarrow w$. Infatti, se $u \Leftrightarrow v$ abbiamo i due cammini $u \rightsquigarrow v$ e $v \rightsquigarrow u$. Se $v \Leftrightarrow w$, abbiamo i due cammini $v \rightsquigarrow w$ e $w \rightsquigarrow v$. Concatenando i cammini a coppie, otteniamo i due cammini $u \rightsquigarrow v \rightsquigarrow w$ e $w \rightsquigarrow v \rightsquigarrow u$. Notiamo che in questo caso è cruciale

che nella definizione di connettività forte tolleriamo il fatto che i cammini possano avere vertici in comune. □

Siamo ora in grado di introdurre la nozione di componente fortemente connessa di un grafo orientato.

Definizione 11.3 *Sia $G = (V, E)$ un grafo orientato. Le* componenti fortemente connesse *di G sono le classi di equivalenza della relazione di connettività forte:*

$$[v] = \{u \in G \mid u \Leftrightarrow v\}$$

Osserviamo che, in base alla definizione di connettività forte, si può dimostrare che due vertici sono fortemente connessi se e solo se fanno parte dello stesso ciclo orientato (si veda il Problema 11.8). Inoltre, è facile verificare che un grafo orientato è fortemente connesso se e solo se ha una sola componente fortemente connessa. In questo paragrafo vedremo come utilizzare l'algoritmo di visita in profondità per calcolare le componenti fortemente connesse di un grafo orientato. Ricordiamo che i due cammini $u \rightsquigarrow v$ e $v \rightsquigarrow u$ tra due vertici fortemente connessi u e v possono avere archi o vertici in comune, o possono essere anche cammini degeneri, come quelli costituiti da un solo vertice: come caso particolare, un vertice è fortemente connesso a se stesso.

Un algoritmo semplice. Il calcolo di una sola componente fortemente connessa non è particolarmente difficile. Infatti, per calcolare la componente fortemente connessa contenente il vertice x, possiamo calcolare preliminarmente gli insiemi:

- i discendenti $D(x)$, ovvero i vertici di G che sono raggiungibili da x;
- gli antenati $A(x)$, ovvero i vertici di G che raggiungono x.

Notiamo che $D(x)$ può essere calcolato in tempo $O(m + n)$ mediante una visita del grafo (sia essa una visita generica, BFS, oppure DFS) a partire dal vertice x. Per calcolare $A(x)$ è sufficiente invertire la direzione di tutti gli archi, ed effettuare ancora una volta una visita a partire da x, ma questa volta nel grafo con gli archi invertiti. Anche in questo caso, il tempo totale è $O(m + n)$.

A questo punto, per calcolare la componente fortemente connessa contenente il vertice x, è sufficiente trovare tutti i vertici che sono in uno stesso ciclo (orientato) con x, ovvero tutti i vertici che sono contemporaneamente discendenti ed antenati di x: $[x] = D(x) \cap A(x)$. Il tempo totale di esecuzione sarà quindi $O(m + n)$. Applicando per n volte questo algoritmo, a partire da ogni vertice $v \in G$, riusciamo a calcolare tutte le componenti fortemente connesse del grafo G in tempo $O(mn + n^2)$. Possiamo fare di meglio?

Un algoritmo con tempo lineare. La risposta è affermativa: presenteremo infatti un algoritmo lineare nella dimensione del grafo, ovvero con tempo di esecuzione $O(m+n)$, per il calcolo delle componenti fortemente connesse. L'algoritmo sfrutta una relazione fondamentale tra connettività forte e proprietà della visita DFS in un grafo orientato: le componenti fortemente connesse di un grafo orientato G appaiono come sottoalberi dell'albero DFS di G. Prima di dimostrare questo risultato, abbiamo bisogno di alcuni lemmi preliminari.

Lemma 11.6 *Sia G un grafo orientato, e siano x e y due vertici di G che appartengono alla stessa componente fortemente connessa. Sia π un cammino orientato di G avente x ed y come estremi, e sia z un qualsiasi vertice di π. Allora z appartiene necessariamente alla stessa componente fortemente connessa di x e y.*

Dimostrazione. Senza ledere la generalità assumiamo che z sia in un cammino (orientato) π_1 da x a y, $x \rightsquigarrow z \rightsquigarrow y$. Siano π_1' il sottocammino $x \rightsquigarrow z$ e π_1'' il sottocammino $z \rightsquigarrow y$. Osserviamo che π_1' fornisce banalmente già un cammino da x a z. Per dimostrare che z è nella stessa componente fortemente connessa di x (e quindi di y), è sufficiente quindi dimostrare che esiste anche un cammino inverso da z a x. Dato che x ed y appartengono alla stessa componente fortemente connessa di G, allora esisterà un cammino π_2 da y a x: $y \rightsquigarrow x$. Combinando π_1'' con π_2 otterremo quindi un cammino da z ad x: $z \rightsquigarrow y \rightsquigarrow x$. □

Siamo ora pronti a dimostrare che una componente fortemente connessa di un grafo orientato G appare come sottoalbero dell'albero DFS generato da una visita in profondità.

Lemma 11.7 *Siano $G = (V, E)$ un grafo orientato, v un qualunque vertice di G, e $[v]$ la componente fortemente connessa di G contenente v. Siano inoltre T un albero DFS di G contenente v, e $T[v]$ il sottografo (non orientato) di T indotto dai vertici in $[v]$. Allora $T[v]$ è un sottoalbero di T, ovvero è connesso.*

Dimostrazione. Supponiamo per assurdo che $T[v]$ non sia connesso. Dati due vertici x ed y in $[v]$ denotiamo con ρ il minimo antenato comune di x ed y in T, ovvero il vertice che è contemporaneamente antenato di x e di y in T (come caso particolare un vertice è antenato di se stesso), e che si trova al livello più profondo in T. Siano inoltre $\pi_{\rho,x}$ il cammino in T da ρ ad x, e $\pi_{\rho,y}$ il cammino in T da ρ ad y. Osserviamo che $\pi_{\rho,x}$ e $\pi_{\rho,y}$ sono cammini (orientati) anche in G.

Se $T[v]$ non è connesso, allora possiamo sempre trovare due vertici x ed y in $[v]$ tali che nel cammino $\pi_{\rho,x}$ oppure nel cammino $\pi_{\rho,y}$ esiste almeno un vertice $z \notin [v]$. A questo punto possono verificarsi tre casi:

(a) $\rho = x$ ($\pi_{\rho,x} = \emptyset$), ed esiste un vertice z in $\pi_{\rho,y} = \pi_{x,y}$, tale che $z \notin [v]$;
(b) $\rho = y$ ($\pi_{\rho,y} = \emptyset$), ed esiste un vertice z in $\pi_{\rho,y} = \pi_{y,x}$, tale che $z \notin [v]$;
(c) $\rho \neq x$ e $\rho \neq y$ ($\pi_{\rho,x} \neq \emptyset$ e $\pi_{\rho,y} \neq \emptyset$), ed esiste un vertice z in $\pi_{\rho,x} \cup \pi_{\rho,y}$, tale che $z \notin [v]$.

Osserviamo preliminarmente che il caso (a) non può verificarsi. Infatti $\pi_{x,y}$ è un cammino orientato, anche nel grafo G, tra due vertici della stessa componente fortemente connessa $[v]$. Per il Lemma 11.6, qualsiasi vertice z di $\pi_{x,y}$ deve necessariamente appartenere a $[v]$. Lo stesso discorso vale per il caso (b): $\pi_{y,x}$ è un cammino orientato, anche in G, tra due vertici della stessa componente fortemente connessa $[v]$, e quindi il Lemma 11.6 esclude la possibilità che esista un vertice $z \notin [v]$ in $\pi_{y,x}$.

Assumiamo quindi di trovarci nel caso (c). Deve necessariamente essere $\rho \notin [v]$, altrimenti, sempre per il Lemma 11.6, tutti i vertici in $\pi_{\rho,x}$ ed in $\pi_{\rho,y}$ apparterrebbero alla componente fortemente connessa $[v]$. Notiamo che questo implica che la stessa componente fortemente connessa $[v]$ si trova almeno in due sottoalberi distinti dell'albero DFS T: un sottoalbero $T_x \subseteq [v]$ contenente x, ed un sottoalbero $T_y \subseteq [v]$ contenente y, con $T_x \cap T_y = \emptyset$ e tale che ρ, il minimo antenato comune di x e y è distinto da x e da y. Senza perdita di generalità assumiamo che T_x si trovi a sinistra di T_y, ovvero x sia stato incontrato prima di y nella visita DFS. Dato che x e y sono nella stessa componente fortemente connessa $[v]$, deve necessariamente esistere in G un cammino π_{xy} da x a y: questo implica la presenza di almeno un "arco trasversale a destra" rispetto all'albero DFS. Come abbiamo visto nel Paragrafo 11.3.3 (Proprietà 11.4), un tale arco non può esistere, e dunque neanche il caso (c) è possibile.

Non potendosi verificare nessuno dei tre casi, allora $T[v]$ deve essere necessariamente un sottoalbero connesso di T. □

Il Lemma 11.7 caratterizza le componenti fortemente connesse di un grafo orientato G in relazione ad un suo albero DFS T: dato che le componenti fortemente connesse di G sono sottoalberi di T, per trovarle potremmo utilizzare come "scheletro" della nostra visita del grafo proprio l'albero DFS. In particolare, partizioneremo T "tagliandolo" in punti opportuni, così da lasciarne solo i sottoalberi corrispondenti alle componenti fortemente connesse di G. Ma quali sono gli archi che dobbiamo tagliare per evidenziare la partizione del grafo nelle sue componenti fortemente connesse? La seguente definizione ci può essere di aiuto.

Definizione 11.4 *Dati un grafo orientato G e un suo albero DFS T, diremo che un vertice v è la* testa *della sua componente fortemente connessa $[v]$ se v è il vertice più in alto di $[v]$ nell'albero DFS.*

Osserviamo che, in base al Lemma 11.7, tutti i vertici di $[v]$ saranno discendenti di v nell'albero DFS. In altri termini, rimuovendo l'arco dell'albero DFS che entra in v, separiamo la componente fortemente connessa $[v]$ dalla radice dell'albero DFS. Il Lemma 11.7 ci suggerisce quindi che per trovare le componenti fortemente connesse del grafo G è sufficiente effettuare una visita in profondità di G, individuare, procedendo dal basso verso l'alto nell'albero DFS, la testa v di ogni componente fortemente connessa e rimuovere l'arco entrante in v.

Per verificare se un vertice v è la testa della sua componente fortemente connessa, esamineremo il sottoalbero dell'albero DFS di radice v: se non ci sono archi fuori da questo sottoalbero, allora v deve essere la testa di $[v]$. Notiamo che gli unici archi che consentono di uscire dal sottoalbero sono gli archi all'indietro o gli archi trasversali a sinistra: infatti, in base alla Proprietà 11.4, gli archi trasversali a destra non possono esistere, mentre gli archi in avanti rimangono all'interno del sottoalbero.

Lemma 11.8 *Siano G un grafo orientato e T un albero DFS prodotto da una visita in profondità a partire da un vertice s di G. Sia v un vertice di G, ed u un discendente di v in T: se durante la visita DFS incontriamo un arco (u, w), dove w è un antenato di v in T, allora v non è la testa della componente fortemente connessa $[v]$.*

Dimostrazione. Se esiste un arco all'indietro (u, w) da un discendente di u di v ad un antenato w di v in T, allora l'arco (u, w) ed il cammino $w \rightsquigarrow v \rightsquigarrow u$ in T garantiscono che w, v ed u fanno tutti parte di un ciclo e quindi della stessa componente fortemente connessa $[v]$: dunque v non potrà essere la testa di $[v]$ perché abbiamo trovato un vertice $w \in [v]$ più in alto di v nell'albero DFS. □

Il caso più complicato da verificare accade quando gli unici archi ad uscire dal sottoalbero di radice v sono archi trasversali a sinistra (ricordiamo che non possono esserci archi trasversali a destra), ovvero archi (u, w) tali che

(i) u è un discendente v, e
(ii) w è stato esaminato prima di v ma non è un antenato di v.

Per eliminare la verifica di questo caso, che appare complicata, progetteremo il nostro algoritmo in modo tale che, non appena la visita DFS finisce di visitare un vertice v che è testa della sua componente fortemente connessa, oltre a cancellare l'arco entrante in v cancella anche tutta la componente fortemente connessa $[v]$. Procedendo in tale modo, cancelleremo anche tutti gli archi trasversali a sinistra entranti in $[v]$. Di conseguenza, i sottoalberi visitati successivamente non potranno contenere vertici da cui escono archi trasversali a sinistra verso i vertici di $[v]$, e quindi gli unici archi che rimarranno da analizzare per verificare se un vertice è la testa della sua componente fortemente connessa saranno gli archi all'indietro. Questo semplificherà notevolmente l'algoritmo. Il seguente lemma dimostra la correttezza di questo approccio.

Lemma 11.9 *Siano G un grafo orientato e T un albero DFS prodotto da una visita in profondità a partire da un vertice s di G, tale che nella visita eliminiamo da T una componente fortemente connessa, con tutti gli archi che le sono incidenti, non appena viene individuata la sua testa. Sia v un vertice di G, ed u un discendente di v in T: se nella visita DFS incontriamo un arco (u, w), dove w è un vertice che è stato visitato prima di v, allora v non è la testa della componente fortemente connessa $[v]$.*

Dimostrazione. Sia (u, w) l'arco incontrato, con u discendente di v, e con w esaminato prima di v dalla visita DFS. Sia inoltre z la testa di $[w]$: osserviamo che per la proprietà della visita DFS, anche z deve essere stato visitato prima di w e quindi di v. Il vertice z, inoltre, deve necessariamente essere un antenato di v: se non lo fosse, avremmo già finito di esaminare tutta la componente fortemente connessa $[w]$, l'algoritmo avrebbe rimosso $[w]$ dall'albero, e quindi non avremmo potuto incontrare l'arco (u, w).

Se z è un antenato di v, allora l'arco (u, w) insieme ai cammini $w \rightsquigarrow z$ (z è la testa di $[w]$), $z \rightsquigarrow v$ (z è un antenato di v in T), $v \rightsquigarrow u$ (v è un antenato di u in T), produce un ciclo contenente z e v: questo implica che $v \in [z]$: essendo z un antenato di v, v non può essere la testa della sua componente fortemente connessa. □

Definiamo il *numero DFS* di un vertice v, che indicheremo con la notazione *numeroDFS*(v), come il numero di vertici incontrati prima di v nella visita in profondità. Notiamo che non è affatto difficile calcolare i numeri DFS durante una visita in profondità: è sufficiente mantenere un contatore, che va incrementato ogni volta che visitamo un vertice del grafo non ancora marcato. La seguente proprietà è immediata:

Proprietà 11.7 *Siano G un grafo orientato e T un albero DFS prodotto da una visita in profondità a partire da un vertice s di G. L'arco (u, v) è un arco all'indietro o un arco trasversale a sinistra relativamente a T se e solo se nella visita numeroDFS(v) < numeroDFS(u).*

Parafrasando i Lemmi 11.8 e 11.9, se nella nostra visita di un sottoalbero DFS di radice v non riusciamo a trovare archi verso vertici che hanno un numero DFS inferiore a quello di v (ovvero visitati prima di v), allora sicuramente v sarà la testa di una componente fortemente connessa. Per costruzione, tali archi saranno proprio gli archi all'indietro che escono al di fuori del sottoalbero DFS di radice v. Vediamo ora come effettuare efficientemente una tale verifica.

Definiamo *fuga*(v) come il più piccolo numero DFS di un vertice raggiungibile con un arco uscente dal sottoalbero di radice v. Se non esiste un tale arco, allora avremo per convenzione fuga(v) = numeroDFS(v). Parafrasando ancora i Lemmi 11.8 e 11.9, possiamo affermare che se fuga(v) = numeroDFS(v), allora il vertice v è la testa della sua componente fortemente connessa.

Osserviamo che il valore fuga(v) può essere facilmente calcolato combinando i valori ottenuti per i figli di v. Più precisamente, per ogni arco (v, w) dell'albero DFS, una volta che è terminata la visita di tutto il sottoalbero di radice w, possiamo aggiornare il valore fuga(v) secondo la relazione:

$$\text{fuga}(v) = \min\{\text{fuga}(v), \text{fuga}(w)\}$$

Se invece (v, w) non è un arco dell'albero DFS, e quindi è un arco all'indietro o in avanti, possiamo aggiornare il valore fuga(v) secondo la relazione:

$$\text{fuga}(v) = \min\{\text{fuga}(v), \text{numeroDFS}(w)\}$$

Abbiamo ora a nostra disposizione tutti gli ingredienti necessari per calcolare i valori numeroDFS(v) e fuga(v) per ogni vertice v durante una visita in profondità. In base a tali valori, se troviamo un vertice v per cui

$$\text{fuga}(v) = \text{numeroDFS}(v)$$

allora possiamo affermare che v è la testa della sua componente fortemente connessa. L'algoritmo di visita in profondità modificato per il calcolo delle componenti fortemente connesse a partire da un vertice v è illustrato nella Figura 11.14. Notiamo che viene utilizzata una pila P per identificare i sottoalberi radicati in un particolare vertice. Ogni volta che visitiamo un vertice, lo inseriamo nella cima della pila P (riga 7). Intuitivamente la pila P memorizza il vertice a cui


```
    procedura visitaFortementeConnesse(vertice v, Pila P)
1.    numeroDFS(v) ← contatore
2.    contatore ← contatore +1
3.    fuga(v) ← numeroDFS(v)
4.    for each ( arco (v, w) in G ) do
5.       if ( w non ancora in T ) then
6.          aggiungi (v, w) a T
7.          P.push(w)
8.          visitaFortementeConnesse(w,P)
9.          fuga(v) ← min {fuga(v), fuga(w)}
10.      else
11.         fuga(v) ← min {fuga(v), numeroDFS(w)}
12.      if ( fuga(v) = numeroDFS(v) ) then
13.         do                          {stampa tutti i vertici della componente [v] }
14.            u ← P.pop()
15.            stampa il vertice u
16.            elimina da T u e tutti gli archi incidenti su u
17.         while (u ≠ v)
```

Figura 11.14 Algoritmo di visita a partire da un vertice v utilizzato nel calcolo delle componenti fortemente connesse di un grafo. La variabile contatore è utilizzata come variabile globale.

tomeremo dopo la chiusura della chiamata ricorsiva: conseguentemente quando abbiamo finito di visitare un vertice v, tutto il suo sottoalbero sarà stato inserito dopo il vertice v nella pila P. Se alla riga 12 scopriamo che v è la testa della sua componente fortemente connessa $[v]$, ed abbiamo già cancellato tutte le altre componenti fortemente connesse localizzate fino a questo punto, allora tutto ciò che affiora nella pila al di sopra di v saranno esattamente i vertici di $[v]$. Questi vengono correttamente mandati in stampa alle righe 13–17 della Figura 11.14.

L'algoritmo `visitaFortementeConnesse`(v) di Figura 11.14 calcola tutte le componenti fortemente connesse raggiungibili a partire dal vertice v. A questo punto ci chiediamo: se partiamo da un vertice di un grafo G, riusciamo ad esplorare tutte le componenti fortemente connesse di G? Purtroppo no, ed infatti in generale non è garantito che esista un vertice capace di raggiungere tutte le componenti fortemente di G. Rimandiamo al Problema 11.11 per una dimostrazione di questo.

Per essere sicuri di visitare tutte le componenti fortemente connesse di G, è sufficiente aggiungere a G un nuovo vertice speciale x, ed archi (x, v) da x a tutti i vertici v di G. Questo vertice speciale x sarà quindi in grado di raggiungere tutto il grafo G: di conseguenza, una visita a partire da x sarà ora in grado di raggiungere tutte le componenti fortemente connesse di G. Questo è illustrato nell'algoritmo di Figura 11.15. L'introduzione del nuovo vertice x, su cui sono incidenti solo archi uscenti, non altera sostanzialmente le componenti fortemente connesse del

grafo originario G. L'unica variazione, infatti, è l'introduzione della componente fortemente connessa $[x]$ che contiene unicamente il vertice x.

```
algoritmo fortementeConnesse(grafo G)
  contatore = 1
  Pila P
  crea un nuovo vertice x con archi (x, v) per tutti i vertici v
  T ← albero contenente un solo vertice x
  visitaFortementeConnesse(x, P)
```

Figura 11.15 Algoritmo per il calcolo delle componenti fortemente connesse di un grafo orientato G.

Teorema 11.2 *Le componenti fortemente connesse di un grafo orientato G con m archi ed n vertici possono essere calcolate in tempo $O(m+n)$ nel caso peggiore.*

Dimostrazione. La correttezza dell'algoritmo deriva dalle considerazioni precedenti. Per stimare il suo tempo di esecuzione, basta osservare che aggiungere un vertice speciale x al grafo G produce un nuovo grafo con $(m+n)$ archi ed $(n+1)$ vertici. Rispetto ad una visita in profondità, l'algoritmo in Figura 11.14 aggiunge al più un lavoro addizionale costante per ogni passo allo scopo di mantenere i valori fuga(v), numeroDFS(v) e la pila P. Il suo tempo totale sarà dunque ancora $O(m+n)$. □

11.6 Problemi

Problema 11.1 Sia $G = (V, E)$ un grafo non orientato connesso, avente n vertici. Dimostrare che G deve necessariamente avere almeno $m \geq n-1$ archi.

Problema 11.2 Sia $G = (V, E)$ un grafo non orientato aciclico, avente n vertici. Dimostrare che G può avere al più $m \leq n-1$ archi.

Problema 11.3 Sia T un albero (connesso) non orientato, avente n vertici. Dimostrare che T ha esattamente $(n-1)$ archi.

Problema 11.4 Descrivere ed analizzare un algoritmo che determina se un grafo non orientato $G = (V, E)$ contiene un ciclo. Il tempo di esecuzione dell'algoritmo dovrebbe essere $O(n)$, e quindi indipendente da m, dove n e m sono rispettivamente il numero di vertici e di archi in G.

Problema 11.5 Sia G un grafo orientato con n vertici. Diciamo che un vertice v di G è un *pozzo* se v ha grado in entrata pari a $(n-1)$ e grado in uscita pari a 0 (ci

sono $(n-1)$ archi entranti in v e nessun arco uscente). Data la matrice di adiacenza di un grafo orientato, descrivere ed analizzare un algoritmo in grado di decidere se il grafo rappresentato da quella matrice ha un vertice pozzo. L'algoritmo dovrebbe essere in grado di esaminare solamente $O(n)$ elementi della matrice, ed avere un tempo di esecuzione totale $O(n)$.

Problema 11.6 Descrivere un algoritmo iterativo di visita in profondità di un grafo $G = (V, E)$, con m archi ed n vertici. Analizzare il suo tempo di esecuzione in funzione di m ed n.

Problema 11.7 Sia $G = (V, E)$ un grafo non orientato, con m archi ed n vertici. Descrivere ed analizzare un algoritmo per il calcolo di tutte le componenti connesse di G. L'algoritmo dovrebbe avere un tempo di esecuzione $O(m+n)$.

Problema 11.8 Sia $G = (V, E)$ un grafo orientato. Dimostrare che x e y sono fortemente connessi se e solo se:

(a) $x = y$, oppure
(b) esiste un ciclo orientato contenente x ed y.

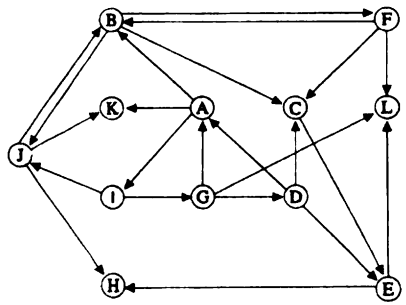


Figura 11.16 Un grafo orientato $G = (V, E)$.

Problema 11.9 Sia $G = (V, E)$ il grafo orientato illustrato in Figura 11.16. Trovare un albero BFS del grafo G generato a partire dal vertice A.

Problema 11.10 Sia $G = (V, E)$ il grafo orientato illustrato in Figura 11.16. Trovare un albero DFS del grafo G generato a partire dal vertice A.

Problema 11.11 Dare un esempio di grafo orientato $G = (V, E)$, tale che non esiste nessun vertice v di G per cui tutte le componenti fortemente connesse di G sono raggiungibili da G.

Problema 11.12 Sia $G = (V, E)$ il grafo orientato illustrato in Figura 11.16. Quante e quali sono le componenti fortemente connesse di G?

Problema 11.13 (**) Un grafo non orientato G con n vertici è uno *scorpione* se ha un vertice di grado uno (il pungiglione), adiacente ad un vertice di grado due (la coda). La coda è a sua volta adiacente ad un vertice di grado $(n-2)$ (il corpo), ed il corpo è adiacente ad altri $(n-3)$ vertici (le zampe). Alcune zampe possono essere connesse ad altre zampe. La Figura 11.17 illustra un esempio di grafo scorpione.

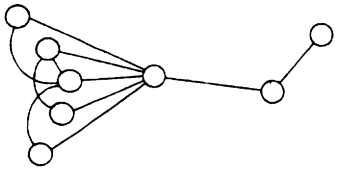


Figura 11.17 Un grafo scorpione.

Data la matrice di adiacenza di un grafo, descrivere ed analizzare un algoritmo in grado di decidere se il grafo rappresentato da quella matrice è uno scorpione. L'algoritmo dovrebbe essere in grado di esaminare solamente $O(n)$ elementi della matrice, ed avere un tempo di esecuzione totale $O(n)$.

11.7 Sommario

In questo capitolo abbiamo introdotto la nozione di grafo, insieme ad una terminologia di base sui grafi. Abbiamo poi passato in rassegna diverse modalità di rappresentazione dei grafi, tra cui le liste di adiacenza e le matrici di adiacenza. Ricordiamo che l'utilizzo di una particolare rappresentazione del grafo può avere un notevole impatto sul tempo di esecuzione di un algoritmo, come abbiamo sperimentato direttamente nella dimostrazione del Teorema 11.1. Abbiamo inoltre definito diverse modalità di visita di un grafo, ed in particolare la visita in ampiezza e la visita in profondità, studiandone in dettaglio alcune proprietà. Tali proprietà, e soprattutto quelle della visita in profondità, sono state molto utili nel progettare algoritmi efficienti per particolari problemi su grafi, come ad esempio il calcolo delle componenti fortemente connesse.

Altri problemi su grafi, come il problema del minimo albero ricoprente, dei cammini minimi, e del massimo flusso saranno discussi in dettaglio rispettivamente nei Capitoli 12, 13 e 14.

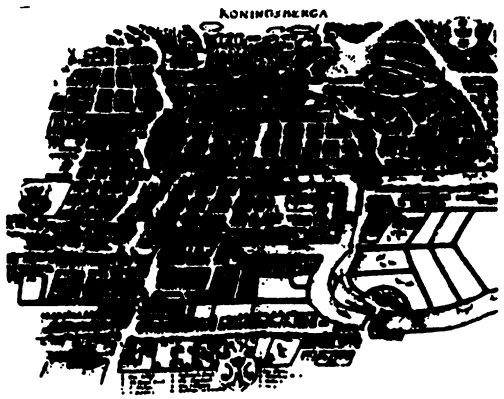


Figura 11.18 Ai tempi di Eulero, la città di Königsberg era attraversata da un fiume con due isole collegate tra loro e alla terraferma tramite sette ponti. Il grafo che astrae tale problema è rappresentato in Figura 11.19.

11.8 Note bibliografiche

Sebbene il termine grafo non sia stato usato fino al 1878, le prime nozioni di teoria dei grafi risalgono al 1735, con un famoso lavoro in cui Eulero risolveva il problema dei ponti di Königsberg. La città di Königsberg era infatti attraversata dal fiume Pregel, che circondava due isole collegate tra di loro e alla terraferma tramite sette ponti, come illustrato dalla Figura 11.18: in tale problema siamo interessati all'esistenza di un ciclo che attraversa ciascuno dei sette ponti una sola volta. Il problema può essere formulato utilizzando il grafo in Figura 11.19, che fu per la prima volta usato nel 1892 da Rouse Ball [9]. Eulero dimostrò che non è possibile tornare esattamente al punto di partenza attraversando ogni ponte una sola volta; esiste una elegante dimostrazione di questa proprietà in termini di grafi, secondo cui il problema ammette soluzione se e solo se ogni vertice del grafo ha grado pari. Tra gli innumerevoli testi di teoria dei grafi che dimostrano questa e molte altre proprietà, ricordiamo [2, 3, 6].

La visita in profondità è stata utilizzata sin dagli anni 1950 e si applica a vari problemi su grafi [11]. La visita in ampiezza è stata scoperta da Moore [8] e da Lee [7]. Il primo algoritmo lineare per calcolare le componenti fortemente connesse di un grafo orientato è stato proposto da Tarjan [11]. Rimandiamo il lettore ai riferimenti [4, 5, 10] per ulteriori approfondimenti relativi agli algoritmi su grafi.

Riferimenti bibliografici

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass. 1974.

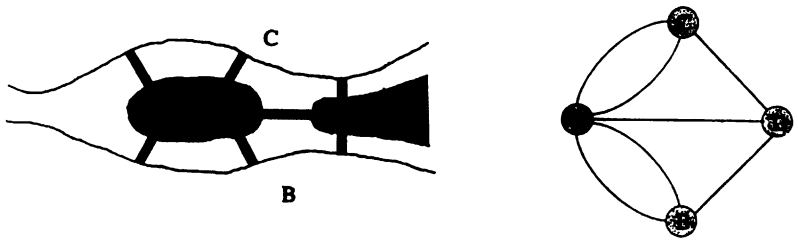


Figura 11.19 Astrazione del problema dei ponti di Königsberg in termini di grafi.

[2] C. Berge, *Graphs*, Elsevier Science Publishers, 1991.
[3] B. Bollobas, *Graph theory: an introductory course*, Springer Verlag, 1979.
[4] S. Even, *Graph Algorithms*, Computer Science Press, 1979.
[5] J. Gross and J. Yellen Eds., *Handbook of Graph Theory*, CRC Press, 2004.
[6] F. Harary, *Graph theory*, Addison Wesley, 1969.
[7] C. Y. Lee, "An algorithm for path connection and its application", *IRE Transactions on Electronic Computers*, EC-10 (1961), 346–365.
[8] E. F. Moore, "The shortest path through a maze", *Procs. Int. Symp. on the Theory of Switching*, 1959, 285–292.
[9] W. W. Rouse Ball, *Mathematical recreations and problems of past and present times*, Macmillan, London, 1892.
[10] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF, Regional Conference Series in Applied Mathematics, SIAM, 1983.
[11] R. E. Tarjan, "Depth first search and linear graph algorithms", *SIAM Journal on Computing*, 1 (1972), 146–160.

12

Minimo albero ricoprente

C'erano dei terribili semi sul pianeta del piccolo principe: erano i semi dei baobab. Il suolo ne era infestato. Ora, un baobab, se si arriva troppo tardi, non si riesce più a sbarazzarsene. Ingombra tutto il pianeta. Lo trapassa con le sue radici. E se il pianeta è troppo piccolo e i baobab troppo numerosi, lo fanno scoppiare.

(Antoine Marie Roger de Saint-Exupéry – Il piccolo principe)

Sia dato un grafo non orientato e connesso $G = (V, E)$, con m archi ed n vertici. Ricordiamo dal Capitolo 11 (si veda ad esempio il Problema 11.1) che se il grafo è connesso deve essere necessariamente $m \geq n - 1$. Introduciamo ora la nozione di *albero ricoprente*.

Definizione 12.1 (Albero ricoprente) *Dato un grafo $G = (V, E)$ non orientato e connesso, un* albero ricoprente *di G è un sottografo $T \subseteq G$ tale che*

(i) T è un albero, e
(ii) T contiene tutti i vertici di G.

Osserviamo che un grafo può avere molti alberi ricoprenti: ad esempio K_4, il grafo completo su quattro vertici, ha esattamente sedici alberi ricoprenti (si veda il Problema 12.1). Supponiamo ora che ogni arco e di G abbiano associato un costo (o peso) $w(e)$. In modo più formale sia definita una funzione costo $w : E \rightarrow R$. In tal caso, possiamo definire il costo di un albero ricoprente T di G come la somma dei costi dei suoi archi:

$$w(T) = \sum_{e \in T} w(e)$$

Alberi ricoprenti diversi avranno in genere costi diversi. Definiamo quindi il concetto di *minimo albero ricoprente*:

Definizione 12.2 (Minimo albero ricoprente) *Dato un grafo $G = (V, E)$ non orientato, connesso e pesato sugli archi, un* minimo albero ricoprente *di G è un albero ricoprente di G di costo minimo.*

Trovare un minimo albero ricoprente è probabilmente uno tra i problemi di ottimizzazione più studiati nella letteratura: la prima pubblicazione scientifica sul minimo albero ricoprente risale infatti al lontano 1926! [1].

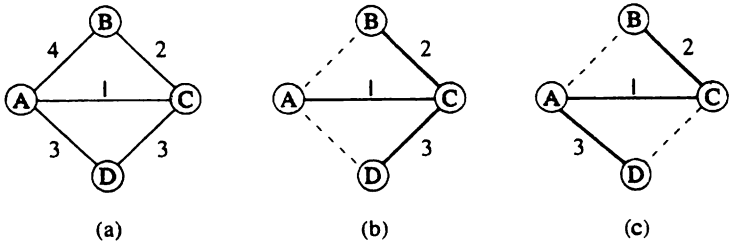


Figura 12.1 Esempio di grafo (a) e dei suoi minimi alberi ricoprenti (b) e (c).

In generale la soluzione non è unica: uno stesso grafo potrebbe ammettere più di un minimo albero ricoprente, come illustrato nella Figura 12.1. Deleghiamo al Problema 12.2 il compito di dimostrare che, se tutti i costi degli archi sono distinti, allora il minimo albero ricoprente è unico.

Le definizioni viste finora possono essere estese al caso in cui il grafo G non sia connesso: si parlerà in tal caso di *minima foresta ricoprente*. Nel resto di questo capitolo, assumeremo che il grafo sia connesso e quindi parleremo soltanto di minimi alberi ricoprenti. Tutte le considerazioni che faremo in questo capitolo possono essere facilmente estese anche al caso di minime foreste ricoprenti.

Esempio 12.1 Il calcolo di un minimo albero ricoprente si presenta frequentemente in vari contesti applicativi. Ad esempio, nel progettare una rete telefonica o di interconnessione, spesso si deve assicurare il collegamento tra località geografiche distinte (ad esempio siti, uffici o agenzie). Per garantire un tale collegamento, è necessario affittare un certo numero di linee telefoniche, i cui costi dipendono spesso dalla coppia di località che deve essere connessa. Tale problema può quindi essere formulato su un grafo non orientato, connesso e pesato sugli archi: i vertici del grafo sono i siti geografici, e gli archi sono i collegamenti tra i siti, con i relativi costi di affitto. L'obiettivo è quello di affittare un insieme di linee che siano in grado di connettere tutte le località richieste al costo totale minimo, ovvero di selezionare un numero di collegamenti (archi nel grafo) che assicurino la connessione tra tutti i siti al costo minimo. La soluzione ricercata deve essere ovviamente un albero ricoprente: infatti, se ci fosse un ciclo nella soluzione, potremmo sempre togliere almeno un collegamento e risparmiare sul costo totale. Tra tutti gli alberi ricoprenti, il costo minimo è assicurato per definizione dalla scelta di un *minimo* albero ricoprente. □

12.1 Proprietà dei minimi alberi ricoprenti

Per risolvere il problema del minimo albero ricoprente, utilizzeremo la tecnica algoritmica *golosa*, oppure *greedy* in inglese, descritta nel Capitolo 10. In base a

tale tecnica, costruiremo un minimo albero ricoprente un arco alla volta, ed effettuando scelte localmente golose. Più in dettaglio, le nostre scelte golose possono consistere nell'includere nella soluzione archi di costo piccolo, scelti opportunamente, oppure nell'escludere dalla soluzione archi di costo elevato, anche questi scelti opportunamente. Formuleremo la tecnica golosa per il problema del minimo albero ricoprente in un paradigma piuttosto generale, in cui sarà possibile inserire gli algoritmi descritti in questo capitolo.

Seguendo la terminologia di Tarjan [14], descriveremo la tecnica golosa come un processo di colorazione. All'inizio tutti gli archi del grafo non saranno colorati. Durante il processo, coloreremo un arco alla volta: quando coloreremo un arco *blu*, lo includeremo nella soluzione; quando coloreremo un arco *rosso*, lo escluderemo dalla soluzione. Per specificare meglio il processo di colorazione, abbiamo bisogno della nozione di taglio in un grafo.

Definizione 12.3 (Taglio) *Dato un grafo non orientato $G = (V, E)$, un taglio in G è una partizione dei vertici V in due insiemi: X e $\overline{X} = V - X$. Un arco $e = (u, v)$ attraversa il taglio $(X, \overline{X})$ se $u \in X$ e $v \in \overline{X}$.*

In base alla Definizione 12.3, un taglio in un grafo è una partizione dell'insieme dei vertici. Talvolta, abusando in qualche modo della notazione, identificheremo il taglio con gli archi che lo attraversano. Enunceremo ora due regole che ci saranno utili nel delineare il paradigma di tecnica golosa per il minimo albero ricoprente.

(REGOLA DEL TAGLIO) *Scegli un taglio in G che non contiene archi blu. Tra tutti gli archi non colorati del taglio, seleziona un arco di costo minimo e coloralo blu.*

L'intuizione alla base della regola del taglio è molto semplice. Consideriamo un taglio che non contiene archi blu. Un minimo albero ricoprente dovrà contenere almeno un arco del taglio, perché altrimenti non connetterebbe tutti i vertici di G, ovvero non ricoprirebbe il grafo G. Quale arco del taglio includeremo nella soluzione? Visto che cerchiamo un *minimo* albero ricoprente, una scelta golosa non può che suggerirci di scegliere l'arco di costo minimo nel taglio.

(REGOLA DEL CICLO) *Scegli un ciclo semplice in G che non contiene archi rossi. Tra tutti gli archi non colorati del ciclo, seleziona un arco di costo massimo e coloralo rosso.*

Anche l'intuizione alla base della regola del ciclo è semplice. Consideriamo un ciclo che non contiene archi rossi. Un minimo albero ricoprente non può contenere tutti gli archi del ciclo, perché altrimenti non sarebbe aciclico e quindi non sarebbe un *albero*. Quale arco del ciclo escluderemo dalla soluzione? Visto che cerchiamo un *minimo* albero ricoprente, una scelta golosa non può che suggerirci di eliminare l'arco di costo massimo nel ciclo.

Il metodo goloso applica una qualsiasi delle due regole ad ogni passo, e termina quando tutti gli archi sono colorati. Dimostriamo ora la correttezza del metodo goloso:

Teorema 12.1 *Sia G un grafo non orientato, connesso, e pesato sugli archi. Ogni passo di colorazione del metodo goloso mantiene la seguente invariante:*

> *"Esiste sempre un minimo albero ricoprente di G che contiene tutti gli archi blu, e che non contiene alcun arco rosso."*

Inoltre, il metodo goloso colora tutti gli archi di G.

Dimostrazione. Dimostriamo innanzitutto che ogni passo di colorazione mantiene l'invariante. Inizialmente, l'invariante è banalmente verificata da qualsiasi albero ricoprente di G (che esiste essendo G connesso), in quanto nessun arco è colorato. Dimostriamo ora che l'invariante viene mantenuta da una generica applicazione della regola del taglio o della regola del ciclo.

Supponiamo che l'invariante sia vera prima di un'applicazione della regola del taglio. Sia $(X, \overline{X})$ il taglio considerato, $e = (u, v)$ l'arco del taglio che è stato colorato blu, e sia T un minimo albero ricoprente di G che verificava l'invariante prima della colorazione di e. Se $e \in T$, allora l'invariante è verificata da T anche dopo la colorazione di e. Se $e \notin T$, allora devone esistere un cammino $\pi(u, v)$ in T tra u e v (gli estremi di e), ed almeno un arco di $\pi(u, v)$, sia esso e', che attraversa il taglio $(X, \overline{X})$ utilizzato per colorare e. Grazie all'invariante, nessun arco di T è rosso. Quindi e' non è rosso. Inoltre, siamo sicuri che il taglio $(X, \overline{X})$ non contiene archi blu, altrimenti non avremmo applicato la regola del taglio a $(X, \overline{X})$. Quindi e' non è neanche blu. Infine, poiché la regola del taglio ha colorato l'arco e, deve essere necessariamente $w(e') \geq w(e)$. In conclusione, e' non è colorato e $w(e') \geq w(e)$. Di conseguenza, l'albero $T' = (T - e') \cup e$ verificherà l'invariante dopo l'applicazione della regola del taglio.

Per dimostrare che un'applicazione della regola del ciclo mantiene l'invariante usiamo un argomento simile. Sia $e = (u, v)$ l'arco del ciclo cbe è stato colorato rosso, e sia T un minimo albero ricoprente di G che verificava l'invariante prima della colorazione di e. Se $e \notin T$, allora l'invariante è verificata da T ancbe dopo la colorazione di e. Supponiamo che $e \in T$. Cancellando e da T lo dividiamo in due sottoalberi, che partizionano l'insieme dei vertici di G in due sottoinsiemi, tali che e ha estremi in entrambi. Consideriamo la regola del ciclo con cui abbiamo colorato e: in tale ciclo deve esistere almeno un altro arco, sia esso e', che ha gli estremi nei due sottoinsiemi di G. Grazie all'invariante, T contiene tutti gli archi blu. Poiché $e' \notin T$, e' non può essere blu. Inoltre, siamo sicuri che il ciclo considerato non contiene archi rossi, altrimenti non vi avremmo applicato la regola del ciclo. Quindi e' non è neanche rosso. Infine, poiché la regola del ciclo ha colorato l'arco e, deve essere necessariamente $w(e') \leq w(e)$. In conclusione, e' non è colorato e $w(e') \leq w(e)$. Di conseguenza, l'albero $T' = (T - e) \cup e'$ verificherà l'invariante dopo l'applicazione della regola del ciclo.

Dimostriamo infine che il metodo goloso termina colorando tutti gli archi di G. Supponiamo per contraddizione che il metodo goloso non riesca più ad applicare né la regola del taglio né la regola del ciclo, e che esista un arco e del grafo che non sia stato ancora colorato. A causa dell'invariante, tutti gli archi blu formano una foresta di alberi, che chiameremo alberi blu. Se gli estremi di e incidono sullo stesso albero blu, possiamo applicare la regola del ciclo sul ciclo indotto da e in quel'albero blu. Se gli estremi di e incidono su due alberi blu distinti, possiamo applicare la regola del taglio sul taglio che separa uno dei due alberi blu dal resto del grafo. In entrambi i casi, riusciamo ad applicare una regola di colorazione, contraddicendo la nostra ipotesi. □

Nei prossimi paragrafi, vedremo come il paradigma del metodo goloso può essere utilizzato per derivare alcuni degli algoritmi classici per il calcolo del minimo albero ricoprente: l'algoritmo di Kruskal, l'algoritmo di Prim, e l'algoritmo di Borůvka.

12.2 Algoritmo di Kruskal

Come visto nella dimostrazione del Teorema 12.1, gli archi blu formano una foresta di alberi, che chiamiamo alberi blu. Durante la sua esecuzione, l'algoritmo di Kruskal mantiene la foresta di alberi blu. All'inizio tale foresta sarà costituita da n alberi disgiunti, ciascuno contenente un singolo vertice. L'algoritmo considera gli archi di $G = (V, E)$ in ordine non decrescente. Per ogni arco, applica il seguente passo:

> (KRUSKAL) *Se l'arco ha entrambi gli estremi nello stesso albero blu, coloralo di rosso. Altrimenti, coloralo di blu.*

Quando tutti gli archi sono siati esaminati, l'algoritmo restituisce l'albero formato dagli archi blu. Lo pseudocodice per l'algoritmo di Kruskal è descritto in Figura 12.2, mentre un esempio di esecuzione dell'algoritmo di Kruskal è illustrato nella Figura 12.3.

```
algoritmo KruskalGenerico(grafo G) → albero
    ordina gli archi di G = (V, E) secondo cosii non decrescenii
    T ← albero vuoto
    for each ( arco (x, y) di G in ordine non decrescenie di cosio ) do
        if ( x e y non sono connessi in T ) then aggiungi l'arco (x, y) a T
    return T
```

Figura 12.2 Versione generica dell'algoritmo di Kruskal.

Dimostriamo innanzitutto la correttezza dell'algoritmo.

Teorema 12.2 *L'algoritmo di Kruskal calcola correttamente un minimo albero ricoprente.*

Dimostrazione. In base al Teorema 12.1 è sufficiente dimostrare che l'algoritmo di Kruskal è un'applicazione del metodo goloso descritto nel Paragrafo 12.1.

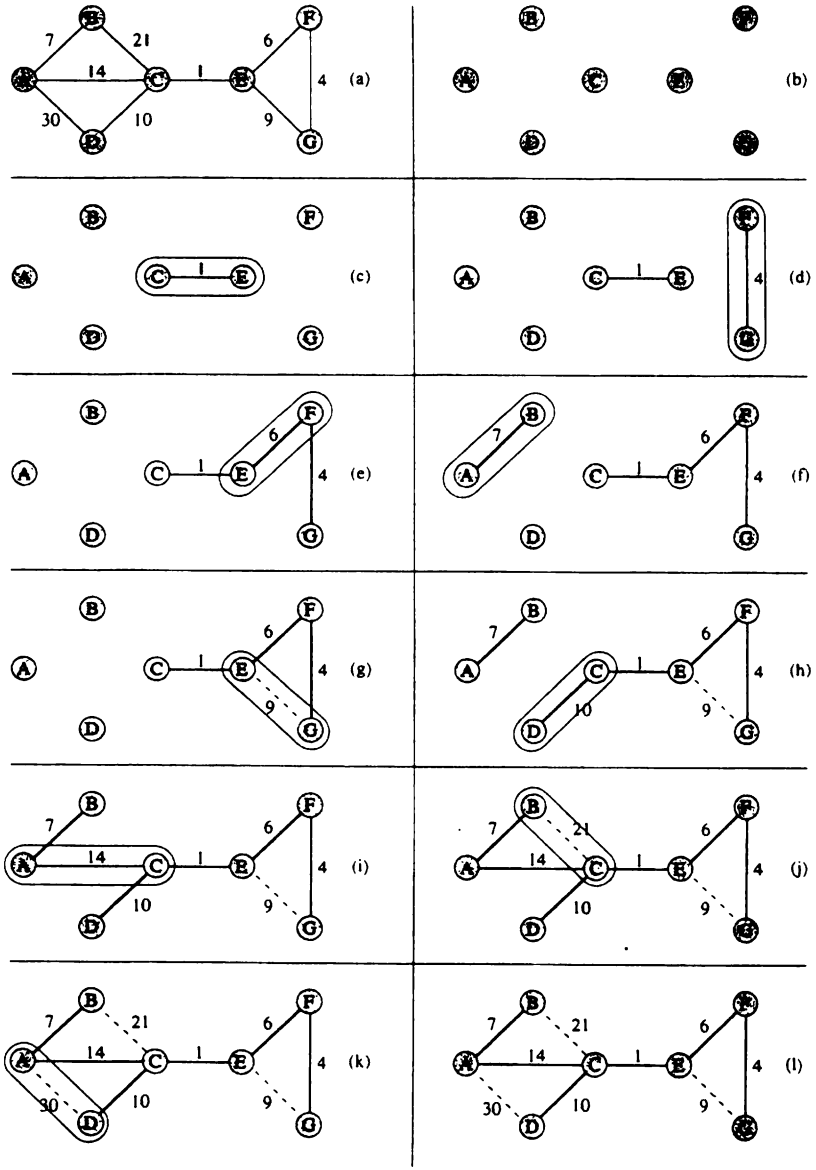


Figura 12.3 Esecuzione dell'algoritmo di Kruskal.

Ricordiamo che gli archi vengono esaminati in ordine non decrescente: quando l'algoritmo di Kruskal esamina l'arco e, e non è ancora stato colorato, mentre l'algoritmo ha già esaminato e colorato tutti e soli gli archi e' di G tali che $w(e') \leq w(e)$.

- Se l'arco e ha entrambi gli estremi nello stesso albero blu, allora e induce un ciclo in tale albero blu: tale ciclo non contiene archi rossi, ed e è l'arco di costo massimo in tale ciclo. L'algoritmo di Kruskal colora correttamente l'arco e di rosso in base alla regola del ciclo.
- Se l'arco e ha gli estremi in due diversi alberi blu, allora e attraversa un taglio che separa tali alberi. Osserviamo che e è il primo arco che attraversa tale taglio ad essere considerato dall'algoritmo: e è quindi l'arco di costo minimo in un taglio che non contiene archi blu, e l'algoritmo di Kruskal colora correttamente e di blu in base alla regola del taglio.

□

In base alla dimostrazione del Teorema 12.2, potremmo parafrasare il generico passo di colorazione di Kruskal nel modo seguente:

(KRUSKAL) *Se l'arco ha entrambi gli estremi nello stesso albero blu, applica la regola del ciclo e coloralo di rosso. Altrimenti, applica la regola del taglio e coloralo di blu.*

Passiamo ora all'analisi del tempo di esecuzione dell'algoritmo di Kruskal. Prima di fare questo, dobbiamo però specificare come verificare se due vertici appartengono allo stesso albero blu. Effettuare questa verifica in modo banale, effettuando cioè una visita degli alberi blu a partire dai due vertici, richiede nel caso peggiore tempo $O(n)$ per ogni arco esaminato, come visto nel Capitolo 3. Essendo m gli archi da esaminare, questo comporterebbe un tempo totale di $O(mn)$.

12.2.1 Implementazione mediante union-find

Vedremo ora come effettuare tale verifica più efficientemente, utilizzando le strutture di dati union-find illustrate nel Capitolo 9: gli alberi blu possono infatti essere rappresentati con insiemi disgiunti, soggetti ad operazioni di `union` e `find`. Una operazione `union` permette di unire due alberi blu (quando viene colorato blu un arco i cui estremi sono in differenti alberi blu), mentre le operazioni `find` consentono di verificare se i due estremi dell'arco considerato appartengono allo stesso albero blu. Un raffinamento dello pseudocodice dell'algoritmo di Kruskal che utilizza strutture di dati union-find è illustrato nella Figura 12.4.

Siamo ora in grado di analizzare il tempo di esecuzione dell'algoritmo di Kruskal.

Teorema 12.3 *Sia $G = (V, E)$ un grafo non orientato, connesso, pesato sugli archi, con m archi ed n vertici. L'algoritmo di Kruskal calcola un minimo albero ricoprente di G in tempo $O(m \log n)$ nel caso peggiore.*

```
algoritmo Kruskal (grafo G) → albero
    UnionFind UF
    T ← albero vuoto
    ordina gli archi di G = (V, E) secondo costi non decrescenti
    for each ( vertice v in G ) do UF.makeSet(v)
    for each ( arco (x,y) in G in ordine non decrescente di costo ) do
        T_x ← UF.find(x)
        T_y ← UF.find(y)
        if ( T_x ≠ T_y ) then
            UF.union(T_x, T_y)
            aggiungi l'arco (x, y) a T
    return T
```

Figura 12.4 Implementazione dell'algoritmo di Kruskal con strutture dati union-find.

Dimostrazione. Scegliendo uno degli algoritmi ottimi di ordinamento presentati nel Capitolo 4, possiamo ordinare gli m archi in tempo $O(m \log m)$. Questo è $O(m \log n)$ visto che m è al più $O(n^2)$. Una volta che gli archi sono ordinati, un minimo albero ricoprente può essere ottenuto con n `makeSet`, $2m$ `find` ed $(n-1)$ `union` su strutture dati union-find, come può desumersi dall'esame del codice in Figura 12.4. Utilizzando i risultati del Capitolo 9, questo richiede $O(m \cdot \alpha(m,n))$ nel caso peggiore, dove $\alpha(m,n)$ è una funzione inversa della funzione di Ackermann. Il tempo totale dell'algoritmo di Kruskal è pertanto $O(m \log n + m \cdot \alpha(m,n)) = O(m \log n)$ nel caso peggiore. □

12.3 Algoritmo di Prim

L'algoritmo di Prim, a differenza dell'algoritmo di Kruskal, mantiene sempre un unico albero blu T. All'inizio tale albero consisterà in un unico vertice, che possiamo scegliere arbitrariamente. Sia s il vertice scelto. Per far "convergere" l'albero blu verso un minimo albero ricoprente, l'algoritmo di Prim applica per $(n-1)$ volte il seguente passo:

(PRIM) *Scegli un arco di costo minimo incidente sull'albero blu T e coloralo di blu.*

Dopo $(n-1)$ passi l'albero blu contiene tutti i vertici. A tal punto l'algoritmo termina e restituisce l'albero blu calcolato. Lo pseudocodice per l'algoritmo di Prim è mostrato in Figura 12.5, mentre un esempio di una sua esecuzione è illustrato nella Figura 12.6. Dimostriamo ora la correttezza dell'algoritmo di Prim.

Teorema 12.4 *L'algoritmo di Prim calcola correttamente un minimo albero ricoprente.*


```
algoritmo PrimGenerico(grafo G) → albero
    T ← albero formato da un solo nodo s
    while ( T ha meno di n nodi ) do
        trova l'arco e di costo minimo incidente su T
        aggiungi l'arco e a T
    return T
```

Figura 12.5 Versione generica dell'algoritmo di Prim.

Dimostrazione. Sfruttando il Teorema 12.1, è sufficiente dimostrare che il generico passo di Prim è semplicemente un'applicazione della regola del taglio. Infatti, sia T l'albero blu prima del generico passo. L'algoritmo ha finora colorato di blu tutti e soli gli archi di T, mentre i rimanenti archi del grafo non sono colorati. Consideriamo il taglio che separa T dal resto del grafo: l'arco scelto dal passo di Prim è esattamente l'arco di costo minimo in tale taglio, e viene correttamente colorato di blu in base alla regola del taglio. □

La dimostrazione del Teorema 12.4 ci presenta l'algoritmo di Prim come una ripetuta applicazione della regola del taglio: l'algoritmo di Prim termina quando non è più possibile applicare tale regola, e restituisce l'albero blu. Tutti gli archi che non sono stati colorati di blu dall'algoritmo, sono implicitamente colorati di rosso.

Passiamo ora all'analisi del suo tempo di esecuzione. Da un'ispezione del codice nella Figura 12.5, appare evidente che l'operazione più costosa è trovare l'arco di costo minimo incidente sull'albero blu. Se eseguiamo una banale ricerca tra tutti gli archi del grafo, possiamo implementare tale ricerca in tempo $O(m)$: dovendo eseguire $O(n)$ passi, in tal modo otteniamo un tempo totale di $O(mn)$. Anche in questo caso però, esattamente come per l'algoritmo di Kruskal, l'utilizzo di strutture di dati opportune può migliorare sensibilmente il suo tempo di esecuzione.

Prima di definire le strutture dati utilizzate, cerchiamo di inviduare quali informazioni sia utile mantenere. Consideriamo l'albero blu ad un certo punto delle iterazioni di Prim. Diciamo che un vertice v è adiacente all'albero blu se v non appartiene all'albero blu ma vi è collegato da almeno un arco. Definiamo la *frontiera dell'albero blu* come l'insieme dei vertici v adiacenti all'albero blu. Per ogni vertice v nella frontiera dell'albero blu, scegliamo un arco di costo minimo tra v e l'albero blu, che chiameremo *arco azzurro* incidente su v. Ad ogni vertice v nella frontiera dell'albero blu associamo quindi il valore $d(v)$, definito come il costo dell'arco azzurro incidente su v. Nel caso in cui v non appartenga alla frontiera dell'albero blu, $d(v) = +\infty$.

Notiamo che gli archi azzurri non sono affatto archi blu, ovvero non appartengono alla soluzione finora costruita. Però sono potenziali candidati a diventare blu: in effetti, il passo di Prim sceglie ogni volta l'arco azzurro di costo minimo e lo colora di blu. Questo aggiungerà un nuovo vertice, diciamo u, all'albe-

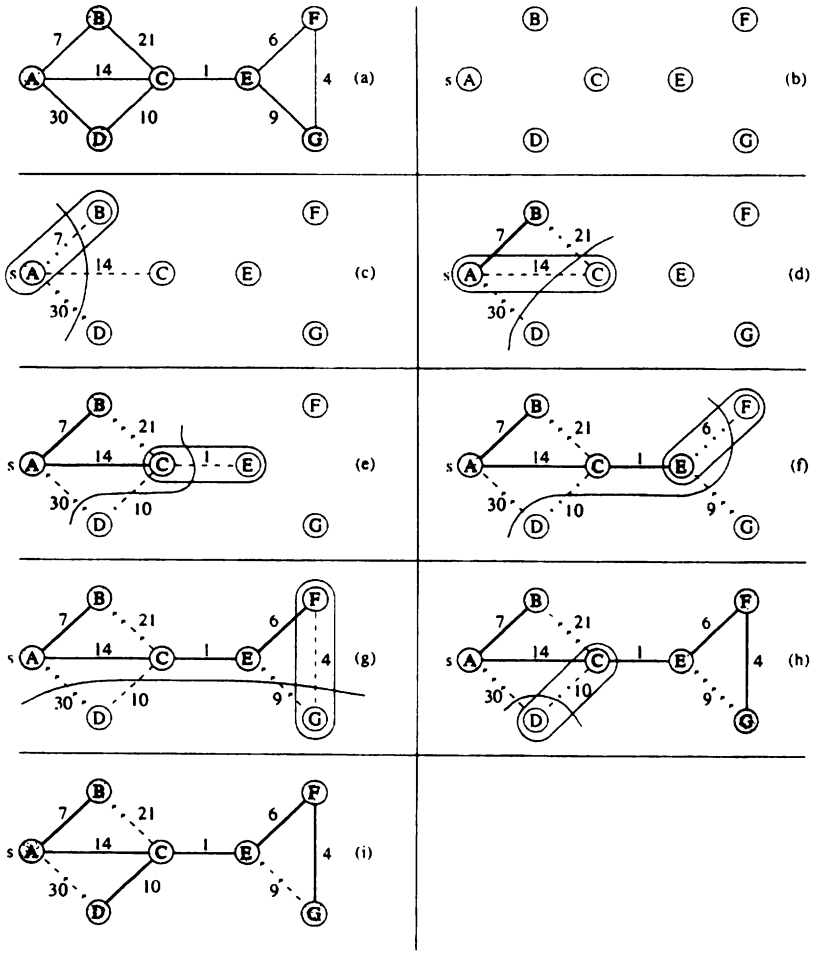


Figura 12.6 Esecuzione dell'algoritmo di Prim.

ro blu. Per aggiornare la frontiera dopo che u è stato inglobato nell'albero blu, esamineremo ogni arco (u, v) incidente su u.

Se v appartiene all'albero blu, allora abbiamo trovato un ciclo che contiene tutti archi blu e l'arco (u, v) che non è colorato. In base alla regola del ciclo, possiamo colorare l'arco (u, v) di rosso. Se invece v non appartiene all'albero blu, possiamo distinguere due casi. Nel primo caso, non ci sono ancora archi azzurri incidenti su v ($d(v) = +\infty$): allora v entrerà a far parte della frontiera di T, colorereremo (u, v) di azzurro, ed inizializzeremo $d(v) = w(u, v)$. Nel secondo caso, esiste già un arco azzurro e incidente su v di costo $d(v)$; se $d(v) > w(u, v)$, allora abbiamo trovato un ciclo che contiene tutti archi blu, il vecchio arco azzurro e di costo $d(v)$ e l'arco (u, v) che non è colorato. Dato che $w(u, v) < d(v)$, in base alla regola del ciclo possiamo colorare l'arco e di rosso; inoltre l'arco (u, v) sarà colorato di azzurro, aggiornando $d(v) = w(u, v)$.

12.3.1 Implementazione mediante code con priorità

Per implementare efficientemente l'algoritmo, è importante gestire efficientemente la frontiera dell'albero blu, che è sottoposta ad operazioni di ricerca del minimo e di aggiornamenti. Per mantenere la frontiera durante la costruzione dell'albero blu, sembra quindi naturale utilizzare una delle code con priorità descritte nel Capitolo 8. In tale coda con priorità manterremo gli archi azzurri incidenti sull'albero blu, sotto forma di coppie del tipo $\langle v, d(v)\rangle$: tale coppia rappresenterà un arco azzurro che connette il vertice v all'albero blu, ed il cui costo è $d(v)$. La priorità utilizzata per ogni coppia nella coda sarà $d(v)$. Lo pseudocodice dell'algoritmo di Prim che utilizza code con priorità è descritto nella Figura 12.7.

Nel corso dell'algoritmo, manterremo un albero T che contiene l'albero blu aumentato degli archi azzurri, e gestiremo la frontiera S mediante una coda con priorità. Ad ogni instante, l'albero blu sarà quindi dato da $T - S$. Il codice alle righe 1–5 ha il compito di inizializzare opportunamente le strutture dati: inizialmente l'albero blu è vuoto, l'algoritmo non ha ancora esaminato alcun arco, e la coda con priorità che realizza la frontiera S per definizione contiene solamente il vertice s, con $d(s) = 0$. Al generico passo, l'algoritmo seleziona il vertice u della frontiera su cui incide l'arco azzurro di costo minimo (riga 7): il vertice u viene eliminato dalla frontiera e quindi inserito nell'albero blu. Le righe 8–16 si preoccupano invece di aggiornare la frontiera dell'albero blu, dopo che il vertice u è stato inglobato nell'albero blu. Alla riga 8 viene considerato ogni arco $f = (u, v)$ incidente sul vertice u. Se v non appartiene alla frontiera dell'albero blu ($d(v) = +\infty$), sarà inserito nella frontiera S con priorità $d(v) = w(u, v)$ (righe 9–12). Se invece v fa già parte della frontiera dell'albero blu, confronteremo il costo $d(v)$ del vecchio arco azzurro incidente su v con il costo del nuovo arco (u, v) che stiamo esaminando: se $w(u, v) < d(v)$, allora è (u, v) ad essere il nuovo arco azzurro incidente su d, e quindi aggiorneremo il corrispondente valore nella coda di priorità (righe 13–16).

Teorema 12.5 *Sia $G = (V, E)$ un grafo non orientato, connesso, pesato sugli*

```
algoritmo Prim (grafo G) → albero
  for each ( vertice v in G ) do d(v) ← +∞
  T ← albero formato da un solo nodo s
  CodaPriorita S
  d(s) ← 0
  S.insert(s,0)
  while ( not S.isEmpty() ) do
    u ← S.deleteMin()
    for each ( arco (u,v) in G ) do
      if (d(v) = +∞) then
        S.insert(v, w(u,v))
        d(v) ← w(u,v)
        rendi u padre di v in T
      else if (w(u,v) < d(v)) then
        S.decreaseKey(v, d(v) − w(u,v))
        d(v) ← w(u,v)
        rendi u nuovo padre di v in T
  return T
```

Figura 12.7 Algoritmo di Prim implementato con coda con priorità secondo lo schema di visita generica illustrato nella Figura 11.5 del Capitolo 11.

archi, con m archi ed n vertici. L'algoritmo di Prim calcola un minimo albero ricoprente di G in tempo $O(m + n \log n)$ nel caso peggiore.

Dimostrazione. Il tempo di esecuzione dell'algoritmo di Prim è dominato dalle operazioni sulla coda con priorità S, e quindi dipende dal tipo di struttura dati scelta per implementarla. In particolare, come possiamo desumere dalla Figura 12.7, su tale coda vengono eseguite n operazioni `deleteMin`, n operazioni `insert`, ed al più $(m - n)$ operazioni `decreaseKey`.

- Se utilizziamo come coda con priorità i d-heap presentati nel Paragrafo 8.1 del Capitolo 8, otteniamo per queste operazioni un tempo di esecuzione pari a $O(nd \log_d n + m \log_d n)$. Per $d = 2$, riotteniamo asintoticamente lo stesso tempo di esecuzione dell'algoritmo di Kruskal, ovvero $O(m \log n)$.
- Se scegliamo per d un valore che bilancia i due termini, come ad esempio $d = \lceil 2 + m/n \rceil$, otteniamo un tempo di esecuzione $O(m \log_{(2+m/n)} n)$. Per $m = \Omega(n^{1+k})$, con $0 < k \leq 1$, abbiamo che $O(m \log_{(2+m/n)} n) = O(m/k)$, e quindi l'algoritmo di Prim con i d-heap è asintoticamente tanto più veloce dell'algoritmo di Kruskal quanto più il grafo di partenza è denso.
- Infine, se utilizziamo come coda con priorità gli heap di Fibonacci presentati nel Capitolo 8, otteniamo per l'algoritmo di Prim un tempo totale di $O(m+n \log n)$ nel caso peggiore.

□

12.4 Algoritmo di Borůvka

L'algoritmo di Borůvka mantiene una foresta di alberi blu. All'inizio tale foresta avrà n alberi, ciascuno contenente un unico vertice. Viene applicato il passo seguente finché non rimane un unico albero blu:

(BORŮVKA) *Per ogni albero blu T, scegli un arco di costo minimo incidente su T. Colora di blu tutti gli archi scelti.*

L'implementazione di questo passo richiede alcune precisazioni. Innanzitutto, dato che ogni arco ha due estremi, uno stesso arco potrebbe essere selezionato due volte nello stesso passo. Questo in realtà non è un problema, dato che basterà colorare tale arco una sola volta. Un problema più serio potrebbe presentarsi invece nel caso in cui i costi degli archi del grafo non siano tutti distinti: in tal caso, infatti, l'arco di costo minimo tra due alberi blu potrebbe non essere necessariamente unico. Per fissare le idee, supponiamo di avere ad un certo punto dell'esecuzione due alberi blu, T_1 e T_2, e due archi e_1 ed e_2, $e_1 \neq e_2$, con $w(e_1) = w(e_2)$, incidenti sia su T_1 che su T_2, che siano entrambi di costo minimo. Osserviamo che selezionare e_1 per T_1 ed e_2 per T_2 introdurrebbe un ciclo nella soluzione.

La correttezza dell'algoritmo di Borůvka, implementato secondo le precedenti specifiche, può essere garantita soltanto nel caso in cui tutti i pesi del grafo siano distinti. Nel caso in cui questa ipotesi non sia soddisfatta, dobbiamo riuscire a garantire che non si creino conflitti nelle scelte simultanee degli archi incidenti sugli alberi blu. Per garantire questa proprietà, potremmo perturbare lievemente i costi degli archi in modo da garantirne l'unicità senza compromettere la soluzione. Nel resto di questo paragrafo non entreremo in questi dettagli ed assumeremo invece che tutti i costi degli archi siano distinti. Un esempio di esecuzione dell'algoritmo di Borůvka è illustrato nella Figura 12.8.

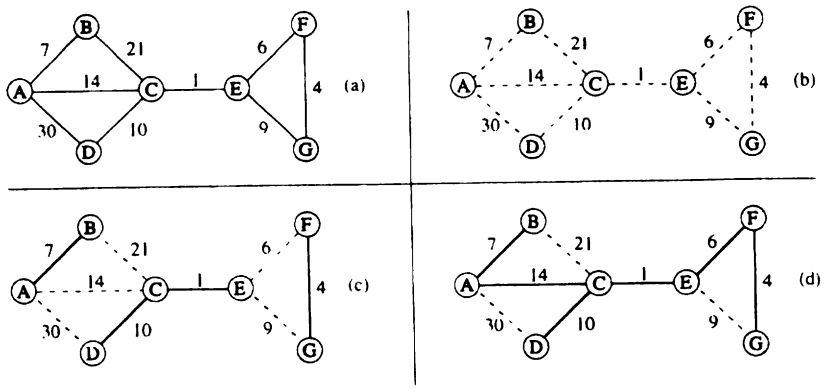


Figura 12.8 Esecuzione dell'algoritmo di Borůvka.

Ci sono vari modi di implementare l'algoritmo di Borůvka: di seguito descriveremo un'implementazione basata su semplici strutture di dati, come le code del

Capitolo 3. Partiamo da una collezione di n vertici distinti, e li inseriamo in una struttura dati coda Q tramite operazioni enqueue. Estraiamo quindi il vertice v in testa a Q tramite un'operazione dequeue nella coda: selezioniamo l'arco e di costo minimo incidente su v nel grafo G, e lo coloriamo di blu. Continuiamo in questo processo fino ad esaminare tutti i vertici nella coda Q. Osserviamo che, se tutti i costi degli archi sono distinti, non è possibile creare cicli contenenti archi blu. Alla fine, quando la coda Q è vuota, contraiamo tutti gli archi del grafo che sono stati colorati blu. Definiamo questo processo come una *fase* dell'algoritmo. Notiamo che ogni super-vertice prodotto dalle contrazioni effettuate alla fine di una fase corrisponde ad un albero blu: le contrazioni essenzialmente hanno il compito di fondere alcuni degli alberi blu presenti all'inizio della fase. I super-vertici prodotti alla fine di una fase vengono utilizzati come vertici di partenza nella fase successiva, e continueremo questo processo fino a che non rimaniamo con un unico super-vertice.

Teorema 12.6 *L'algoritmo di Borůvka può essere implementato in modo da richiedere tempo $O(m \log n)$ nel caso peggiore.*

Dimostrazione. Ognuna delle fasi dell'algoritmo può essere implementata in tempo $O(m)$. Infatti, l'arco di costo minimo incidente in ogni vertice può essere selezionato in tempo totale lineare nel numero di archi. Per effettuare le contrazioni, dobbiamo ricostruire per ogni vertice una rappresentazione con liste di adiacenza nel nuovo grafo contratto (che ha un unico super-vertice per ogni componente identificata dagli archi che sono stati colorati di blu nella fase). Anche questo può essere effettuato in tempo $O(m)$: archi del grafo originario che sono all'interno di una stesso albero blu vengono scartati, mentre archi che sono tra due alberi blu distinti vengono mantenuti nel nuovo grafo contratto.

Per stimare il numero totale di fasi necessarie all'algoritmo, osserviamo che ogni super-vertice prodotto nella contrazione è il risultato della fusione di almeno due vertici presenti all'inizio della fase. Di conseguenza, alla fine della fase i, per ogni $i > 0$, ogni super-vertice corrisponderà ad almeno 2^i vertici del grafo originario: il numero totale delle fasi dell'algoritmo di Borůvka sarà quindi al più $\log_2 n$.

Dato che ci sono al più $O(\log n)$ fasi, ed ogni fase può essere implementata in tempo $O(m)$ nel caso peggiore, il tempo di esecuzione totale di questa implementazione dell'algoritmo di Borůvka è $O(m \log n)$ nel caso peggiore. □

12.5 Problemi

Problema 12.1 Considerare K_4, il grafo completo con 4 vertici e sei archi (ovvero tutti i possibili archi tra coppie di vertici). Disegnare i sedici diversi alberi ricoprenti di K_4.

Problema 12.2 Sia dato un grafo non orientato, connesso e pesato sugli archi. Dimostrare che se i pesi degli archi sono tutti distinti, allora esiste un unico minimo albero ricoprente.
Suggerimento: Studiare con attenzione la dimostrazione del Teorema 12.1.

Problema 12.3 Sia dato un grafo G non orientato, connesso e pesato sugli archi. Descrivere ed analizzare un algoritmo efficiente per calcolare un *massimo* albero ricoprente di G, ovvero un albero ricoprente di costo massimo

Problema 12.4 Sia $G = (V, E)$ un grafo non orientato pesato sugli archi, con m archi ed n vertici. Risolvere il seguente problema:

Sia T un albero ricoprente di G, con radice r. Per ogni vertice v, definiamo $S(v)$ come l'insieme degli archi di $G - T$ che hanno come estremi un vertice che è discendente di v ed un vertice che non è discendente di v. Sia inoltre `fugaMinima`(v) l'arco di costo minimo in $S(v)$ se $S(v) \neq \emptyset$ oppure or `null` se $S(v) = \emptyset$. Notiamo che come caso particolare un vertice è un discendente di se stesso. Descrivere ed analizzare un algoritmo con tempo di esecuzione $O(m \log n)$ che, dato T, sia in grado di calcolare `fugaMinima`(v) per tutti i vertici v in G.
Suggerimento: Può tornare utile una soluzione al Problema 9.10 del Capitolo 9.

Problema 12.5 Sia $G = (V, E)$ un grafo non orientato sugli archi e sia T un suo minimo albero ricoprente. Dato un arco $e \in T$, la *sensibilità* di e è definita come la quantità più grande che può essere aggiunta al peso di e senza far variare la proprietà che T è un minimo albero ricoprente. Descrivere, analizzare e dimostrare la correttezza di un algoritmo con tempo di esecuzione $O(n)$ che, una volta che sia stato calcolato `fugaMinima`(v) come definito nel Problema 12.4, sia in grado di etichettare ogni arco di T con la sua sensibilità. (Assumere che sia disponibile una soluzione al Problema 12.4 anche se non lo si è risolto esplicitamente.)

Problema 12.6 Considerare il seguente algoritmo, che è un ibrido tra l'algoritmo di Prim e l'algoritmo di Borůvka:

1. Eseguire $O(\log \log n)$ passi dell'algoritmo di Borůvka
2. Contrarre ogni albero blu in un super-vertice.
3. Eseguire l'algoritmo di Prim sul grafo ottenuto dopo la contrazione del passo precedente.

Rispondere alle seguenti domande.

(a) Perché l'algoritmo è corretto?
(b) Qual è il suo tempo di esecuzione?

Problema 12.7 Considerare l'algoritmo `ancoraMinimiAlberiRicoprenti` descritto nello pseudocodice della Figura 12.9. Rispondere alle seguenti domande:

```
algoritmo ancoraMinimiAlberiRicoprenti (grafo G) → albero
    T ← albero vuoto
    for each ( vertice v di G ) do
        V_i ← {v_i}
        E_i ← {(v_i, v) ∈ E(G)}
    while ( esiste almeno un insieme V_i ) do
        scegli arbitrariamente un insieme V_i
        trova l'arco di costo minimo (u, v) in E_i
        senza ledere la generalità sia u ∈ V_i e v ∈ V_j
        if ( i ≠ j ) then
            aggiungi l'arco (u, v) a T
            V_i ← V_i ∪ V_j
            E_i ← E_i ∪ E_j
    return T
```

Figura 12.9 Un ulteriore algoritmo per il calcolo del minimo albero ricoprente.

(a) Dimostrare che l'algoritmo è corretto.
(b) Quale algoritmo ci ricorda?
(c) Completare tutti i dettagli dell'implementazione (ad esempio quali strutture dati sono utilizzate)
(d) Analizzare il tempo di esecuzione dell'implementazione proposta.

12.6 Sommario

In questo capitolo abbiamo analizzato il problema del calcolo di un minimo albero ricoprente in un grafo non orientato. Abbiamo dapprima introdotto un paradigma generale, che abbiamo visto essere un'applicazione della tecnica golosa, illustrata nel Capitolo 10. Nel caso del minimo albero ricoprente, la tecnica golosa può essere descritta mediante un processo di colorazione. All'inizio tutti gli archi del grafo non sono colorati. Durante il processo, coloriamo un arco alla volta: quando coloriamo un arco *blu*, lo includiamo nella soluzione; quando coloriamo un arco *rosso*, lo escludiamo dalla soluzione. Per specificare meglio il processo di colorazione, abbiamo introdotto due regole golose: la regola del taglio e la regola del ciclo.

La regola del taglio può essere descritta nel modo seguente. Consideriamo un taglio che non contiene archi blu: visto che cerchiamo un *minimo* albero ricoprente, una scelta golosa non può che suggerirci di scegliere e di colorare blu l'arco di costo minimo nel taglio. Per descrivere la regola del ciclo, consideriamo un ciclo che non contiene archi rossi. Dato che un albero non può contenere cicli, dobbiamo necessariamente escludere un arco del ciclo. Visto che cerchiamo un *minimo* albero ricoprente, una scelta golosa non può che suggerirci di eliminare l'arco di costo massimo nel ciclo.

Abbiamo quindi utilizzato il metodo goloso, ed in particolare la regola del taglio e la regola del ciclo, per derivare e dimostrare la correttezza dei più noti algoritmi per il calcolo del minimo albero ricoprente: l'algoritmo di Kruskal, l'algoritmo di Prim, e l'algoritmo di Borůvka. Nel caso degli algoritmi di Kruskal e di Prim, abbiamo infine visto come l'utilizzo di opportune strutture di dati, come le strutture dati union-find del Capitolo 9, e le code con priorità del Capitolo 8, siano in grado di migliorare sensibilmente i tempi di esecuzione.

12.7 Note bibliografiche

Il problema del minimo albero ricoprente è probabilmente uno dei problemi "storici" delle discipline informatiche, ma continua ad affascinare molti ricercatori e scienziati fino ai giorni nostri. Tra gli algoritmi più antichi ci sono proprio gli algoritmi che abbiamo studiato in questo capitolo: l'algoritmo di Borůvka [1] del 1926, e l'algoritmo noto come algoritmo di Prim, che in realtà fu proposto per la prima volta nel 1930 da Jamík [10], per poi essere riscoperto da Prim [13] e successivamente da Dijkstra [4]. La rassegna di Graham ed Hell [7] fornisce un'ottima panoramica degli algoritmi noti per questo problema a partire dall'algoritmo di Borůvka [1] fino all'utilizzo degli heap di Fibonacci proposti da Fredman e Tarjan [5, 8], che consente di ottenere i tempi di esecuzione nell'enunciato del Teorema 12.5 del Paragrafo 12.3.

Ci sono algoritmi più efficienti di quelli studiati in questo capitolo. In particolare, Fredman e Willard [6] hanno mostrato che un minimo albero ricoprente può essere calcolato in tempo lineare ($O(m+n)$) nel caso in cui i costi degli archi sono numeri interi. Esistono anche algoritmi randomizzati: Karger, Klein e Tarjan [11] hanno presentato un algoritmo randomizzato lineare ($O(m+n)$): in questo caso non ci sono restrizioni sui costi degli archi, ma la soluzione è corretta soltanto con alta probabilità. Notiamo che entrambi gli algoritmi sono ottimi: dato che $\Omega(m+n)$ è un lower bound banale del problema (il grafo va almeno letto), non è possibile ottenere algoritmi randomizzati più veloci né algoritmi deterministici più efficienti nel caso di costi interi.

Gli algoritmi fin qui esaminati lasciano però ancora aperto il caso generale, in cui possiamo avere grafi con costi qualsiasi, e desideriamo calcolare un minimo albero ricoprente in modo deterministico. Basandosi su una nuova coda con priorità, i *soft heap* [2], Chazelle [3] ha recentemente proposto un algoritmo che ha tempo di esecuzione $O(m\alpha(m,n))$ dove α è una funzione inversa della funzione di Ackermann, come abbiamo visto nel Capitolo 9. Nel 2002, Pettie e Ramachandran [12] hanno presentato un ulteriore algoritmo deterministico per il calcolo del minimo albero ricoprente di un grafo, e hanno dimostrato che il loro algoritmo è ottimo. Può sembrare sorprendente, ma questo risultato non chiude ancora completamente il problema: il tempo di esecuzione dell'algoritmo di Pettie e Ramachandran non è infatti noto. Indicando con $T^*(m,n)$ tale tempo di esecuzione, possiamo solo affermare che $\Omega(m+n) \leq T^*(m,n) \leq O(m\alpha(m,n))$.

Nonostante quasi un secolo di lavoro, le ultime parole sul problema del minimo albero ricoprente non sono ancora state scritte!

Riferimenti bibliografici

[1] O. Borůvka, "O jistèm problému minimaálím", *Moravské Přírodovedĕcké Společnosti* 3 (1926), 37–58.

[2] B. Chazelle, "The soft heap: An approximate priority queue with optimal error rate", *Journal of the Association for Computing Machinery* 47, 6 (2000), 1012–1027.

[3] B. Chazelle, "A minimum spanning tree algorithm with inverse-Ackermann type complexity", *Journal of the Association for Computing Machinery* 47, 6 (2000), 1028–1047.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs", *Numer. Math.* 1 (1957), 269–271.

[5] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *Journal of the Association for Computing Machinery* 34, (1987), 596–615.

[6] M. Fredman, and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths", *Journal of Computer and Systems Sciences* 48, 3 (1994), 533–551.

[7] R. L. Graham, and P. Hell, "On the history of the minimum spanning tree problem", *Ann. Hist. Comput.* 7 (1985), 43–57.

[8] H. Gabow, Z. Galil, R. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs", *Combinatorica* 6 (1986), 109–122.

[9] J. N. Kruskal, "On the shortest spanning tree of a graph and the traveling salesman problem", *Proc. American Mathematical Society* 7 (1956), 48-50.

[10] V. Jarník, "O jistèm problému minimaálím", *Moravské Přírodovedĕcké Společnosti* 6 (1930), 57–63.

[11] D. Karger, P. Klein, R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees", *Journal of the Association for Computing Machinery* 42 (1995), 321-328.

[12] S. Pettie, V. Ramachandran, "An optimal minimum spanning tree algorithm", *Journal of the Association for Computing Machinery* 49, 1 (2002), 16–34.

[13] R. C. Prim, "Shortest connection networks and some generalizations", *Bell Syst. Tech. J.* 36 (1957), 1389-1401.

[14] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF, Regional Conference Series in Applied Mathematics, SIAM, 1983.

13

Cammini minimi

Su caminu curzu imbezzat s'ainu
(Il cammino corto fa invecchiare l'asino)
(Proverbio sardo)

Contrariamente all'asino del nostro proverbio sardo, nella nostra vita quotidiana tendiamo a spostarci minimizzando le distanze che dobbiamo percorrere. Se dovessimo viaggiare in macchina da Bolzano a Reggio Calabria (vedi Figura 13.1), non penseremmo certo di passare per Torino. Allo stesso modo, non sembra in generale una buona idea far passare per Sidney un pacchetto di dati in Internet che deve essere trasmesso da Roma a Parigi (anche se alle volte purtroppo può accadere!). Calcolare cammini minimi è un problema classico molto importante con applicazioni su reti di telecomunicazioni e di trasporto, e appare spesso come sottoproblema di altri problemi.

13.1 Cammini minimi e distanze in un grafo

Dato un grafo orientato $G = (V, E)$, assumiamo che ogni arco (u, v) abbia associato un costo reale $w(u, v)$ che dobbiamo pagare per attraversarlo, descritto da una *funzione costo* $w : E \to \mathbb{R}$. Se il grafo rappresenta ad esempio una rete stradale, $w(u, v)$ potrebbe essere la distanza in chilometri o il tempo medio di percorrenza tra le città u e v. Diremo che due vertici x e y sono *connessi* se esiste un cammino da x a y. Il *costo di un cammino* è definito come la somma dei costi dei suoi archi:

Definizione 13.1 (Costo di un cammino) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Il costo $w(\pi)$ di un cammino $\pi = \langle v_0, v_1, \ldots, v_k \rangle$ in G è dato da:*

$$w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Come caso particolare, un cammino formato da un solo vertice ha costo zero.

Figura 13.1 Cammino con tempo di percorrenza minimo fra Bolzano e Reggio Calabria sulla rete stradale italiana calcolato mediante il programma SignPost™ di ROUTE 66 Geographic Information Systems B.V.

Un *cammino minimo* può essere definito come segue.

Definizione 13.2 (Cammino minimo) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Un* cammino (di costo) minimo *fra una coppia di vertici x e y connessi in G è un cammino π^*_{xy} che ha un costo minore o uguale a quello di ogni altro cammino π_{xy} tra gli stessi vertici, cioè tale che:*

$$w(\pi^*_{xy}) = \min_{\pi_{xy} \subseteq G} w(\pi_{xy}).$$

Come definiremo formalmente nel Paragrafo 13.1.2, il costo di un cammino minimo può essere pensato come la *distanza* tra i suoi estremi. I cammini minimi godono di una interessante proprietà strutturale:

Lemma 13.1 (Sottostruttura ottima) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Allora ogni sottocammino di un cammino minimo in G è esso stesso un cammino minimo in G.*

Dimostrazione. Sia $\pi^*_{xy} = \langle x, \ldots, i, \ldots, j, \ldots, y \rangle$ un cammino minimo fra x e y in G. Supponiamo per assurdo che il sottocammino $\pi_{ij} \subset \pi^*_{xy}$ fra i e j non sia minimo. Allora deve esistere un cammino π'_{ij} con $w(\pi'_{ij}) < w(\pi_{ij})$. Ma quindi, rimpiazzando π_{ij} con π'_{ij} in π^*_{xy}, otterremmo un cammino complessivamente più corto fra x e y, contro l'ipotesi che π^*_{xy} fosse minimo. □

Come vedremo, la proprietà di sottostruttura ottima ci permetterà di usare sia la tecnica golosa che quella di programmazione dinamica (vedi il Capitolo 10) per costruire cammini minimi a partire da loro sottocammini.

Osserviamo che in generale potrebbe esistere più di un cammino minimo tra una stessa coppia di vertici. In questo capitolo, saremo interessati a trovarne uno qualunque. Curiosamente, se nel grafo ci sono archi con costi negativi, tra una coppia di vertici connessi potremmo anche non avere alcun cammino minimo come specificheremo meglio nel paragrafo successivo.

13.1.1 Cammini minimi in grafi con cicli

Supponiamo che i costi degli archi possano essere negativi e che vi siano in particolare dei cicli nel grafo con costo minore di zero. È facile convincerci che, se giriamo ripetutamente lungo un ciclo negativo, otteniamo un cammino sempre più lungo in termini di archi attraversati, ma con costo sempre più piccolo. Al limite, otteniamo un cammino con lunghezza infinita e costo $-\infty$! Se ad esempio due vertici x e y appartengono a un ciclo negativo, non siamo in grado di ottenere dalla Definizione 13.2 alcun cammino minimo finito tra di essi.

Si noti che, anche in presenza di cicli negativi, potremmo essere comunque interessati a trovare un cammino semplice (cioè senza ripetizioni di vertici) con il minimo costo. Sorprendentemente, il semplice fatto di dover impedire a un vertice di apparire più di una volta ha implicazioni notevoli sulla difficoltà del problema, che diventa NP completo (vedi il Capitolo 16). Per questo tipo di problemi non sono infatti noti algoritmi che richiedano tempo polinomiale. La non negatività dei cicli ci permetterà invece di sviluppare algoritmi polinomiali molto veloci, come vedremo in questo capitolo.

Nel resto del capitolo assumeremo quindi che nel grafo non ci siano cicli con costo minore di zero: per ogni coppia di vertici connessi potremo allora assumere che esista almeno un cammino minimo finito. Il seguente lemma ci dice inoltre che, sebbene nel grafo possano esserci cicli, possiamo sempre assumere che i cammini minimi siano semplici, cioè non contengano al loro interno cicli.

Lemma 13.2 *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Se non ci sono cicli negativi, fra ogni coppia di vertici connessi in G esiste sempre un cammino minimo semplice, in cui cioè i vertici sono tutti distinti.*

Dimostrazione. Mostriamo che possiamo sempre trasformare un cammino minimo qualsiasi in un cammino minimo semplice. Sia $\pi^*_{xy} = \langle x, \cdots, v, \cdots, v, \cdots, y \rangle$ un cammino minimo non semplice fra x e y in cui il vertice v è ripetuto, e sia $\pi_{vv} = \langle v, \cdots, v \rangle \in \pi^*_{xy}$ un sottocammino fra due occorrenze consecutive di v in π^*_{xy}. Si noti che π_{vv} forma un ciclo in π^*_{xy}. Poiché non ci sono cicli negativi, $w(\pi_{vv}) \geq 0$. Ma possiamo escludere anche il caso $w(\pi_{vv}) > 0$, poiché altrimenti

rimpiazzando il sottocammino π_{vv} con il solo vertice v in π^*_{xy}, otterremmo un cammino più corto, contraddicendo l'ipotesi che π^*_{xy} sia minimo. Quindi deve essere necessariamente $w(\pi_{vv}) = 0$. Allora rimpiazzando π_{vv} con v in π^*_{xy} come detto sopra otteniamo un cammino con lo stesso costo, ma con un ciclo in meno. Possiamo ripetere il procedimento finché non otteniamo un cammino semplice. □

13.1.2 Distanza fra vertici in un grafo

Come abbiamo già osservato, la nozione di cammino minimo ci permette di introdurre il concetto di *distanza* tra vertici in un grafo.

Definizione 13.3 (Distanza) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Definiamo la distanza d_{xy} tra due vertici x e y in G come il costo di un cammino minimo che li connette, o $+\infty$ se non esiste nessun cammino tra di essi:*

$$d_{xy} = \begin{cases} w(\pi^*_{xy}), & \textit{se esiste un cammino tra } x \textit{ e } y \textit{ in } G \\ +\infty, & \textit{altrimenti} \end{cases}$$

Si noti che, come caso particolare, la distanza di ogni vertice da se stesso è zero, cioè $d_{vv} = 0$ per ogni $v \in V$. È facile dimostrare che le distanze in un grafo soddisfano la *disuguaglianza triangolare*, che è una delle proprietà degli spazi metrici:

Lemma 13.3 (Disuguaglianza triangolare) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Allora per ogni tripla di vertici x, y e z in V vale:*

$$d_{xz} \leq d_{xy} + d_{yz}$$

Dimostrazione. Un cammino minimo da x a z ha un costo d_{xz} minore o uguale a quello di ogni altro cammino fra x e z, e quindi anche della concatenazione del cammino minimo fra x e y con il cammino minimo fra y e z, che ha costo $d_{xy} + d_{yz}$. □

Si noti che, se G è non orientato e i costi degli archi sono strettamente positivi, possiamo facilmente dimostrare che l'insieme dei vertici V e la funzione distanza d_{xy} soddisfano anche le altre proprietà degli spazi metrici (vedi Problema 13.1). La proprietà di disuguaglianza triangolare ci permette di derivare la seguente ulteriore proprietà, nota come *condizione di Bellman*, che come vedremo avrà un ruolo fondamentale per comprendere il funzionamento degli algoritmi per il calcolo di cammini minimi che discuteremo in questo capitolo.

Lemma 13.4 (Condizione di Bellman) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Allora per ogni arco $(u, v) \in E$ e per ogni vertice $s \in V$, le distanze fra vertici soddisfano la seguente disuguaglianza:*

$$d_{su} + w(u, v) \geq d_{sv}.$$


```
algoritmo cammino(grafo G, distanze d, vertice x, vertice y) → cammino
    π ← ⟨y⟩; v ← y
    while (v ≠ x) do
        for each ( arco (u, v) in G ) do
            if (d_xu + w(u, v) = d_xv) then          {applicazione del Lemma 13.5}
                aggiungi u come primo vertice in π
                v ← u
                break
    return π
```

Figura 13.2 Algoritmo per costruire un cammino minimo a partire dalle distanze nel grafo.

Dimostrazione. Basta osservare che $d_{uv} \leq w(u, v)$, e quindi $d_{su} + w(u, v) \geq d_{su} + d_{uv} \geq d_{sv}$. □

13.1.3 Costruire cammini minimi a partire da distanze

Osserviamo che la conoscenza delle distanze in un grafo ci permette di costruire facilmente i cammini minimi. Infatti, è facile dimostrare che un arco (u, v) appartiene a un cammino minimo tra s e v se e solo se la condizione di Bellman vale con l'uguaglianza per quell'arco: cioè $d_{su} + w(u, v) = d_{sv}$ (vedi il Problema 13.2):

Lemma 13.5 (Appartenenza a un cammino minimo) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Un arco $(u, v) \in E$ appartiene a un cammino minimo tra s e v se e solo se u è raggiungibile da s e:*

$$d_{su} + w(u, v) = d_{sv}.$$

L'algoritmo di Figura 13.2 è in grado di costruire un cammino minimo $\pi^*_{xy} = \langle v_0, v_1, \ldots, v_k \rangle$ con $v_0 = x$ e $v_k = y$ a partire dalle sole le distanze da x a ogni altro vertice. L'algoritmo trova l'arco (v_{i-1}, v_i) per $i = k, \ldots, 2, 1$ fra gli archi entranti in v_i usando il Lemma 13.5. Il tempo richiesto è pertanto $O(\delta_{out} \cdot k)$, dove δ_{out} è il massimo grado uscente di un vertice del grafo. Nel caso peggiore questo è $O(n \cdot k)$.

13.1.4 Alberi di cammini minimi

Come abbiamo visto nel Paragrafo 13.1.3, ricostruire un cammino a partire dalle distanze nel grafo potrebbe richiedere tempo pari a n volte la sua lunghezza. In questo paragrafo studiamo come mantenere informazioni sui cammini minimi di un grafo in modo da poterli ricostruire in tempo proporzionale al numero di vertici che contengono.

Una possibile soluzione potrebbe essere quella di mantenere esplicitamente i cammini minimi di interesse sotto forma di liste di vertici in modo da poterli poi riprendere quando servono in tempo proporzionale alla loro lunghezza. Se siamo interessati ad esempio ai cammini minimi che portano da uno stesso vertice s a tutti gli altri vertici del grafo, sembrerebbe che dobbiamo usare spazio $O(n^2)$: infatti avremmo $O(n)$ cammini che potrebbero avere lunghezza $O(n)$.

Si può fare di meglio? In effetti è possibile ridurre lo spazio a $O(n)$ rappresentando i cammini minimi mediante un albero radicato in s, che chiameremo *albero dei cammini minimi*. L'esistenza di un tale albero è una conseguenza della proprietà di sottostruttura ottima dei cammini minimi enunciata nel Lemma 13.1, ed è dimostrata nel seguente lemma.

Lemma 13.6 (Esistenza albero dei cammini minimi) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$ e sia s un vertice sorgente prefissato. Allora esiste un albero T che contiene i vertici di G raggiungibili da s tale che ogni cammino in T è un cammino minimo in G.*

Dimostrazione. Dimostreremo l'enunciato costruttivamente. Partiamo da un albero T che contiene il solo nodo s: ovviamente, ogni cammino in T (il solo nodo s) è un cammino minimo in G. Mostriamo che possiamo sempre estendere T in modo da raggiungere nuovi vertici mantenendone la proprietà desiderata. Sia π^*_{sv} un cammino minimo in G tale che v non è ancora in T. Procedendo all'indietro a partire da v, troviamo il primo vertice u in π^*_{sv} che appartiene già a T (al limite, $u = s$). Il sottocammino π^*_{uv} da u a v in π^*_{sv} non contiene altri nodi già in T oltre a u, quindi attaccando π^*_{uv} al nodo u come nuovo ramo di T, quest'ultimo rimane un albero e v diventa una nuova foglia. Assumendo che ogni cammino in T è un cammino minimo in G prima di attaccare il nuovo ramo, mostriamo che la proprietà si preserva dopo l'operazione. Per il Lemma 13.1, il sottocammino π^*_{su} in π^*_{sv} è minimo, e quindi π^*_{sv} non cambierà costo se rimpiazziamo π^*_{su} con il cammino minimo da s a u in T. Ma allora il cammino da s alla foglia v in T è minimo e nuovamente per il Lemma 13.1 lo è anche ogni suo sottocammino. □

13.1.5 Varianti del problema dei cammini minimi

A seconda dello scenario applicativo, potremmo essere interessati a calcolare le distanze solo tra alcune coppie di vertici del grafo. Nel resto del capitolo, parleremo di:

- Problema dei *cammini minimi fra tutte le coppie*, se saremo interessati a calcolare le distanze tra ogni coppia di vertici nel grafo.
- Problema dei *cammini minimi a sorgente singola*, se saremo interessati a calcolare le distanze solo a partire da un dato vertice chiamato *sorgente*.
- Problema dei *cammini minimi fra una singola coppia*, se saremo interessati a calcolare la distanza fra due vertici prefissati.


```
algoritmo distanzeGenerico(grafo G) → distanze
    inizializza D in modo che D_xy ≥ d_xy per ogni coppia di interesse (x, y)
    while ( esiste (x, y) fra le coppie di interesse tale che D_xy > d_xy ) do
        considera un vertice v e un cammino π_vy
        if (D_xv + w(π_vy) < D_xy) then D_xy ← D_xv + w(π_vy)        {rilassamento}
    return D
```

Figura 13.3 Metodo generico basato sulla tecnica del rilassamento per il calcolo delle distanze fra un certo insieme di coppie di vertici.

Nei paragrafi 13.3, 13.4 e 13.5 vedremo algorimi per il problema dei cammini minimi a sorgente singola, mentre nel Paragrafo 13.6 vedremo come risolvere il problema dei cammini minimi tra tutte le coppie. Sorprendentemente, non c'è ancora alcun algoritmo noto per il problema dei cammini minimi fra una singola coppia che non risolva anche il problema dei cammini minimi a sorgente singola!

13.2 La tecnica del rilassamento

Tutti gli algoritmi che vedremo nel resto del capitolo calcolano distanze per identificare cammini minimi. Essi partono da stime per eccesso $D_{xy} \geq d_{xy}$ delle distanze nel grafo, e le aggiornano progressivamente decrementandole finché esse non diventano esatte, cioè $D_{xy} = d_{xy}$. L'aggiornamento di una stima D_{xy} consiste genericamente nel considerare un vertice v e un cammino π_{vy} e nell'applicare il seguente *passo di rilassamento*:

$$\text{(RILASSAMENTO)} \quad \textbf{if } (D_{xv} + w(\pi_{vy}) < D_{xy}) \textbf{ then } D_{xy} \leftarrow D_{xv} + w(\pi_{vy})$$

Questo passo verifica una violazione locale della proprietà di disuguaglianza triangolare, e aggiorna D_{xy} tentando di eliminarla. I vari algoritmi che vedremo differiscono principalmente per il modo in cui viene scelto il vertice v e il cammino π_{vy}, ma possono essere ricondotti tutti al metodo generico illustrato in Figura 13.3, che ricorda l'algoritmo di estrazione del minimo da un insieme. Come vedremo, π_{vy} potrà essere un singolo arco (v, y), e quindi $w(\pi_{vy}) = w(v, y)$, oppure un intero cammino tale che $w(\pi_{vy}) = D_{vy}$. In ogni istante durante l'esecuzione di questo tipo di algoritmi, D_{xy} è sempre uguale al costo di qualche cammino fra x e y (o eventualmente $+\infty$), e la determinazione delle distanze si basa interamente su confronti fra costi di cammini nel grafo.

L'uso del termine "rilassamento" per indicare un'operazione che avvicina D_{xy} a d_{xy} è storico, anche se non molto intuitivo. La ragione per cui è stato introdotto è la seguente. Se $D_{xv} + w(\pi_{vy}) < D_{xy}$, allora certamente D non soddisfa la disuguaglianza triangolare, e quindi non può essere una distanza. È quindi necessario rendere localmente vera la condizione $D_{xv} + w(\pi_{vy}) \geq D_{xy}$. Dopo

```
algoritmo BellmanFord(grafo G, vertice s) → distanze
1.   inizializza D tale che D_sv = +∞ per v ≠ s, e D_ss = 0
2.   for i = 1 to n do
3.      for each ( (u,v) ∈ E ) do
4.         if (D_su + w(u,v) < D_sv) then D_sv ← D_su + w(u,v)   {rilassamento}
5.   return D
```

Figura 13.4 Algoritmo di Bellman e Ford per il calcolo delle distanze a partire da una sorgente s in un grafo orientato G con n vertici ed m archi.

l'operazione di rilassamento $D_{xy} \leftarrow D_{xv} + w(\pi_{vy})$, la necessità di rendere vero il vincolo cessa, cioè è "rilassata".

13.3 Algoritmo di Bellman e Ford

L'algoritmo che vedremo in questo paragrafo calcola i cammini minimi a partire da un vertice sorgente prefissato s. Per comprendeme il funzionamento, consideriamo un cammino minimo inizialmente ignoto $\pi^*_{sv_k} = \langle s, v_1, \ldots, v_k \rangle$ e supponiamo di volerne trovare il costo d_{sv_k}. In base al Lemma 13.5, possiamo scrivere:

$$d_{sv_k} = d_{sv_{k-1}} + w(v_{k-1}, v_k).$$

Possiamo ricondurre quindi il calcolo di d_{sv_k} a quello di $d_{sv_{k-1}}$!

Usando lo schema generale di Figura 13.3, possiamo inizializzare $D_{ss} = d_{ss} = 0$ e porre $D_{sv} = +\infty$ per ogni $v \neq s$. Ora, se fossimo in grado di effettuare i passi di rilassamento nel seguente ordine:

$$\begin{array}{llllll}
1. & D_{sv_1} & \leftarrow & D_{ss} & + & w(s, v_1) \\
2. & D_{sv_2} & \leftarrow & D_{sv_1} & + & w(v_1, v_2) \\
3. & D_{sv_3} & \leftarrow & D_{sv_2} & + & w(v_2, v_3) \\
\vdots & \vdots & & & & \\
k. & D_{sv_k} & \leftarrow & D_{sv_{k-1}} & + & w(v_{k-1}, v_k)
\end{array}$$

arriveremmo in k passi ad avere $D_{sv_k} = d_{sv_k}$ senza "sbagliare un colpo". Il rilassamento all'i-esima iterazione avviene considerando il costo di un cammino minimo tra s e v_{i-1} calcolato al passo precedente e quello del cammino formato dal solo arco (v_{i-1}, v_i). Questo sembra essere quanto di meglio possiamo sperare di fare. Il punto è che non conosciamo gli archi sul cammino minimo da s a v_k, né tantomeno il loro ordine.

Come fare a "indovinare" gli archi da usare e il loro ordine senza spendere troppo tempo? L'algoritmo di Bellman e Ford si basa sulla seguente semplice idea. Se effettuiamo una prima passata "rilassando" tutti gli archi $(u, v) \in E$ nel seguente modo:

(RILASSAMENTO) **if** $(D_{su} + w(u,v) < D_{sv})$ **then** $D_{sv} \leftarrow D_{su} + w(u,v)$

certamente includeremo anche il rilassamento del passo 1 visto sopra, ottenendo quindi correttamente $D_{sv_1} = d_{sv_1}$. Ripetendo la passata su tutti gli archi un'altra volta, includeremo il rilassamento del passo 2, ottenendo $D_{sv_2} = d_{sv_2}$. Poiché il base al Lemma 13.2 si ha $k < n$, basteranno n passate per arrivare a calcolare correttamente D_{sv_k} per ogni k. Se abbiamo m archi nel grafo, ogni passata richiede tempo $O(m)$ e quindi il tempo totale richiesto per le n passate è $O(mn)$:

Teorema 13.1 *Sia $G = (V, E)$ un grafo orientato con n vertici ed m archi e con funzione costo $w : E \rightarrow \mathbb{R}$. Allora l'algoritmo di Bellman e Ford richiede tempo $O(mn)$ per calcolare tutte le distanze da un vertice s a tutti gli altri vertici nel grafo.*

L'algoritmo di Bellman e Ford è mostrato in Figura 13.4. Si noti che, se un vertice x non è raggiungibile da s, allora alla fine dell'esecuzione dell'algoritmo rimarrà correttamente $D_{sx} = +\infty$.

Osserviamo che l'algoritmo di Bellman e Ford fa tipicamente un grande numero di operazioni di rilassamento del tutto inutili. Se infatti $D_{sv_{i-1}} > d_{sv_{i-1}}$, come può accadere se rilassiamo ad esempio l'arco (v_{i-1}, v_i) prima di (v_{i-2}, v_{i-1}), porre $D_{sv_i} \leftarrow D_{sv_{i-1}} + w(v_{i-1}, v_i)$ non contribuisce in alcun modo ad aumentare la nostra conoscenza dei cammini minimi: se $D_{sv_{i-1}}$ è una stima errata della distanza, allora lo sarà certamente anche D_{sv_i}! L'operazione $D_{sv_i} \leftarrow D_{sv_{i-1}} + w(v_{i-1}, v_i)$ dovrà essere ripetuta più avanti, quando $D_{sv_{i-1}}$ sarà diventato uguale a $d_{sv_{i-1}}$.

Nei paragrafi seguenti vedremo come sfruttare opportuni vincoli sul grafo, come l'aciclicità o la non negatività dei costi degli archi, per ridurre il numero totale di operazioni di rilassamento.

13.4 Algoritmo per grafi diretti aciclici

Come abbiamo visto nel Paragrafo 13.3, effettuare i passi di rilassamento nell'ordine giusto è cruciale per evitare di fare operazioni inutili. In questo paragrafo vedremo che, se il grafo G è aciclico, allora possiamo identificare un ordine per le operazioni di rilassamento in modo da "rilassare" ogni arco esattamente una volta, e non n volte come nell'algoritmo di Bellman e Ford.

13.4.1 Ordinamento topologico

In un grafo aciclico, se esiste un cammino da u a v, allora cettamente non esiste un cammino da v a u. Questo ci permette di definire un ordinamento totale dei vertici tale che u precede v nell'ordinamento se esiste un cammino da u a v. Un ordinamento di questo tipo viene detto *ordinamento topologico*.

Definizione 13.4 (Ordinamento topologico) *Sia $G = (V, E)$ un grafo orientato aciclico con n vertici. Un* ordinamento topologico *dei vertici di G è una funzione $\ell : V \to \{1, \ldots, n\}$ tale che $\ell(u) < \ell(v)$ se esiste un cammino da u a v in G.*

Si noti che, allineando in ordine topologico i vertici di un grafo aciclico, tutti gli archi andranno nello stesso verso, come illustrato in Figura 13.5 (b). Osserviamo inoltre che, se esistono due vertici u e v non connessi, potrà essere sia $\ell(u) < \ell(v)$ che $\ell(v) < \ell(u)$, e quindi in generale l'ordinamento topologico di un grafo aciclico non è unico. Nell'esempio di Figura 13.5, invertendo la posizione dei vertici E ed G si ottiene un altro possibile ordinamento topologico.

Calcolare un ordinamento topologico è molto semplice: basta identificare un vertice u che non ha archi entranti, porre $\ell(u) = 1$ e cancellarlo dal grafo. Ripetendo il ragionamento, troviamo un altro vertice v che non ha entranti, poniamo $\ell(v) = 2$ e lo cancelliamo dal grafo. Continuando in questo modo, se il grafo è aciclico arriviamo a ordinare correttamente tutti i vertici del grafo, che arriva a svuotarsi. Questo semplice algoritmo è illustrato in Figura 13.6, dove viene mostrato come costruire una lista di vertici ord in cui ogni nodo u ha posizione $\ell(u)$: per evitare di distruggere il grafo G in ingresso, l'algoritmo lavora con una copia $\widehat{G}$ di G. Se il grafo ha n vertici e m archi, esso richiede tempo $O(m + n)$ poiché consideriamo ogni vertice e ogni arco una sola volta. Dimostriamo ora che l'algoritmo trova correttamente un ordine topologico.

Lemma 13.7 *Sia $G = (V, E)$ un grafo orientato aciclico. L'algoritmo* `ordinamentoTopologico` *mostrato in Figura 13.6 calcola correttamente un ordinamento topologico di G.*

Dimostrazione. Denotiamo con ord_i e con $\widehat{G}_i$ il contenuto della lista ord e il grafo $\widehat{G}$ all'inizio dell'iterazione i-esima, rispettivamente. Si noti che per ogni i, ord_i e $V(\widehat{G}_i)$ formano una partizione di V. Dimostriamo per induzione su i che non può esserci alcun cammino in G che porta da un vertice in $\widehat{G}_i$ a uno in ord_i. Il passo base per $i = 1$ è banalmente verificato, poiché $ord_1 = \emptyset$ e $\widehat{G}_1 = G$. Assumiamo per ipotesi induttiva che non ci sia nessun cammino da un vertice in $\widehat{G}_k$ a uno in ord_k per $k > 1$. Al passo k-esimo, scegliamo un vertice u senza

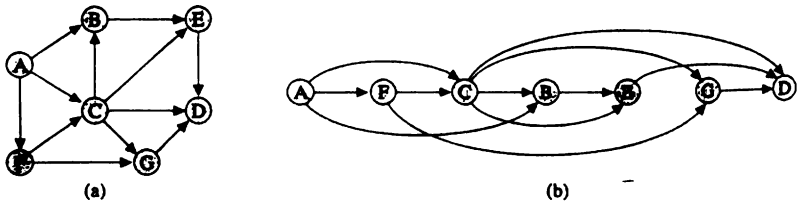


Figura 13.5 (a) Esempio di grafo aciclico; (b) Ordinamento topologico del grafo (a).

algoritmo `ordinamentoTopologico`($grafo\ G$) $\to$ *lista*
1. $\widehat{G} \leftarrow G$
2. $ord \leftarrow$ lista vuota di vertici
3. **while** (esiste un vertice u senza archi entranti in $\widehat{G}$) **do**
4. appendi u come ultimo elemento di ord
5. rimuovi da $\widehat{G}$ il vertice u e tutti i suoi archi uscenti
6. **if** ($\widehat{G}$ non è diventato vuoto) **then errore** il grafo G non è aciclico
7. **return** ord

Figura 13.6 Algoritmo per il calcolo di un ordinamento topologico ℓ del grafo orientato aciclico G. L'algorimo restituisce una lista di vertici ord in cui ogni nodo u ha posizione $\ell(u)$.

entranti in $\widehat{G}_k$, e quindi non ci sono cammini in G che arrivano in u usando solo vertici di $\widehat{G}_k$. L'unico modo per arrivare da un nodo in $\widehat{G}_k$ a u sarebbe tramite un vertice in ord_k, ma questo è impossibile per la nostra ipotesi induttiva.

Per completare la dimostrazione, si osservi che in un grafo aciclico c'è sempre almeno un vertice senza archi entranti (vedi Problema 13.3), e che togliendo vertici a un grafo aciclico esso continua a rimanere aciclico. Quindi, a meno che non ci sia un ciclo, è sempre possibile trovare un vertice senza archi entranti e l'algoritmo termina quando tutti i vertici sono stati considerati. □

13.4.2 Rilassamento in ordine topologico

In Figura 13.7 mostriamo una variante dell'algoritmo di Bellman e Ford per grafi aciclici che effettua le operazioni di rilassamento secondo un ordinamento topologico. Vengono prima rilassati tutti gli archi (u, v) con $\ell(u) = 1$, poi quelli con $\ell(u) = 2$, ecc. Se consideriamo un cammino minimo $\pi^*_{sv_k} = \langle s, v_1, \ldots, v_k \rangle$, si ha $\ell(s) < \ell(v_1) < \cdots < \ell(v_k)$ e quindi i rilassamenti che permettono di determinarne il costo avvengono esattamente nell'ordine discusso nel Paragrafo 13.3:

$$
\begin{array}{llllll}
1. & D_{sv_1} & \leftarrow & D_{ss} & + & w(s, v_1) \\
2. & D_{sv_2} & \leftarrow & D_{sv_1} & + & w(v_1, v_2) \\
3. & D_{sv_3} & \leftarrow & D_{sv_2} & + & w(v_2, v_3) \\
\vdots & \vdots & & & & \\
k. & D_{sv_k} & \leftarrow & D_{sv_{k-1}} & + & w(v_{k-1}, v_k)
\end{array}
$$

Alla fine dell'esecuzione si avrà pertanto correttamente $D_{sv_k} = d_{sv_k}$. Ogni arco viene rilassato una volta soltanto, e quindi l'algoritmo richiede tempo $O(m + n)$.

Teorema 13.2 *Sia $G = (V, E)$ un grafo orientato aciclico con n vertici ed m archi e con funzione costo $w : E \to \mathbb{R}$. Allora l'algoritmo di Figura 13.7 richiede*

```
algoritmo distanzeAciclico(grafo G, vertice s) → distanze
    inizializza D tale che D_sv = +∞ per v ≠ s, e D_ss = 0
    ord ← ordinamentoTopologico(G)
    for i = 1 to n do
        sia u l'i-esimo vertice nell'ordinamento topologico ord
        for each ( (u,v) ∈ E ) do
            if (D_su + w(u,v) < D_sv) then D_sv ← D_su + w(u,v)    {rilassamento}
    return D
```

Figura 13.7 Algoritmo per il calcolo delle distanze a partire da una sorgente s in un grafo orientato aciclico G con n vertici ed m archi.

tempo $O(m+n)$ per calcolare tutte le distanze da un vertice s a tutti gli altri vertici nel grafo.

Ricordando il tempo $O(mn)$ richiesto dall'algoritmo di Bellman e Ford, si noti l'enorme miglioramento prestazionale ottenibile sfruttando l'aciclicità!

13.5 Algoritmo di Dijkstra

Come abbiamo visto nel Paragrafo 13.4, effettuando le operazioni di rilassamento in ordine topologico riusciamo a considerare ogni arco di un grafo aciclico esattamente una volta, e non n volte come nell'algoritmo di Bellman e Ford. In questo paragrafo vedremo che nel caso in cui il grafo può contenere cicli, ma gli archi hanno costi maggiori o uguali a zero, riusciremo ancora a ottenere l'ordine giusto in cui effettuare le operazioni di rilassamento in modo da evitare di rilassare più di una volta lo stesso arco.

Alla fine degli anni '50, E. W. Dijkstra (pronuncia: "Daikstra") [9] e indipendentemente altri ricercatori come G. B. Dantzig, P. D. Whiting e J. A. Hillier [37], scoprirono una proprietà interessante dei cammini minimi, enunciata nel seguente lemma.

Lemma 13.8 (Dijkstra) *Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}^+ \cup \{0\}$ non negativa. Se T è un albero di cammini minimi radicato in s, ma che non include tutti i vertici raggiungibili da s in G, allora l'arco (u, v) con $u \in V(T)$ e $v \notin V(T)$ che minimizza la quantità $d_{su} + w(u, v)$ appartiene a un cammino minimo da s a v.*

Dimostrazione. Supponiamo per assurdo che (u, v) non appartenga a un cammino minimo da s a v, e che quindi $d_{su} + w(u, v) > d_{sv}$. Allora ci deve essere un cammino minimo π_{sv} con costo minore di $d_{su} + w(u, v)$ che porta in v senza passare per (u, v). Poiché $s \in T$ e $v \notin T$, allora ci deve essere un arco $(x, y) \in \pi_{sv}$ con $x \in T$ e $y \notin T$ (si veda Figura 13.8). Sia π_{sy} il sottocammino di π_{sv} da

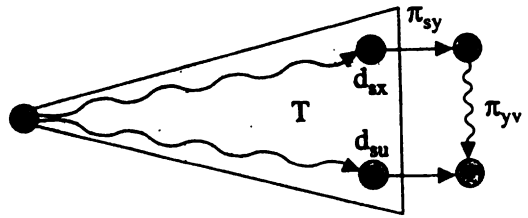


Figura 13.8 Dimostrazione del Lemma 13.8.

s a y e π_{yv} il sottocammino di π_{sv} da y a v. Per il Lemma 13.1 (sottostruttura ottima dei cammini minimi) $w(\pi_{sy})$ è un cammino minimo, e dunque per il Lemma 13.5 (appartenenza di un arco a un cammino minimo) $w(\pi_{sy}) = d_{sx} + w(x, y)$. Pertanto

$$w(\pi_{sv}) = w(\pi_{sy}) + w(\pi_{yv}) = d_{sx} + w(x, y) + w(\pi_{yv}).$$

Poiché (u, v) minimizza $d_{su} + w(u, v)$, allora deve essere

$$d_{sx} + w(x, y) > d_{su} + w(u, v),$$

e allora

$$w(\pi_{sv}) > d_{su} + w(u, v) + w(\pi_{yv}).$$

Infine, poiché i costi degli archi sono non negativi, allora $w(\pi_{yv}) \geq 0$. Questo ci porta alla contraddizione:

$$w(\pi_{sv}) > d_{su} + w(u, v) > d_{sv}$$

□

Il Lemma 13.8 ci fornisce un metodo semplicissimo per costruire un albero di cammini minimi T che include tutti i vertici raggiungibili da s. All'inizio tale albero conterrà solo il vertice s. Assumiamo per semplicità che ogni vertice sia raggiungibile da s. Per far "convergere" l'albero T verso un albero dei cammini minimi da s a ogni altro vertice, l'algoritmo di Dijkstra applica per $(n-1)$ volte il seguente passo:

(DIJKSTRA) *Scegli un arco (u, v) con $u \in V(T)$ e $v \notin V(T)$ che minimizza $D_{su} + w(u, v)$, effettua il passo di rilassamento $D_{sv} \leftarrow D_{su} + w(u, v)$, e aggiungilo a T.*

Dopo $(n-1)$ passi l'albero T contiene tutti i vertici. A tal punto l'algoritmo termina e restituisce l'albero calcolato. Lo pseudocodice per l'algoritmo di Dijkstra è mostrato in Figura 13.9, mentre un esempio di una sua esecuzione è illustrato nella Figura 13.10. Si noti la perfetta somiglianza con l'algoritmo di Prim per il calcolo di un minimo albero ricoprente descritto nella Figura 12.5 del Capitolo 12. Come l'algoritmo di Prim, anche l'algoritmo di Dijkstra si basa sulla

tecnica golosa vista nel Capitolo 10: notiamo infatti che la selezione dell'arco (u, v) con $D_{su} + w(u, v)$ minimo fra quelli incidenti sull'albero corrente T è una scelta golosa, e la sua aggiunta all'albero T effettuato alla riga 6 di Figura 13.9 è permanente, mentre gli archi scartati non verranno più considerati in futuro.

L'operazione che sembra cruciale per implementare efficientemente l'algoritmo di Dijkstra è proprio quella di selezione dell'arco (u, v) con $D_{su} + w(u, v)$ minimo fra quelli incidenti sull'albero corrente T. Se eseguiamo una banale ricerca tra tutti gli archi del grafo, possiamo implementare tale ricerca in tempo $O(m)$: dovendo eseguire $O(n)$ passi, in tal modo otteniamo lo stesso tempo totale $O(mn)$ dell'algoritmo di Bellman e Ford.

13.5.1 Implementazione mediante code con priorità

Esattamente come per gli algoritmi di Kruskal e Prim visti nel Capitolo 12, anche per l'algoritmo di Dijkstra l'utilizzo di strutture di dati opportune può migliorare sensibilmente il suo tempo di esecuzione. Infatti, mantenendo gli archi (u, v) incidenti su T ordinati per $D_{su} + w(u, v)$ in una coda con priorità (vedi Capitolo 8), possiamo estrarre l'arco "minimo" in tempo $O(\log n)$. In questo modo riusciamo a risolvere il problema in tempo $O(m \log n)$ nel caso peggiore invece di $O(mn)$.

Mostriamo ora che in realtà è possibile fare ancora meglio. Procederemo in modo simile a quanto fatto nel Paragrafo 12.3.1 del Capitolo 12 per l'algoritmo di Prim, ottenendo un algoritmo che richiede tempo $O(m + n \log n)$ nel caso peggiore. L'idea sarà di ridurre da $O(m)$ a $O(n)$ il numero totale di elementi che devono passare per la coda con priorità, scegliendo quest'ultima opportunamente in modo da poter effettuare il decremento di una chiave in tempo costante.

L'osservazione chiave che useremo è la seguente: se abbiamo due archi (u_1, v) e (u_2, v) con $u_1, u_2 \in V(T)$ e $v \notin V(T)$, al più uno di essi potrà finire in T, visto che T dovrà essere un albero. Quindi, non ha senso tenerli entrambi simultaneamente nella coda con priorità. Se ad esempio (u_1, v) è già in coda quando analizziamo l'arco (u_2, v) per effettuarne il rilassamento, e scopriamo che $D_{su_2} + (u_2, v) < D_{su_1} + (u_1, v)$, allora possiamo *rimpiazzare* (u_1, v) con (u_2, v) nella coda con priorità, decrementandone la priorità mediante un'operazio-

```
    algoritmo DijkstraGenerico(grafo G, vertice s) → albero
1.    inizializza D tale che D_sv = +∞ per v ≠ s, e D(s) = 0
2.    T ← albero formato dal solo nodo s
3.    while ( T ha meno di n nodi ) do
4.        trova l'arco (u, v) incidente su T con D_su + w(u, v) minimo
5.        D_sv ← D_su + w(u, v)                                  {rilassamento}
6.        rendi u padre di v in T
7.    return T
```

Figura 13.9 Versione generica dell'algoritmo di Dijkstra.

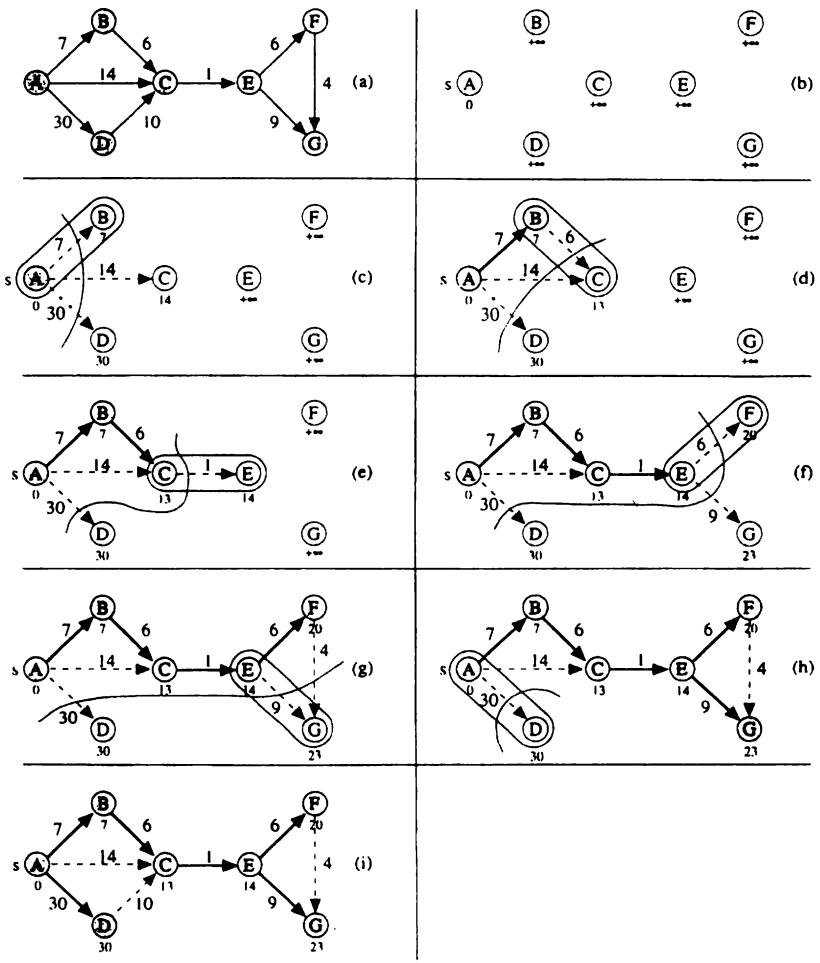


Figura 13.10 Esecuzione dell'algoritmo di Dijkstra con sorgente $s = A$. Sotto ogni vertice v è mostrato il valore corrente di D_{sv}. Gli archi dell'albero T sono in grassetto, mentre l'arco (u, v) con $u \in V(T)$ e $v \notin V(T)$ che minimizza $d_{su} + w(u, v)$ (vedi Lemma 13.8) è cerchiato. Gli archi incidenti sull'albero T sono tratteggiati.

```
algoritmo Dijkstra(grafo G, vertice s) → albero
1.   for each ( vertice u in G ) do D_su ← +∞
2.   T̂ ← albero formato dal solo nodo s
3.   CodaPriorita S
4.   D_ss ← 0
5.   S.insert(s,0)
6.   while ( not S.isEmpty() ) do
7.      u ←S.deleteMin()
8.      for each ( arco (u,v) in G ) do
9.         if (D_sv = +∞) then
10.           S.insert(v, D_su + w(u,v))
11.           D_sv ← D_su + w(u,v)
12.           rendi u padre di v in T̂
13.        else if (D_su + w(u,v) < D_sv) then
14.           S.decreaseKey(v, D_sv − (D_su + w(u,v)) ) + w(u,v))
15.           D_sv ← D_su + w(u,v)
16.           rendi u nuovo padre di v in T̂
17.  return T̂
```

Figura 13.11 Algoritmo di Dijkstra implementato con coda con priorità secondo lo schema di visita generica illustrato nella Figura 11.5 del Capitolo 11.

ne `decreaseKey`. Si noti che come semplificazione potremmo tenere nella coda con priorità direttamente il vertice v, invece che l'arco (u_1, v) o l'arco (u_2, v).

Da queste osservazioni, traiamo ora due conseguenze cruciali che ci permetteranno di ottenere un tempo di esecuzione $O(m + n \log n)$:

(1) mantenendo vertici invece che archi nella coda con priorità, essa vedrà passare al suo interno al più n elementi, e il costo totale per estrarli sarà $O(n \log n)$;
(2) rilassare un arco (u, v) richiede un'operazione `decreaseKey`$(v, D_{sv} - (D_{su} + w(u,v)))$ se $D_{su} + w(u,v) < D_{sv}$: usando una coda con priorità basata su heap di Fibonacci (vedi Capitolo 8), l'operazione `decreaseKey` ha un costo $O(1)$ ammortizzato, e quindi il tempo totale per gli $O(m)$ rilassamenti è $O(m)$.

Mettendo insieme queste idee, possiamo implementare l'algoritmo di Dijkstra come mostrato in Figura 13.11 usando lo schema di visita generica introdotto nel Paragrafo 11.3 del Capitolo 11. Si noti ancora una volta la perfetta somiglianza con l'implementazione basata su coda con priorità che avevamo dato per l'algoritmo di Prim nel Paragrafo 12.3.1 del Capitolo 12.

Commentiamo ora lo pseudocodice di Figura 13.11. Denoteremo con S la frontiera dell'albero T che contiene tutti i vertici v non in T a cui si può arrivare da T attraversando un solo arco. Nel corso dell'algoritmo, manterremo la

frontiera S usando una coda con priorità. Manterremo inoltre un albero $\widehat{T}$ che può essere pensato come ottenuto da T aggiungendo per ogni $v \in S$ l'arco (u, v) incidente a T con il minimo valore di $D_{su} + w(u, v)$. Ad ogni instante, avremo $V(T) = V(\widehat{T}) - S$. Alla fine dell'esecuzione, avremo $\widehat{T} = T$. Nel seguito faremo riferimento a T anche se non è mantenuto esplicitamente dall'algoritmo di Figura 13.11.

Il codice alle righe 1–5 ha il compito di inizializzare opportunamente le strutture dati: inizialmente l'albero $\widehat{T}$ è vuoto, l'algoritmo non ha ancora esaminato alcun arco, e la coda con priorità che realizza la frontiera S per definizione contiene solamente il vertice s, con $D_{ss} = 0$. Al generico passo, l'algoritmo seleziona il vertice u della frontiera su cui incide l'arco (z, u) in $\widehat{T} - T$ con $D_{sz} + w(z, u)$ minimo (riga 7): il vertice u viene eliminato dalla frontiera e quindi diventa un vertice di T. Le righe 8–16 si preoccupano invece di aggiornare la frontiera S dell'albero T, dopo che il vertice u è stato inglobato nell'albero T. Alla riga 8 viene considerato ogni arco (u, v) uscente dal vertice u. Se v non appartiene alla frontiera dell'albero T ($D_{sv} = +\infty$), sarà inserito nella frontiera S con priorità $D_{sv} = D_{su} + w(u, v)$ e l'arco (u, v) verrà rilassato (righe 9–12). Se invece v fa già parte della frontiera dell'albero T, confronteremo il costo D_{sv} del vecchio arco in $\widehat{T} - T$ incidente su v con il costo del nuovo arco (u, v) che stiamo esaminando: se $D_{su} + w(u, v) < D_{sv}$, allora è (u, v) ad essere il nuovo arco in $\widehat{T} - T$ incidente su v, e quindi lo rilasseremo e aggiorneremo il corrispondente valore nella coda di priorità (righe 13–16).

Teorema 13.3 *Sia $G = (V, E)$ un grafo orientato con n vertici ed m archi e con funzione costo $w : E \to \mathbb{R}^+ \cup \{0\}$ non negativa. Allora l'algoritmo di Dijkstra richiede tempo $O(m + n \log n)$ nel caso peggiore per calcolare tutte le distanze da un vertice s a tutti gli altri vertici nel grafo.*

Dimostrazione. La dimostrazione è del tutto analoga a quella del Teorema 12.5 visto nel Capitolo 12 per l'algoritmo di Prim. Il tempo di esecuzione dell'algoritmo di Dijkstra è dominato dalle operazioni sulla coda con priorità S, e quindi dipende dal tipo di struttura dati scelta per implementarla. In particolare, come possiamo desumere dalla Figura 13.11, su tale coda vengono eseguite n operazioni `deleteMin`, n operazioni `insert`, ed al più $(m - n)$ operazioni `decreaseKey`.

- Se utilizziamo come coda con priorità i d-heap presentati nel Paragrafo 8.1 del Capitolo 8, otteniamo per queste operazioni un tempo di esecuzione pari a $O(nd \log_d n + m \log_d n)$. Per $d = 2$, otteniamo un tempo di esecuzione $O(m \log n)$.
- Se scegliamo per d un valore che bilancia i due termini, come ad esempio $d = \lceil 2 + m/n \rceil$, otteniamo un tempo di esecuzione $O(m \log_{(2+m/n)} n)$. Per $m = \Omega(n^{1+k})$, con $0 < k \leq 1$, abbiamo che $O(m \log_{(2+m/n)} n) = O(m/k)$, e quindi l'algoritmo di Dijkstra con i d-heap è asintoticamente tanto più veloce di $O(m \log n)$ quanto più il grafo di partenza è denso.

- Infine, se utilizziamo come coda con priorità gli heap di Fibonacci presentati nel Capitolo 8, otteniamo per l'algoritmo di Dijkstra un tempo totale di $O(m + n \log n)$ nel caso peggiore.

□

Concludiamo il paragrafo osservando che, avendo a disposizione un algoritmo che calcola i cammini minimi a partire da una sorgente, è possibile trovare i cammini minimi tra tutte le coppie di vertici semplicemente applicando l'algoritmo a partire da ogni vertice del grafo:

Teorema 13.4 *Sia $G = (V, E)$ un grafo orientato con n vertici ed m archi e con funzione costo $w : E \to \mathbb{R}^+ \cup \{0\}$ non negativa. Allora applicando ripetutamente l'algoritmo di Dijkstra a partire da ogni vertice di G, possiamo risolvere il problema delle distanze tra tutte le coppie di vertici in tempo $O(mn + n^2 \log n)$ nel caso peggiore.*

Nel prossimo paragrafo studieremo un metodo alternativo per calcolare le distanze tra tutte le coppie di vertici in un grafo anche nel caso in cui ci siano archi con costo negativo.

13.6 Algoritmo di Floyd e Warshall

L'ultimo algoritmo che descriviamo in questo capitolo è dovuto a una combinazione di idee di Robert Floyd e Stephen Warshall, e calcola le distanze fra tutte le coppie di vertici in un grafo diretto con n vertici in tempo $O(n^3)$. Come nel caso dell'algoritmo di Bellman e Ford, i costi degli archi possono anche essere negativi, purchè non ci siano cicli negativi. L'algoritmo è basato sulla tecnica di programmazione dinamica introdotta nel Paragrafo 10.2 del Capitolo 10.

Consideriamo un grafo diretto $G = (V, E)$ con n vertici e con funzione costo $w : E \to \mathbb{R}$, e denotiamo i vertici di V come $v_1, v_2, \ldots, v_n$. Per ogni fissato k in $\{0, \ldots, n\}$, indicheremo con d^k_{xy} la distanza più corta che possiamo percorrere per andare da x a y senza passare per *nessuno* dei vertici intermedi $v_{k+1}, \ldots, v_n$. Chiameremo questa quantità *distanza k-vincolata* e il corrispondente cammino percorso *cammino minimo k-vincolato*. Più formalmente:

Definizione 13.5 (Distanze e cammini minimi k-vincolati) *Sia $G = (V, E)$ un grafo diretto con funzione costo $w : E \to \mathbb{R}$ e $V = \{v_1, v_2, \ldots, v_n\}$. Diremo che un cammino π^k_{xy} è un cammino minimo k-vincolato se esso ha il costo minimo fra quelli che, a parte gli estremi x e y, non usano nessun vertice in $\{v_{k+1}, \ldots, v_n\}$. Definiamo inoltre la distanza k-vincolata d^k_{xy} ponendo $d^k_{xy} = w(\pi^k_{xy})$ se un π^k_{xy} esiste, e $d^k_{xy} = +\infty$ altrimenti.*

È immediato osservare che $d^0_{xy} = w(x, y)$ se $(x, y) \in E$ e $d^0_{xy} = +\infty$ altrimenti. Inoltre, $d^n_{xy} = d_{xy}$. Come i cammini minimi, anche i cammini minimi k-vincolati godono della proprietà di sottostruttura ottima.


```
algoritmo FloydWarshall(grafo G) → distanze
    inizializza D tale che D_xy = w(x,y) se (x,y) ∈ E, e D_xy = +∞ altrimenti
    for each v ∈ V do
        for each ((x,y) ∈ V × V) do
            if (D_xv + D_vy < D_xy) then D_xy ← D_xv + D_vy        {rilassamento}
    return D
```

Figura 13.12 Algoritmo di Floyd e Warshall per il calcolo delle distanze tra tutte le coppie di vertici.

Lemma 13.9 (Sottostruttura ottima dei cammini minimi k-vincolati) *Sia $G = (V, E)$ un grafo diretto con funzione costo $w : E \to \mathbb{R}$ e $V = \{v_1, v_2, \ldots, v_n\}$. Allora ogni sottocammino di un cammino minimo k-vincolato in G è esso stesso un cammino minimo k-vincolato in G.*

Dimostrazione. Sia G_k il sottografo ottenuto da G rimuovendo tutti i vertici v_{k+1}, $\ldots$, v_n diversi da x e da y. Chiaramente, un cammino da x a y è un cammino minimo k-vincolato in G se e solo se è un cammino minimo in G_k. L'enunciato segue quindi immediatamente dal Lemma 13.1 applicato a G_k. □

Floyd [14], basandosi su una teorema precedente di Warshall [36], osservò che è facile costruire d^k_{xy} per ogni $x, y \in V$ se conosciamo d^{k-1}_{xy}. Essi sono infatti legati dalla seguente relazione.

Lemma 13.10 (Relazione tra distanze vincolate) *Per ogni k in $\{1, \ldots, n\}$ e per ogni $x, y \in V$, si ha:*

$$d^k_{xy} = \min\{\, d^{k-1}_{xy}, d^{k-1}_{xv_k} + d^{k-1}_{v_k y} \,\}$$

Dimostrazione. Sia π^k_{xy} un cammino minimo k-vincolato. Possono aversi due casi:

(1) v_k non è un vertice interno di π^k_{xy}. Allora π^k_{xy} è anche un cammino minimo $(k-1)$-vincolato, da cui $d^k_{xy} = d^{k-1}_{xy}$;

(2) v_k è un vertice interno di π^k_{xy}. In base al Lemma 13.9 i sottocammini di π^k_{xy} da x a v_k e da v_k a y sono cammini minimi k-vincolati. Poiché essi non usano v_k come vertice interno, essi sono anche cammini minimi $(k-1)$-vincolati. Ne segue che $d^k_{xy} = d^{k-1}_{xv_k} + d^{k-1}_{v_k y}$.

Allora il costo d^k_{xy} di π^k_{xy} deve essere necessariamente uguale al minimo fra d^{k-1}_{xy} e $d^{k-1}_{xv_k} + d^{k-1}_{v_k y}$. □

Usando la tecnica di programmazione dinamica introdotta nel Capitolo 10, possiamo procedere dal basso verso l'alto e calcolare prima $D_{xy} = d^0_{xy}$ per ogni x, y,

poi usare il Lemma 13.10 per calcolare $D_{xy} = d^1_{xy}$ per ogni x, y, e così via fino ad ottenere $D_{xy} = d^n_{xy} = d_{xy}$ per ogni x, y. L'algoritmo è illustrato in Figura 13.12. In un grafo con n vertici il tempo richiesto è chiaramente $O(n^3)$ poiché l'inizializzazione (riga 1) richiede tempo $O(n^2)$ e ripetiamo n volte (riga 2) un'operazione di aggiornamento di D_{xy} per ogni x, y (righe 3–4).

13.7 Problemi

Problema 13.1 (Grafo non orientato come spazio metrico) Sia $G = (V, E)$ un grafo non orientato con funzione costo positiva $w : E \to \mathbb{R}^+$. Dimostrare che V e la funzione distanza d_{xy} data in Definizione 13.3 formano uno *spazio metrico*, e cioè che valgono le seguenti proprietà per ogni tripla di vertici x, y e z:

1. $d_{xy} \geq 0$;
2. $d_{xx} = 0$;
3. se $d_{xy} = 0$, allora $x = y$;
4. $d_{xy} = d_{yx}$ (simmetria);
5. $d_{xz} \leq d_{xy} + d_{yz}$ (disuguaglianza triangolare);

Problema 13.2 Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Dimostrare che un arco $(u, v) \in E$ appartiene a un cammino minimo tra i vertici s e v se e solo se u è raggiungibile da s e $d_{su} + w(u, v) = d_{sv}$.

Problema 13.3 Dimostrare che in ogni grafo diretto aciclico esiste almeno un vertice che non ha archi entranti.

Problema 13.4 (Cammino più costoso in un grafo aciclico) Sia $G = (V, E)$ un grafo orientato aciclico con funzione costo $w : E \to \mathbb{R}$. Progettare un algoritmo che calcola in tempo $O(m + n)$ il costo del cammino più costoso da un vertice sorgente s a ogni altro vertice v. Cosa succederebbe se il grafo non fosse aciclico?

Problema 13.5 (Cammini minimi dinamici con soli inserimenti) Un algoritmo dinamico è un algoritmo che è in grado di aggiornare il risultato di una funzione a fronte di modifiche dei dati di ingresso senza doverlo ricalcolare da capo. Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}$. Progettare un algoritmo dinamico che è in grado di aggiornare le distanze fra tutte le coppie di vertici del grafo a fronte dell'inserimento di un nuovo arco nel grafo. Se l'inserimento dell'arco introduce un ciclo negativo nel grafo, l'algoritmo deve essere in grado di lanciare un'eccezione. Quali difficoltà sorgerebbero se volessimo cancellare archi invece di inserirli?

Problema 13.6 (Dijkstra su grafi con costi arbitrari) Esibire un grafo in cui alcuni archi hanno costo negativo su cui l'algoritmo di Dijkstra non produce un risultato corretto. Esistono casi in cui l'algoritmo di Dijkstra produce un risultato corretto nonostante la presenza di archi con costo negativo nel grafo?

Problema 13.7 (Cammini con età) Sia $G = (V, E)$ un grafo orientato in cui ogni arco ha associata una certa età. Definiamo l'età di un cammino come l'età dell'arco con età minima attraversato nel cammino. Sia s un vertice sorgente da cui ogni altro vertice è raggiungibile. Progettare un algoritmo che sia in grado di trovare per ogni $v \neq s$ un cammino con la massima età possibile che connette s a v. L'algoritmo funziona anche con età negative? Cosa potrebbe succedere se accettassimo unche archi con costi zero?

Problema 13.8 (Sottografo dei cammini minimi) Sia $G = (V, E)$ un grafo orientato con funzione costo $w : E \to \mathbb{R}^+$ strettamente positiva. Dato un vertice sorgente s, denotiamo con $G_s = (V, E_s)$ il sottografo di G tale che per ogni $(u, v) \in E$, $(u, v) \in E_s$ se e solo se $d_{su} + w(u, v) = d_{sv}$. Dimostrare che G_s, noto come *grafo dei cammini minimi*, è aciclico.

Problema 13.9 Dato un grafo non orientato G in cui gli archi hanno tutti costo 1, sia $diametro(G) = \max_{u,v \in V} d_{uv}$. Dimostrare formalmente o confutare la seguente affermazione: esiste un grafo con diametro 10 che ha un albero di BFS (vedi il Paragrafo 11.3.1 del Capitolo 11) di altezza 4. Qual è il massimo valore del diametro in un grafo che ha un albero di BFS di altezza t?

13.8 Sommario

In questo capitolo abbiamo studiato il problema di trovare cammini con costo minimo in grafi orientati, dove il costo di un cammino è la somma dei costi degli archi che lo compongono. Se ad esempio il grafo modella una rete stradale, il costo $w(u, v)$ di un arco (u, v) potrebbe essere il tempo di percorrenza tra le città u e v. Se i costi degli archi possono essere unche minori di zero, è importante che nel grafo non ci siano cicli con costo negativo, altrimenti trovare cammini minimi di lunghezza finita diventa estremamente difficile.

Sotto l'ipotesi quindi che nel grafo non ci siuno cicli negativi, abbiamo discusso alcune proprietà salienti dei cammini minimi, come il fatto che ogni sottocammino di un cammino minimo è unch'esso minimo, e il fatto che i cammini minimi a partire da un vertice sorgente s possono essere rappresentati mediunte un albero. La nozione di cammino minimo ci ha permesso di introdurre il concetto di distanza fra vertici in un grafo, che gode di proprietà interessunti comuni agli spazi metrici come la disuguagliunza triungolare. Abbiamo visto che in generale la conoscenza delle distanze è sufficiente a ricostruire i cammini minimi in un grafo.

Nella seconda parte del capitolo, abbiamo descritto alcuni algoritmi classici per il calcolo dei cammini minimi, tutti basati su una stessa tecnica generale che consiste nel muntenere delle stime per eccesso delle distanze nel grafo, decrementandole progressivamente finché non diventano corrette. Questo avviene mediunte l'applicazione ripetuta di un'operazione nota come *rilassamento*, che verifica violazioni locali della proprietà di disuguaglianza triangolare, e aggiorna le stime delle distanze per eliminarle.

cammini minimi	a sorgente singola	fra tutte le coppie
`BellmanFord`	$O(mn)$	$O(mn^2)$
`distanzeAciclico`	$O(m+n)$	$O(mn+n^2)$
`Dijkstra`	$O(m+n\log n)$	$O(mn+n^2\log n)$
`FloydWarshall`	-	$O(n^3)$

Tabella 13.1 Tempo nel caso peggiore richiesto dalla migliore implementazione nota di ciascuno degli algoritmi visti in questo capitolo per calcolare cammini minimi a sorgente singola o tra tutte le coppie in un grafo con n vertici e m archi. Per calcolare i cammini minimi tra tutte le coppie usando `BellmanFord`, `distanzeAciclico` e `Dijkstra` assumiamo di ripetere l'esecuzione a partire da ogni vertice del grafo.

Come primo caso, abbiamo studiato l'algoritmo di Bellman e Ford. Abbiamo visto che, dato un grafo con n vertici e m archi, esso è in grado di calcolare le distanze da un vertice sorgente a ogni altro vertice in tempo $O(mn)$. Sorprendentemente, a quasi 50 anni dalla sua apparizione questo è ancora il migliore algoritmo noto per questo problema se i costi degli archi possono essere anche negativi! Denotando con D_{xy} la distanza stimata tra x e y, l'algoritmo parte con $D_{ss} = 0$ e $D_{sv} = +\infty$ per ogni $v \neq s$, e applica ripetutamente il passo di rilassamento $D_{sv} \leftarrow D_{su} + w(u,v)$ su ogni arco (u,v) tale che la disuguaglianza $D_{sv} \leq D_{su} + w(u,v)$ è violata.

Abbiamo poi visto che, traendo vantaggio da vincoli sulla struttura del grafo o sui costi degli archi, possiamo ridurre sostanzialmente il tempo di esecuzione. Nel caso di grafi aciclici, è infatti possibile ordinare i vertici in modo che per ogni coppia di vertici u e v, u preceda v nell'ordinamento se esiste un cammino da u a v. Effettuando le operazioni di rilassamento secondo questo ordine, noto come *ordine topologico*, si ottiene una variante dell'algoritmo di Bellman e Ford che rilassa ogni arco una volta soltanto e richiede tempo totale $O(m+n)$.

Se invece il grafo non è aciclico, ma i costi degli archi sono maggiori o uguali a zero, possiamo usare l'algoritmo di Dijkstra, che richiede tempo $O(m+n\log n)$. L'algoritmo sfrutta una interessante proprietà delle distanze in un grafo che permette di estendere in modo goloso un albero di cammini minimi aggiungendo progressivamente archi (u,v) estratti da una coda con priorità in ordine di $D_{su} + w(u,v)$ crescente. Le operazioni di rilassamento che aggiornano le stime delle distanze sono pertanto effettuate in un ordine ben preciso che, come nel caso dell'ordine topologico per i grafi aciclici, ci permette di non rilassare mai due volte lo stesso arco. L'uso di una coda con priorità per determinare l'ordine di rilassamento introduce però un fattore logaritmico che non avevamo nel caso di grafi aciclici. Sorprendentemente, l'algoritmo di Dijkstra è molto simile a quello di Prim per il calcolo di un minimo albero ricoprente visto nel Capitolo 12. Entrambi gli algoritmi fornisconо esempi significativi di quanto l'uso di strutture dati opportune possa essere cruciale nel progetto di algoritmi efficienti.

Come ultimo argomento, abbiamo studiato l'algoritmo di Floyd e Warshall, che calcola le distanze tra tutte le coppie di vertici in tempo $O(n^3)$ utilizzando la tecnica della programmazione dinamica. In particolare, l'algoritmo calcola quelle che abbiamo chiamato distanze k-vincolate, cioè distanze ottenute usando cammini che non passano per nessuno dei vertici $v_{k+1}, \ldots, v_n$, posto che i vertici del grafo siano $v_1, \ldots, v_n$. Per ogni coppia di vertici x, y la distanza k-vincolata d^k_{xy} può essere ottenuta prendendo il minimo della distanza $(k-1)$-vincolata d^{k-1}_{xy} e della somma delle distanze $(k-1)$-vincolate $d^{k-1}_{xv_k} + d^{k-1}_{v_k y}$. Partendo da $d^0_{xy} = w(x,y)$ per ogni arco (x,y) del grafo e $d^0_{xy} = +\infty$ altrimenti, è immediato sfruttare questa proprietà usando la tecnica della programmazione dinamica per calcolare d^1_{xy} da d^0_{xy} per ogni x, y, poi d^2_{xy} da d^1_{xy} per ogni x, y, e così via. Alla fine, poiché $d^n_{xy} = d_{xy}$, otteniamo le distanze nel grafo. Le prestazioni degli algoritmi descritti in questo capitolo sono riassunte nella Figura 13.1.

13.9 Note bibliografiche

Per la sua fondamentale importanza in molti scenari applicativi come ad esempio le reti di telecomunicazioni e di trasporti, il problema dei cammini minimi gode di una letteratura vastissima. I primi articoli che descrivono metodi efficienti per il calcolo delle distanze in un grafo risalgono agli anni '50.

Il primo a usare la tecnica del rilassamento fu Lestor Ford nel 1956 [15]. Negli anni seguenti, molti ricercatori ne studiarono le proprietà, tra cui Edward Moore [28] e ancora Lestor Ford insieme a Ray Fulkerson [16]. L'algoritmo di Bellman e Ford con tempo di esecuzione $O(mn)$ che abbiamo descritto nel Paragrafo 13.3 è stato proposto da Richard Bellman nel 1958 [4] come variante dell'algoritmo generico basato su rilassamento descritto precedentemente da Ford in [15]. Come già osservato, a quasi mezzo secolo dalla sua apparizione, questo è ancora il migliore algoritmo noto per il problema dei cammini minimi a sorgente singola nel caso in cui i costi degli archi possono essere anche negativi.

L'algoritmo per grafi aciclici studiato nel Paragrafo 13.4 è descritto da Eugene Lawler in [27] come un risultato folkloristico.

Le idee racchiuse nel Lemma 13.8, che hanno permesso di ottenere i primi algoritmi con tempo di esecuzione migliore di $O(mn)$ per grafi con costi non negativi, sono state scoperte intorno al 1959-60 da Dijkstra [9] e indipendentemente da George Dantzig [8] e P.D. Whiting e J.A. Hillier [37]. Nel suo articolo originale del 1959 [9], Dijkstra aveva proposto un'implementazione del suo algoritmo simile a quella di Figura 13.11, con un tempo di esecuzione $O(n^2)$: i vertici della frontiera erano mantenuti in una lista non ordinata e ogni minimo veniva estratto in tempo $O(n)$. In effetti, in quell'articolo non si parlava ancora di code con priorità, il cui studio sistematico sarebbe iniziato solo molti anni dopo. Il primo a proporre l'uso di code con priorità per implementare l'algoritmo di Dijkstra fu Donald B. Johnson in un articolo del 1977 [24]. La migliore implementazione nota, basata sugli heap di Fibonacci descritti nel Capitolo 8, è stata descritta da Michael Fredman e Robert Tarjan nel 1987 [18].

Esistono anche algoritmi specializzati per cammini minimi a sorgente singola in grafi con costi interi $\leq C$, come l'implementazione dell'algoritmo di Dijkstra proposta nel 1969 da Robert Dial [11] e indipendentemente da Robert Wagner nel 1976 [35], che richiede tempo $O(m + nC)$. Ancora Donald Johnson [25], basandosi su lavori precedenti di Peter Van Emde Boas, Robert Kaas ed Erik Zijlstra [5] ha proposto nel 1982 un'ulteriore variante dell'algoritmo di Dijkstra che richiede tempo $O(m \log \log C)$. Progettando una coda con priorità per valori interi che supporta l'operazione `decreaseKey` in tempo $O(\log \log \min\{n, C\})$, Mikkel Thorup [34] ha ottenuto nel 2003 un'implementazione dell'algoritmo di Dijkstra che garantisce il migliore tempo di esecuzione noto per grafi con costi interi pari a $O(m + n \log \log \min\{n, C\})$. Altri risultati per grafi con costi interi appaiono in [2, 10, 12, 19].

L'algoritmo di Floyd e Warshall per il calcolo delle distanze fra tutte le coppie di vertici che abbiamo descritto nel Paragrafo 13.6 è stato pubblicato da Robert Floyd nel 1962 [14], e si basa su un risultato di Stephen Warshall [36] dello stesso anno per il problema della raggiungibilità fra vertici in un grafo orientato (anche noto come problema della chiusura transitiva).

Si noti che eseguendo la migliore implementazione dell'algoritmo di Dijsktra a partire da ogni vertice possiamo calcolare i cammini minimi tra tutte le coppie in tempo $O(mn + n^2 \log n)$ (vedi Figura 13.1), dove m è il numero di archi nel grafo. Per grafi con $m = o(n^2)$, questo è asintoticamente migliore del tempo $O(n^3)$ ottenibile con l'algoritmo di Floyd e Warshall. Recentente, Seih Petie [29] ha mostrato un metodo che richiede tempo $O(mn + n^2 \log \log n)$, ottenendo quindi un ulteriore miglioramento asintotico per grafi con $m = o(n \log n)$.

Ancora una volta, nel caso speciale in cui i costi degli archi siano interi piccoli non negativi la situazione è migliore. Usando un approccio completamente diverso basato su proprietà aritmetiche degli interi, è possibile ridurre il problema delle distanze fra tutte le coppie di vertici al problema di moltiplicare matrici, per cui sono noti algoritmi subcubici come abbiamo visto nel Paragrafo 10.1.2 del Capitolo 10 (si veda anche [7, 32]): Noga Alon, Zvi Galil e Oded Margalit [3] sono stati i primi negli anni '90 a mostrare come farlo. Il migliore risultato attuale per grafi orientati con costi interi positivi $\leq C$ è dovuto a Uri Zwick [38], che nel 1998 ha mostrato come calcolare le distanze fra tutte le coppie di vertici in tempo $O(C^{0.681} n^{2.575})$. Altri algoritmi di questo tipo per grafi orientati e non orientati sono descritti in [20, 21, 30, 31].

Nonostante decenni di ricerca e centinaia di articoli pubblicati sul problema, è tutt'ora un problema aperto progettare un algoritmo per calcolare le distanze fra tutte le coppie di vertici con tempo di esecuzione $O(n^{3-\epsilon})$ per qualche $\epsilon > 0$ nel caso peggiore in cui $m = \Theta(n^2)$ e con costi degli archi non interi. Un primo passo in questa direzione è stato fatto da Michael Fredman [17] e in seguito da Tadao Takaoka [33] che hanno mostrato come infrangere la barriera cubica di un fattore polilogaritmico: il migliore risultato di questo tipo è dovuto a Takaoka, che ha mostrato un algoritmo con tempo di esecuzione $O(n^3 \sqrt{\log \log n / \log n})$. David Karger, Daphne Koller e Steven Phillips [26] hanno mostrato nel 1993 che, nel caso generale di algoritmi basati su confronti di costi di cammini come quelli che abbiamo visto in questo capitolo, esiste un limite inferiore $\Omega(mn)$ al tempo necessario per risolvere il problema. Sorprendentemente quindi, per ottenere algoritmi con tempo di esecuzione subcubico nel caso peggiore in cui $m = \Theta(n^2)$, dovremmo progettare algoritmi che effettuano confronti tra valori che *non* rappresentano costi di cammini del grafo!

Un'ottima trattazione del problema dei cammini minimi appare nel testo di Ravindra Ahuja, Thomas Magnanti e James Orlin [1]. I risultati di an'approfondita analisi sperimentale di algoritmi per cammini minimi sono riportati da Boris Cherkassky, Andrew Goldberg e Tomasz Radzik in un articolo del 1996 [6]. Fra gli altri risultati sperimentali, ricordiamo quelli pobblicati da Giorgio Gallo e Stefano Pallottino nel 1988 [22] e da Divoky e Hung nel 1988 [23] e nel 1990 [13].

Riferimenti bibliografici

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] R.K. Ahuja, K. Mehlhorn, J.B. Orlin, R.E. Tarjan. Faster Algorithms for the Shortest Path Problem. *Journal of the ACM* 37(2): 213–223 (1990).

[3] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences*, 54(2):255–262, April 1997.

[4] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[5] P. Van Emde Boas, R. Kaas and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10, 99–127, 1977.

[6] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[8] G.B. Dantzig. On the Shortest Route Through a Network. *Management Science*, 2, 187-190, 1960. Method also appears in: G.B. Dantzig, Linear Programming and Extension, Princeton, New Jersey: Princeton University Press, 1963.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] E. V. Denardo and B. L. Fox. Shortest-route methods: 1, reaching, pruning and buckets. *Operations Research*, 27:161–186, 1979.

[11] Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969.

[12] R. Dial, F. Glover, D. Kamey and D.Klingman. A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. *Networks* 9, 215–248, 1979.

[13] J.J. Divoky and M.S. Hung.

Performance of shortest path algorithms in network flow problems. *Management Science*, 36(6), 661–673, 1990.
[14] Robert W. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 5(6):345, 1962.
[15] Lestor R. Ford. Network flow theory. Report P-923, Rand. Corp., Santa Monica, CA, 1956.
[16] Lestor R. Ford, Jr., and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
[17] M. L. Fredman. New bounds on the complexity of the shortest path problems. *SIAM Journal on Computing*, pages 87–89, 1976.
[18] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms". *Journal of the ACM* 34, (1987), 596–615.
[19] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.
[20] Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103–139, 1 May 1997.
[21] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243–254, April 1997.
[22] G. Gallo, S. Pallottino. Shortest Path Algorithms. *Annals of Operations Research*. J.C. Baltzer Scientific Publishing Company, 1988.
[23] Ming S. Hung and James J. Divoky. A computational study of efficient shortest path algorithms. *Computers and Operations Research*, 15(6):567–576, 1988.
[24] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM* 24, (1977), 1–13.
[25] Donald B. Johnson. A priority queue in which initialization and queue operations take O(loglog D) time. *Mathematical Systems Theory*, 15, 295–310, 1982.
[26] D. Karger, D. Koller, and S.J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.
[27] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.
[28] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, 285–292. Harvard University Press, 1959.
[29] Seth Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1), (2004), 47–74.
[30] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, December 1995.
[31] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *40th Annual Symposium on Foundations of Computer Science: October 17–19, 1999, New York City, New York,*, pages 605–614, 1999.

[32] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14:354–356, 1969.
[33] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, September 1992.
[34] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, June 09-11, 2003, San Diego, CA, USA.
[35] Robert A. Wagner. A Shortest Path Algorithm for Edge-Sparse Graphs. *Journal of the ACM* 23(1): 50–57 (1976).
[36] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.
[37] P.D. Whiting and J.A. Hillier. A Method for Finding the Shortest Route through a Road Network. *Operations Research Quarterly*, 11, 37–40, 1960.
[38] U. Zwick. All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms. In *Proc. of the 39th IEEE Annual Symposium on Foundations of Computer Science (FOCS'98)*, pages 310–319, Los Alamitos, CA, November 8–11, 1998.

14

Flusso

Πάντα ῥεῖ
(Eraclito)

Come abbiamo visto nel Capitolo 13, nel problema dei cammini minimi attraversare un arco per spostarsi da un vertice all'altro ha un costo, e percorrere un cammino minimo significa muoversi essendo sicuri di incorrere nel minimo costo totale possibile por raggiungere una data destinazione. Questo costo è però completamente indipendente dalla presenza simultanea di altri oggetti, come pacchetti di dati in una rete di comunicazione, automobili in una rete stradale, o molecole di liquido in una rete di distribuzione idrica, che si muovono attraversando gli stessi archi nello stesso tempo. In molti casi, è realistico pensare che un arco, oltre ad avere un costo, abbia anche una capacità, che rappresenta il massimo numero di oggetti che lo possono attraversare simultaneamente. Data una rete, in questo capitolo studieremo come calcolare il massimo numero di oggetti che, partendo da un vertice sorgente, possono raggiungere un vertice destinazione su più cammini simultaneamente senza eccedere le capacità degli archi. Questo è noto come il problema del *flusso massimo in una rete*. Il problema dei cammini minimi e quello del flusso massimo sono in un certo modo complementari: l'uno trova cammini che minimizzano il costo di percorrenza senza preoccuparsi delle capacità degli archi, l'altro massimizza la quantità di oggetti trasferiti senza eccedere le capacità degli archi, ma non si preoccupa del costo di percorrenza. Una combinazione di questi due problemi, di cui rimandiamo la trattazione a testi specifici su flusso nelle reti [1], è nota come problema del *flusso massimo a costo minimo*, e modella efficacemente nella loro generalità molti problemi di distribuzione, trasporto e comunicazione. Soluzioni efficienti per questo problema si basano sull'uso combinato tecniche sviluppate in questo capitolo e nel Capitolo 13.

14.1 Reti di flusso

Una *rete di flusso* è un grafo diretto $G = (V, E, s, t, c)$, dove s e t sono due vertici speciali chiamati rispettivamente *sorgente* e *pozzo*, e c è una funzione *capacità*

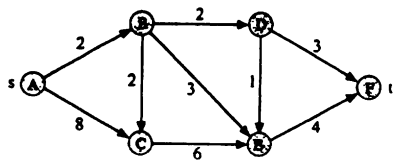


Figura 14.1 Rete di flusso $G = (V, E, s, t, c)$. Le capacità sono mostrate a fianco degli archi. La sorgente e il pozzo sono indicati con s e t, rispettivamente.

$c : V^2 \rightarrow \mathbb{R}^+$ che associa ad ogni arco $(u, v) \in E$ una capacità strettamente positiva $c(u, v)$, mentre $c(u, v) = 0$ se $(u, v) \notin E$. In Figura 14.1 è mostrato un esempio di rete di flusso. Si noti che $c(A, B) = 2$, mentre $c(B, A) = 0$ poiché $(B, A) \notin E$.

Data una rete di flusso, per ogni arco saremo interessati a determinare quanti oggetti in transito dalla sorgente al pozzo potranno attraversarlo simultaneamente senza eccederne la capacità. In uno scenario continuo e non discreto, accetteremo che questo numero, che chiameremo *flusso* associato all'arco, possa essere frazionario.

Facendo un'analogia idraulica, possiamo immaginare che gli archi di una rete di flusso in cui circola del liquido siano tubi di un certo diametro che ne determina la portata (capacità), e il flusso in ogni tubo rappresenta la quantità di liquido in transito dalla sorgente al pozzo che lo attraversa nell'unità di tempo. Si noti che la lunghezza dei tubi è del tutto irrilevante: a parità di diametro, in un tubo lungo un metro o dieci metri potremo far passare la stessa quantitià di liquido nell'unità di tempo. Più formalmente, definiremo un flusso nel modo seguente.

Definizione 14.1 (Flusso) *Un* (assegnamento di) flusso *su G è una funzione $f : V^2 \rightarrow \mathbb{R}^+ \cup \{0\}$ che soddisfa i seguenti tre vincoli:*

(1) *Capacità:*	$f(u, v) \leq c(u, v)$	$\forall (u, v) \in V \times V$
(2) *Conservazione:*	$\sum_{v \in V} f(u, v) = 0$	$\forall u \in V - \{s, t\}$
(3) *Antisimmetria:*	$f(u, v) = -f(v, u)$	$\forall (u, v) \in V \times V$

In base alla Definizione 14.1, il flusso associato ad ogni coppia di vertici non deve eccedere la capacità di quella coppia (vincolo di capacità) e per ogni vertice distinto dalla sorgente e dal pozzo, la somma algebrica dei flussi passanti per quel vertice deve essere zero (vincolo di conservazione). Si noti inoltre che, in base al vincolo di capacità, f può essere diversa da zero solo per le coppie connesse da un arco nella rete.

La *quantità di flusso* inviato da s a t è semplicemente il flusso uscente dalla sorgente, come definito sotto.

Definizione 14.2 (Quantità di flusso) *Dato un flusso f, denotiamo con $|f|$ la quantità di flusso che esce dalla sorgente:*

$$|f| = \sum_{(s,v) \in E} f(s, v)$$

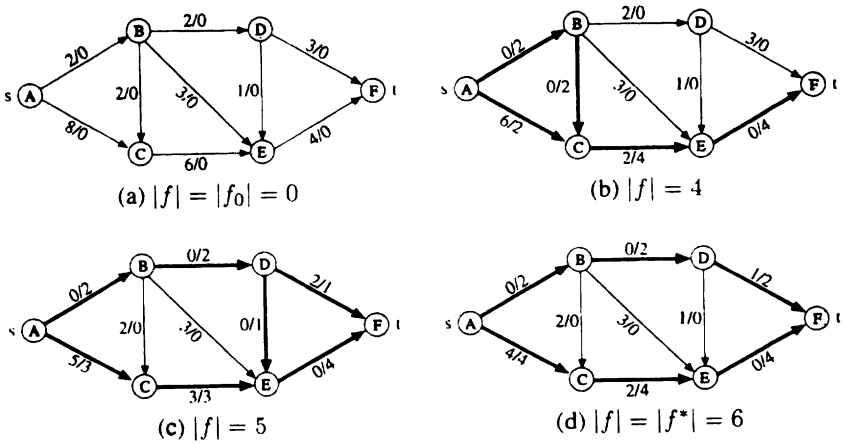


Figura 14.2 Quattro diverse assegnazioni di flusso agli archi della rete di Figura 14.1. Per ogni arco (x, y) indichiamo la capacità residua $r(x, y)$ e il flusso assegnato $f(x, y)$ con la notazione $r(x, y)/f(x, y)$ a fianco dell'arco stesso. Gli archi che portano flusso > 0 sono in grassetto.

Diremo che un flusso è nullo se è zero ovunque:

Definizione 14.3 (Flusso nullo) *Definiamo* flusso nullo *la funzione $f_0 : V^2 \rightarrow \mathbb{R}^+ \cup \{0\}$ tale che $f_0(x, y) = 0$ per ogni arco $(x, y) \in G$.*

È immediato verificare che questa definizione soddisfa i vincoli di capacità, conservazione e antisimmetria visti sopra. Si osservi inoltre che $|f_0| = 0$. Un flusso massimo f^* è invece un flusso con il massimo $|f^*|$:

Definizione 14.4 (Flusso massimo) *Definiamo* flusso massimo *un flusso f^* tale che $|f^*| = max_f |f|$.*

La *capacità residua* di un arco rispetto a una data assegnazione di flusso è la quantità di flusso che possiamo ancora mandare su quell'arco senza eccederne le capacità:

Definizione 14.5 (Capacità residua) *Definiamo* capacità residua *di un arco la quantità $r(u, v) = c(u, v) - f(u, v)$.*

Si noti che, poiché $f(u, v) \leq c(u, v)$ in base alla Definizione 14.1, si ha sempre $r(u, v) \geq 0$.

Esempio 14.1 In Figura 14.2 mostriamo quattro possibili assegnamenti di flusso sulla rete di Figura 14.1. Il primo assegnamento (Figura 14.2 (a)) è un flusso

nullo, e quindi $|f| = 0$. Per il secondo (Figura 14.2 (b)) si ha $|f| = 4$, mentre per il terzo (Figura 14.2 (c)) si ha $|f| = 5$. L'ultimo assegnamento (Figura 14.2 (d)) è un flusso massimo $f = f^*$ con $|f| = |f^*| = 6$. □

Concludiamo questo paragrafo definendo la somma di flussi, che useremo in seguito.

Definizione 14.6 (Somme di flussi) *Per ogni coppia di flussi f_1 e f_2 in G, definiamo $f_1 + f_2$ come $(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$.*

14.2 Metodo delle reti residue

Un metodo molto generale e potente per calcolare un flusso massimo in una rete è il *metodo delle reti residue*, che illustreremo in questo paragrafo.

Nella sua versione iterativa, il metodo è incrementale: si parte da un flusso nullo e lo si aumenta progressivamente finché non diventa un flusso massimo. L'idea è la seguente. Se abbiamo una rete G di cui conosciamo un flusso f (in generale non massimo), possiamo costruire una *rete residua* G_f derivata da G (che definiremo formalmente più avanti) tale che, se f' è un flusso in G_f, allora $f + f'$ è un flusso in G. Possiamo allora incrementare il flusso f in G effettuando il seguente *passo aumentante*:

(PASSO AUMENTANTE) *Troviamo un flusso f' non nullo in G_f, se esiste, e aggiungiamo f' a f.*

Tenendo aggiornata la rete residua G_f rispetto a f, l'applicazione ripetuta di questo passo permette in generale di arrivare a un flusso massimo. Questo metodo generico è illustrato in Figura 14.3.

Se le capacità degli archi della rete sono intere, l'algoritmo arriva sempre a trovare un flusso massimo in tempo finito. La "velocità" di convergenza del metodo dipende da come viene scelto il flusso f'. Nel Paragrafo 14.4 vedremo un modo di scegliere f' in modo da ottenere un algoritmo con tempo di esecuzione $O(m \cdot |f^*|)$, dove $|f^*|$ è la quantità di flusso massimo trovata. Con una semplice modifica della strategia di scelta di f', nel Paragrafo 14.5 vedremo come scendere a $O((m + n \log n) \cdot m \log |f^*|)$. I migliori algoritmi noti basati su questo metodo arrivano a $O(mn \log n)$, cioè sono indipendenti dal valore del flusso massimo calcolato (si veda [1] per una descrizione approfondita di questi algoritmi).

Sorprendentemente, esistono casi con capacità reali in cui l'algorimo potrebbe richiedere invece tempo infinito! Questo può succedere ad esempio se troviamo ad ogni passo un flusso f' tale che $|f'| = (|f^*| - |f|)/2$. In questo modo, $|f|$ si avvicina asintoticamente a $|f^*|$ in funzione del numero di iterazioni senza raggiungerlo mai.

Nel resto di questo paragrafo mostreremo come costruire una rete residua (riga 3 di Figura 14.3), mentre nel Paragrafo 14.3 vedremo un metodo generale per trovare (se esiste) un flusso f' con $|f'| > 0$ in una rete residua G_f (riga 4 di


```
   algoritmo flussoMassimo(rete G) → flusso
1.   f ← f0
2.   repeat
3.       calcola la rete residua G_f di G
4.       trova un flusso f' in G_f tale che |f'| > 0, se esiste
5.       if ( un tale f' non esiste in G_f ) then break
6.       f ← f + f'
7.   return f
```

Figura 14.3 Metodo generico iterativo per il calcolo del massimo flusso in una rete basato sulla tecnica delle reti residue.

Figura 14.3). Come già osservato, a seconda di come viene trovato il flusso f', il metodo generico si specializza dando luogo ad algoritmi con prestazioni diverse.

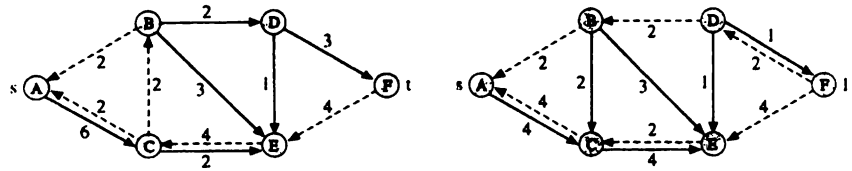
Definiamo formalmente una *rete residua* nel modo seguente.

Definizione 14.7 (Rete residua) *Data una rete di flusso $G = (V, E, s, t, c)$ e un flusso f su G, definiamo* rete residua *la rete di flusso $G_f = (V, E_f, s, t, r)$ in cui $(u, v) \in E_f$ se e solo se $r(u, v) > 0$.*

Osserviamo che la rete residua G_f è una rete di flusso sugli stessi vertici di G e ha come capacità le capacità residue di G rispetto a f. Inoltre, gli archi di G_f collegheranno le stesse coppie di vertici originariamente collegate in G. Notiamo poi che, se un arco (u, v) in G ha un flusso associato $f(u, v)$ con $0 < f(u, v) < c(u, v)$, allora questo arco si sdoppierà in G_f in un *arco in avanti* (u, v) con capacità $r(u, v) = c(u, v) - f(u, v)$ e in un *arco all'indietro* (v, u) con capacità $r(v, u) = c(v, u) - f(v, u) = c(v, u) + f(u, v)$. Il significato degli archi all'indietro sarà chiarito nel seguito. Poiché (v, u) potrebbe essere assente in G, G_f potrebbe avere strettamente più archi di G, ma mai più del doppio. Si noti che, se f è un flusso nullo, allora $G_f = G$.

Esempio 14.2 In Figura 14.4 sono mostrate le reti residue derivate dalla reti (b) e (d) di Figura 14.2. Si osservi che ad esempio l'arco (C, E) è sdoppiato nella rete residua G_f di Figura 14.4 (a) in un arco in avanti (C, E) (disegnato come una linea continua) con capacità residua $r(C, E) = 2$ e un arco all'indietro (E, C) (tratteggiato) con capacità residua $r(E, C) = 4$. Si noti che $r(C, E) + r(E, C) = c(C, E) = 6$, mentre $r(C, E) + r(E, C) \neq c(E, C) = 0$. Poiché $r(E, C) = -f(E, C) = f(C, E)$, per capire quanto flusso passa sull'arco in avanti (C, E) basta guardare la capacità residua del corrispondente arco all'indietro tratteggiato (E, C). □

Per dimostrare la correttezza del metodo di Figura 14.3, rimane da dimostrare che calcolare un flusso f' in G_f ci permette di aumentare il flusso f in G.



(a) $|f| = 4$ (b) $|f| = |f^*| = 6$

Figura 14.4 (a) Rete residua derivata dalla rete (b) di Figura 14.2. (b) Rete residua derivata dalla rete (d) di Figura 14.2. Le linee continue rappresentano archi in avanti di G_f che appartengono anche a G, mentre gli archi tratteggiati sono quelli all'indietro (y, x) che appaiono in G_f quando su (x, y) viene inviato del flusso. Le etichette degli archi rappresentano le loro capacità residue.

Lemma 14.1 (Relazione tra flussi in G e in G_f) *Se f è un flusso in G e f' è un flusso in G_f, allora $f + f'$ è un flusso in G.*

Dimostrazione. Poichè f' è un flusso in G_f, valgono le seguenti relazioni:

(1) Capacità:	$f'(u,v) \leq r(u,v)$	$\forall (u,v) \in V \times V$
(2) Conservazione:	$\sum_{v \in V} f'(u,v) = 0$	$\forall u \in V - \{s,t\}$
(3) Antisimmetria:	$f'(u,v) = -f'(v,u)$	$\forall (u,v) \in V \times V$

Per dimostrare che $f + f'$ è un flusso in G, basta: sommare $f(x,y)$ a entrambi i membri della relazione (1) e usare l'equazione $r(x,y) = c(x,y) - f(x,y)$; sommare $\sum_{v \in V} f(u,v)$ a entrambi i membri della relazione (2); sommare $f(x,y)$ a entrambi i membri della relazione (3). □

Osserviamo che la condizione del Lemma 14.1 è sia necessaria che sufficiente (vedi Problema 14.7). Questo implica che, se f non è massimo, allora esiste sempre in G_f un f' tale che $f + f' = f^*$, e quindi il metodo termina solo quando è stato trovato un flusso massimo.

---

algoritmo `flussoMassimoRic`($rete\ G$) $\rightarrow$ *flusso*
1. trova un flusso f in G tale che $|f| > 0$, se esiste
2. **if** (un tale f non esiste in G) **then return** f_0
3. calcola la rete residua G_f di G
4. $f^* \leftarrow f +$ `flussoMassimoRic`(G_f)
5. **return** f^*

Figura 14.5 Versione ricorsiva del metodo delle reti residue.

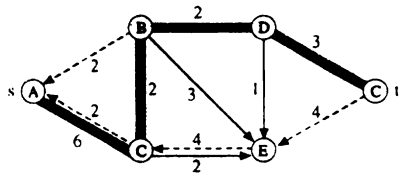


Figura 14.6 Esempio di cammino aumentante nella rete residua di Figura 14.4 (a).

14.2.1 Versione ricorsiva del metodo delle reti residue

In questo paragrafo descriviamo una elegante variante ricorsiva del del metodo delle reti residue, mostrata in Figura 14.5. La correttezza di questa versione si basa sul seguente lemma, la cui dimostrazione è lasciata per esercizio al lettore (vedi Problema 14.8).

Lemma 14.2 (Relazione tra flussi massimi in G e in G_f) *Se f è un flusso in G e f_f^* è un flusso massimo in G_f allora $f + f_f^*$ è un flusso massimo in G.*

In base a questo lemma, se siamo in grado di trovare un flusso f non nullo in G (riga 1), allora calcolando ricorsivamente un flusso massimo $\hat{f}$ in G_f saremo in grado di ottenere un flusso massimo nella rete G di partenza come somma $f^* = f + \hat{f}$ (riga 4). Si noti che, se f non è nullo, abbiamo $|\hat{f}| < |f^*|$, e quindi calcolare $\hat{f}$ significa di fatto ridursi a un problema strettamente più "piccolo" di quello iniziale che chiedeva di calcolare f^*. Il passo base della ricorsione (riga 2) si raggiunge quando non siamo più in grado di trovare un flusso f non nullo in G.

14.3 Metodo dei cammini aumentanti

Il *metodo dei cammini aumentanti*, noto già nel 1956 dai primi lavori sul flusso di Lestor Ford e Ray Fulkerson [6], è una particolare variante del metodo generale iterativo delle reti residue descritto nel Paragrafo 14.2. Per calcolare un flusso non nullo f' in G_f, questo metodo identifica un cammino π tra s e t e sceglie f' in modo che $f'(x,y) > 0$ per ogni arco $(x,y) \in \pi$ e $f'(x,y) = 0$ altrimenti. Un flusso di questo tipo si chiama *flusso su un cammino*. Poichè come abbiamo visto nel Paragrafo 14.2 calcolare f' in G_f ci permette di aumentare f, un cammino che porta da s a t in G_f viene chiamato *cammino aumentante* per f.

Esempio 14.3 Un esempio di cammino aumentante $\pi = \langle A, C, B, D, F \rangle$ nella rete residua di Figura 14.4 (a) è mostrato in Figura 14.6. □

Come vedremo nel Paragrafo 14.4 e nel Paragrafo 14.5, la particolare scelta del cammino aumentante π ci permetterà di specializzare ulteriormente il metodo ottenendo algoritmi con prestazioni diverse. Una volta scelto un cammino, dobbiamo capire quanto flusso può attraversarlo. Ovviamente, poiché il nostro scopo finale è quello di calcolare un flusso massimo, sembra una buona idea mandare lungo quel cammino la massima quantità di flusso possibile. Questa quantià è limitata dalla *capacità del cammino*.

Definizione 14.8 (Capacità di un cammino) *Dato un cammino π in una rete G, la capacità del cammino $c(\pi)$ è la capacità del suo arco con capacità più piccola, cioè:*

$$c(\pi) = \min_{(u,v)\in\pi} \{ c(u,v) \}$$

Poichè nel nostro contesto abbiamo a che fare con cammini in G_f, useremo r piuttosto che c nel determinare la capacità di un cammino aumentante.

Esempio 14.4 La capacità $r(\pi)$ del cammino aumentante $\pi = \langle A, C, B, D, F \rangle$ nella rete residua di Figura 14.6 è 2, ed è pari alla capacità degli archi (C, B) e (B, D) che rappresentano il "collo di bottiglia" sul cammino. □

Il metodo dei cammini aumentanti può essere riassunto nel modo seguente. Si parte da un flusso $f = f_0$ e si esegue ripetutamente finché possibile il seguente passo aumentante:

(PASSO AUMENTANTE) *Troviamo un cammino aumentante π per il flusso corrente f, se esiste, definiamo $f'(u,v) = r(\pi)$ per ogni arco $(u,v) \in \pi$ (e $f'(u,v) = 0$ altrimenti), e sommiamo infine f' a f.*

Intuitivamente, questo corrisponde a incrementare di $r(\pi)$ il flusso $f(u,v)$ di ogni arco $(u,v) \in \pi$. Si noti che poiché $f'(y,x) > 0$ e per antisimmetria $f'(x,y) = -f'(y,x) < 0$, se il cammino aumentante contiene un arco tratteggiato (y,x), allora sommare algebricamente f' a f significa di fatto annullare una certa quantità di flusso precedentemente inviata da f su (x,y). In pratica è come "spingere" all'indietro del flusso che prima passava per (x,y) in G: questo ci consente di reinstradarlo in una direzione diversa che ci permette di incrementare il flusso totale.

Esempio 14.5 Consideriamo nuovamente il cammino aumentante $\pi = \langle A, C, B, D, F \rangle$ nella rete residua G_f di Figura 14.6. Poiché come osservato $r(\pi) = 2$, allora possiamo inviare 2 unità di flusso in G_f lungo π. Si noti che π contiene un arco tratteggiato (C, B) che corrisponde a un arco in avanti (B, C) nella rete originale G. Sommare quindi $f'(B,C) = -2$ a $f(B,C) = 2$ significa di fatto "rimangiarsi" le due unità di flusso precedentemente assegnate da f all'arco (B, C). □

Nei paragrafi successivi, vedremo due diverse strategie per scegliere il cammino aumentante, che daranno luogo ad altrettanti algoritmi per il massimo flusso con prestazioni diverse.

14.4 Algoritmo di Ford e Fulkerson

In Figura 14.7 illustriamo un primo esempio di algoritmo basato sul metodo dei cammini aumentanti di Ford e Fulkerson descritto nel Paragrafo 14.3. Questo algoritmo, proposto dagli stessi Ford e Fulkerson come applicazione diretta del loro metodo [6], si basa su una scelta arbitraria del cammino aumentante π, che viene calcolato ad ogni passo con una qualunque visita della rete residua che cerca il pozzo t a partire dalla sorgente s. L'algoritmo termina quando non è più possibile raggiungere il pozzo t nella rete residua.

```
    algoritmo FordFulkerson(rete G) → flusso
1.    f ← f0
2.    repeat
3.        calcola la rete residua G_f
4.        cerca un cammino aumentante π in G_f
5.        if ( G_f non ha un cammino aumentante ) then break
6.        incrementa f di r(π) lungo π
7.    return f
```

Figura 14.7 Algoritmo di Ford e Fulkerson.

Quanto tempo impiega l'algoritmo di Ford e Fulkerson per calcolare un flusso massimo in una rete? Se assumiamo che le capacità degli archi della rete sono intere, possiamo dimostrare il seguente teorema:

Teorema 14.1 *Se le capacità della rete sono intere, l'algoritmo di Ford e Fulkerson richiede tempo $O(m|f^*|)$ nel caso peggiore.*

Dimostrazione. L'algoritmo parte con un flusso nullo e termina quando non c'è più nessun cammino aamentante tra s e t nella rete residua. Se le capacità degli archi sono intere, allora l'algoritmo incrementerà il valore $|f|$ del flusso corrente di almeno una unità ad ogni iterazione; infatti, G_f contiene solo gli archi (u,v) con capacità residua positiva, cioè $r(u,v) = c(u,v) - f(u,v) > 0$. Quindi, si possono avere al più $|f^*|$ iterazioni. Inoltre, ad ogni iterazione si deve effettuare una visita della rete residua che richiede di esplorarne tutti gli archi; poiché essa ha al più il doppio degli archi della rete iniziale, questo richiede al più tempo $O(m)$. Quindi, nel caso peggiore il tempo di esecuzione dell'intero algoritmo sarà limitato da $O(m|f^*|)$, dove $|f^*|$ è il valore del massimo flusso calcolato. □

Facciamo notare che esistono esempi per cui questo tempo di esecuzione è effettivamente richiesto. Come abbiamo già osservato nel Puragrafo 14.2 discutendo il metodo delle reti residue, la situazione nel caso in cui le capacità sono valori reali potrebbe essere addirittura disastrosa: vi sono infatti esempi di reti con capacità reali su cui l'algoritmo di Ford e Fulkerson potrebbe richiedere tempo infinito!

14.5 Algoritmo di Edmonds e Karp

Jack Edmonds e Richard Karp fecero una osservazione molto semplice [4]: poiché il numero di iterazioni dell'algoritmo di Ford e Fulkerson dipende dall'entità dell'incremento di flusso ad ogni passo, perché non tentare di usare ad ogni iterazione il cammino aumentante con la più alta capacità nella rete residua? Usando questa idea, progettarono una variante dell'algoritmo di Ford e Fulkerson, che riportiamo in Figura 14.8. Si noti che l'unica differenza fra i due algoritmi è alla linea 4: invece di selezionare un cammino π arbitrario, Edmonds e Karp calcolano un cammino π con il più alto valore di $r(\pi)$. Come ci accingiamo a vedere, in caso di capacità intere, questa semplice modifica porta ad un sostanziale miglioramento del tempo di esecuzione.

```
   algoritmo EdmondsKarp(rete G) → flusso
1.   f ← f0
2.   repeat
3.     calcola la rete residua G_f
4.     cerca un cammino aumentante π con la più alta capacità residua r(π) in G_f
5.     if ( G_f non ha un cammino aumentante ) then break
6.     incrementa f di r(π) lungo π
7.   return f
```

Figura 14.8 Algoritmo di Edmonds e Karp.

Per determinare il numero massimo di iterazioni effettuate dall'algoritmo, studiamo dapprima una importante proprietà generale dei flussi nelle reti, nota come proprietà di decomposizione.

Lemma 14.3 (Decomposizione) *Qualunque flusso in G può essere decomposto nella somma di un flusso nullo e di al più m flussi su cammini in G.*

Dimostrazione. Possiamo innanzitutto assumere che il flusso su ciascun arco sia uguale alla sua capacità: infatti, se così non fosse, possiamo sempre riportarci a questo caso riducendo artificialmente le capacità degli archi senza modificare in alcun modo il flusso. Di conseguenza, le capacità degli archi che non portano flusso possono essere ridotte a zero, e gli archi stessi rimossi dalla rete. L'utilità di questo aspetto sarà chiarito a breve.

Sia f un flusso in G. Se $|f| = 0$, l'enunciato del lemma è verificato. Altrimenti, possiamo costruire una sequenza di flussi su cammini con il procedimento seguente. Sia π un cammino da s a t nella rete G e sia $c(\pi) \geq 0$ la capacità che rappresenta il collo di bottiglia di questo cammino (cioè la capacità più piccola fra quella degli archi del cammino). Sia inoltre f_π il flusso sul cammino π tale che $f_\pi(u,v) = c(\pi)$ per ogni arco (u,v) in π, e zero altrove. Aggiungiamo f_π alla nostra sequenza ed effettuiamo gli aggiornamenti $f \leftarrow f - f_\pi$ e $c \leftarrow c - f_\pi$. Dopo questa operazione, l'arco che rappresentava il collo di bottiglia su π viene

ad avere capacità zero, e quindi può essere rimosso dalla rete. Ripetiamo questo procedimento, aggiungendo ad ogni passo un nuovo flusso su cammino alla nostra sequenza fino a quando non arriviamo ad un flusso nullo. Poiché ad ogni passo rimuoviamo almeno un arco dalla rete, avremo collezionato al più m flussi su cammini. □

Consideriamo il flusso corrente f ad una certa iterazione dell'algoritmo di Edmonds e Karp, e sia f_f^* il valore (ancora ignoto) del massimo flusso nella rete residua G_f. In base al Lemma 14.2, sappiamo che $|f_f^*| = |f^*| - |f|$, dove f^* è il massimo flusso nella rete iniziale G. Per il Lemma 14.3 (lemma di decomposizione), f_f^* può essere immaginato come somma di m flussi su cammini, e quindi il cammino π selezionato dall'algoritmo a quell'iterazione avrà capacità residua $r(\pi)$ almeno $|f_f^*|/m$. Inviando $r(\pi)$ unità addizionali di flusso lungo π, $|f_f^*|$ decrescerà di almeno $|f_f^*|/m$ unità (e $|f|$ crescerà della stessa quantità). Se assumiamo per un momento che m sia una costante, il flusso massimo f_f^* nella rete residua G_f si riduce di una frazione costante ad ogni iterazione: intuitivamente, questo significa che in un numero logaritmico di passi $|f_f^*|$ diventa zero. Come vedremo nel sequente teorema, l'algoritmo di Edmonds e Karp non farà mai più di $O(m \log |f^*|)$ iterazioni, contro le $O(|f^*|)$ iterazioni dell'algoritmo di Ford e Fulkerson.

Teorema 14.2 *Se le capacità della rete sono intere, l'algoritmo di Edmonds e Karp richiede tempo $O((m + n \log n) \cdot m \log |f^*|)$ nel caso peggiore.*

Dimostrazione. Dimostriamo innanzitutto formalmente che l'algoritmo di Edmonds e Karp termina in al più $m \ln |f^*|$ iterazioni. Sia $|f^*|$ il valore del massimo flusso in G e siano $f_0, \ldots, f_k$ i flussi correnti ad ogni iterazione dell'algoritmo, con $|f_0| = 0$ e $|f_k| = |f^*|$. Siano inoltre $f_0^*, \ldots, f_k^*$ i flussi massimi nelle reti residue $G_{f_0}, \ldots, G_{f_k}$ ad ogni iterazione (ignoti all'algoritmo), dove $G_{f_0} = G$. In base al Lemma 14.2, si ha $|f_i| + |f_i^*| = |f^*|$ per ogni i, $0 \leq i \leq k$, e quindi $|f_0^*| = |f^*|$. Inoltre, ad ogni iterazione i dell'algoritmo, in base al Lemma di Decomposizione applicato a G_{f_i} instradiamo almeno $|f_i^*|/m$ unità di flusso da s a t, e quindi $|f_{i+1}^*| = |f_i^*| - |f_i^*|/m$. Riassumendo, possiamo quindi scrivere:

$$\begin{cases} |f_0^*| = |f^*| \\ |f_{i+1}^*| \leq |f_i^*| \left(1 - \frac{1}{m}\right) \end{cases}$$

Mediante sostituzioni successive, otteniamo:

$$|f_k^*| \leq |f^*| \left(1 - \frac{1}{m}\right)^k .$$

Siccome $|f_k^*| = 0$, vogliamo ora individuare dei valori di k per cui la quantità $|f^*| \left(1 - \frac{1}{m}\right)^k$ diventa ≤ 1. Usando la ben nota relazione

$$\left(1 + \frac{a}{n}\right)^n < e^a$$

derivata dal limite notevole dell'Equazione 17.5 riportato in Appendice, possiamo riformulare la relazione some segue:

$$|f^*|\left(1-\frac{1}{m}\right)^k = |f^*|\left(1+\frac{-1}{m}\right)^{m\frac{k}{m}} < |f^*|\cdot e^{-\frac{k}{m}} \leq 1.$$

Risolvendo $|f^*|\cdot e^{-\frac{k}{m}} \leq 1$ rispetto a k, otteniamo $k \geq m \ln |f^*|$. Questo significa che bastano $k = m \ln |f^*|$ iterazioni per raggiungere il massimo flusso. In base all'Equazione 17.1 per il cambio di base dei logaritmi riportata in Appendice, questo è $O(m \log |f^*|)$.

Poiché ogni iterazione richiede tempo $O(m+n\log n)$ per cercare il cammino π (si veda il Problema 14.3) e tempo $O(m)$ per aggiornare il flusso e la rete residua correnti, il tempo di esecuzione complessivo dell'algoritmo nel caso peggiore sarà limitato da $O((m+n\log n)\cdot m\log|f^*|)$. □

14.6 Flusso massimo e taglio minimo

In questo paragrafo, mostreremo che il problema di calcolare la quantità massima di flusso in una rete è equivalente a quello di calcolare la capacità del taglio con capacità minima. In modo simile a quanto fatto nel Paragrafo 12.3 del Capitolo 12, definiamo un *taglio* in una rete nel modo seguente.

Definizione 14.9 (Taglio) *Un* taglio (s,t) *in una rete G è una partizione di V in due insiemi X e $\overline{X}$ tali che $s \in X$ e $t \in \overline{X}$. La capacità del taglio è data da:*

$$c(X,\overline{X}) = \sum_{\substack{u\in X\\ v\in\overline{X}}} c(u,v)$$

Esempio 14.6 Sia $X = \{A,B,C,E\}$ e $\overline{X} = \{D,F\}$ un taglio nella rete di flusso di Figura 14.1. Poichè gli unici archi che portano da un vertice in X a un vertice in $\overline{X}$ sono (B,D) e (E,F) si ha $c(X,\overline{X}) = c(B,D)+c(E,F) = 6$. □

È immediato vedere che la quantità di flusso che attraversa ogni taglio di una rete è uguale alla quantità di flusso che transita da s a t.

Lemma 14.4 *Dato un flusso f e un taglio $(X,\overline{X})$, la quantità di flusso $f(X,\overline{X})$ che attraversa il taglio è uguale a $|f|$.*

Dimostrazione.

$$f(X,\overline{X}) = \sum_{\substack{u\in X\\ v\in\overline{X}}} f(u,v) = \sum_{\substack{u\in X\\ v\in V}} f(u,v) - \sum_{\substack{u\in X\\ v\in X}} f(u,v) =$$

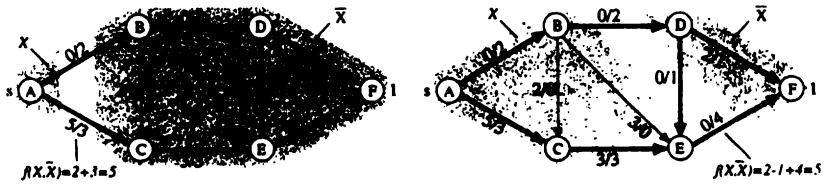


Figura 14.9 Due tagli nella rete di Figura 14.2 (c).

$$\sum_{\substack{u\in X-\{s\}\\ v\in V}} f(u,v) + \sum_{v} f(s,v) - \sum_{\substack{u\in X\\ v\in X}} f(u,v) = |f|$$

poiché $\sum_{u\in X-\{s\},v} f(u,v) = 0$ in base al vincolo di conservazione del flusso e $\sum_{u\in X, v\in X} f(u,v) = 0$ per antisimmetria. □

Esempio 14.7 In Figura 14.9 mostriamo due tagli diversi nella rete di Figura 14.4 (a). Si noti che i due tagli sono attraversati dallo stesso flusso pari a $|f| = 5$, come stabilito dal Lemma 14.4. □

In base al vincolo di capacità, il flusso che attraversa un taglio non può eccedere la capacità del taglio. Quindi, la capacità del minimo taglio (s,t) è un limite superiore al valore del massimo flusso. Il teorema *massimo-flusso/minimo-taglio* afferma che questi due valori in effetti coincidono.

Teorema 14.3 (Teorema del massimo flusso – minimo taglio) *Le seguenti tre affermazioni sono equivalenti:*

(1) f è un flusso massimo;
(2) non c'è nessun cammino aumentante per f;
(3) esiste un taglio $(X,\overline{X})$ tale che $c(X,\overline{X}) = |f|$.

Dimostrazione. Dimostriamo in modo circolare che (1) ⇒ (2), poi (2) ⇒ (3), e infine (3) ⇒ (1).

(1) ⇒ (2): Se esistesse un cammino aumentante π per f, allora potremmo incrementare il flusso lungo π, e quindi f non sarebbe massimo.

(2) ⇒ (3): Consideriamo la rete residua G_f. Poiché non c'è nessun cammino aumentante per f, non c'è nessun cammino diretto da s a t in G_f. Sia X l'insieme dei vertici raggiungibili da s in G_f e sia $\overline{X} = V - X$. Osserviamo che $s \in X$ e $t \in \overline{X}$, e quindi $(X,\overline{X})$ è un taglio $\{s,t\}$. In base al Lemma 14.4:

$$|f| = \sum_{\substack{u\in X\\ v\in\overline{X}}} f(u,v).$$

Poiché $\overline{X}$ non è raggiungibile da s in G_f, $u \in X$ e $v \in \overline{X}$ implica che (u, v) non è un arco di G_f, e cioè, $f(u, v) = c(u, v)$. Possiamo così concludere che:

$$|f| = \sum_{\substack{u \in X \\ v \in \overline{X}}} f(u, v) = \sum_{\substack{u \in X \\ v \in \overline{X}}} c(u, v) = c(X, \overline{X}).$$

(3) $\Rightarrow$ (1): Poiché un qualsiasi flusso f e un qualsiasi taglio $(X, \overline{X})$ soddisfano la relazione $|f| \leq c(X, \overline{X})$, il flusso che soddisfa $|f| = c(X, \overline{X})$ deve essere massimo (e $(X, \overline{X})$ un taglio minimo). □

Il teorema del massimo flusso–minimo taglio stabilisce un nesso profondo tra due problemi apparentemente molto diversi. È però importante notare che l'equivalenza riguarda soltanto la massima quantità di flusso e non l'effettivo assegnamento di flusso agli archi della rete. In generale, conoscere un minimo taglio non è di alcun aiuto nella determinazione di un massimo assegnamento di flusso (vedi Problema 14.6).

14.7 Problemi

Problema 14.1 (Esistenza di flussi massimi diversi) Dare un esempio di rete di flusso per cui esistono due flussi massimi diversi.

Problema 14.2 Si dimostri che, dato un flusso f in una rete $G = (V, E, s, t, c)$, la somma dei flussi uscenti dalla sorgente s è uguale alla somma dei flussi entranti nel pozzo t, ovvero:

$$|f| = \sum_{(s,v) \in E} f(s, v) = \sum_{(v,t) \in E} f(v, t).$$

Problema 14.3 (Cammino aumentante con capacità massima) Si progetti un algoritmo che, data una rete di flusso G, trovi un cammino π dalla sorgente al pozzo con la più alta capacità $c(\pi)$. Se la rete G contiene n vertici ed m archi, l'algoritmo deve richiedere al più tempo $O(m + n \log n)$ nel caso peggiore.

Problema 14.4 (Sorgenti multiple) Supponiamo di dover spedire flusso in una rete da più sorgenti verso il pozzo. Come possiamo modificare la struttura della rete in modo da poterci ricondurre al caso in cui abbiamo una sorgente sola trovando un assegnamento di flusso che risolva il problema originario con più sorgenti?

Problema 14.5 (Flusso in reti non orientate) Considerare la rete di flusso di Figura 14.1. Assumendo di non considerare l'orientazione degli archi, cioè ammettendo di poter attraversare ogni arco in qualsiasi direzione, quanto sarebbe il valore del flusso massimo? Si ricordi che è sempre possibile ricondurre un grafo non orientato G a un grafo orientato G' rimpiazzando ogni arco $(x, y) \in G$ con una coppia di archi orientati (x, y) e (y, x) in G'. Mostrare inoltre la rete residua G'_f di G' corrispondente al flusso massimo trovato.

Problema 14.6 (Massimo flusso e minimo taglio) Utilizzando il Teorema 14.3, che lega flussi massimi con tagli minimi in una rete, presentare un algoritmo in grado di trovare un taglio minimo in tempo $O(m)$ a partire da un flusso massimo. Se al contrario un taglio minimo fosse dato, saremmo in grado di usare questa informazione per trovare un massimo flusso più efficientemente che applicare il migliore algoritmo noto per il massimo flusso?

Problema 14.7 (Relazione tra flussi in G e in G_f) Dimostrare che, se f e $f + f'$ sono flussi in G, allora f' è un flusso in G_f.

Problema 14.8 (Relazione tra flussi massimi in G e in G_f) Dimostrare che, se f è un flusso in G e f^*_f è un flusso massimo in G_f allora $f + f^*_f$ è un flusso massimo in G.

14.8 Sommario

In questo capitolo, abbiamo studiato il problema di spedire la massima quantità di flusso da una sorgente a un pozzo in una rete senza eccedere le capacità degli archi. Questo problema è di grande rilevanza applicativa e modella ad esempio il flusso di pacchetti di dati in una rete di comunicazione, il flusso di automobili in una rete stradale, il flusso di molecole di liquido in una rete di distribuzione idrica, il flusso di corrente in una rete elettrica, e così via.

Il problema dei cammini minimi visto nel Capitolo 13 e quello del flusso massimo sono in un certo modo complementari: l'uno trova cammini che minimizzano il costo di percorrenza senza preoccuparsi delle capacità degli archi, l'altro massimizza la quantità di flusso trasferito senza eccedere le capacità degli archi, ma non si preoccupa del costo di percorrenza.

Abbiamo descritto un metodo generale per calcolare un massimo flusso chiamato *metodo delle reti residue*. Nella sua versione iterativa, il metodo è incrementale: si parte da un flusso nullo e lo si aumenta progressivamente finché non diventa un flusso massimo. L'idea è che, se abbiamo una rete G di cui conosciamo un flusso f (in generale non massimo), possiamo costruire una *rete residua* G_f derivata da G tale che, se f' è un flusso in G_f, allora $f + f'$ è un flusso in G. Trovando allora un flusso f' in G_f, sommando f' a f otteniamo un flusso maggiore. Tenendo aggiornata la rete residua G_f rispetto a f, l'applicazione ripetuta di questo passo permette in generale di arrivare a un flusso massimo.

Abbiamo poi mostrato come specializzare il metodo delle reti residue calcolando f' come flusso su un cammino da s a t. Questa specializzazione del metodo generale, nota come *metodo dei cammini aumentanti* di Ford e Fulkerson, ci ha permesso di ottenere due algoritmi per il calcolo del massimo flusso. Se il cammino aumentante è scelto in modo arbitrario, Ford e Fulkerson hanno dimostrato che si ottiene un massimo flusso in tempo $O(m \cdot |f^*|)$ nel caso peggiore, purchè le capacità degli archi della rete sono intere. Nel caso generale di capacià reali, il metodo potrebbe però addirittura non convergere. Scegliendo come cammino aumentante un cammino da s a t con la più alta capacità in G_f, si ottiene un tempo

di esecuzione $O((m + n \log n) \cdot m \log |f^*|)$ nel caso peggiore. Questo secondo algoritmo è dovuto a Edmonds e Karp.

Come ultimo argomento del capitolo, abbiamo mostrato che il problema di calcolare la quantità massima di flusso in una rete è equivalente a quello di calcolare la capacità del taglio con capacità minima. Questo risultato ci fornisce un esempio interessante dell'esistenza di nessi profondi tra problemi apparentemente molto diversi.

14.9 Note bibliografiche

Come per il problema dei cammini minimi studiato nel Capitolo 13, anche per problemi di flusso nelle reti esiste una letteratura vastissima, con risultati pubblicati già nei primi anni '50. Le tecniche sviluppate nell'ultimo mezzo secolo sono molto raffinate, e vanno ben oltre quelle che abbiamo studiato in questo capitolo.

Il primo algoritmo per il calcolo del flusso massimo fu proposto nel 1951 da George Dantzig [2]. Esso richiedeva tempo $O(n^2 \cdot m \cdot U)$, dove n ed m sono il numero di vertici e archi della rete, rispettivamente, e U è la massima capacità di un arco. La maggior parte dei concetti studiati in questo capitolo, come quello di rete residua e di cammino aumentante sono dovuti al lavoro pionieristico di Lestor Ford e Ray Fulkerson intorno alla metà degli anni '50 [6]. Gli algoritmi proposti in quel decennio sono pseudopolinomiali: il loro tempo di esecuzione è infatti un polinomio che coinvolge la massima capacità U degli archi, che come abbiamo visto può essere esponenziale nel numero di vertici della rete.

Nei primi anni '70, E.A. Dinitz [3] e J. Edmonds e R. Karp [4] scoprirono che inviare flusso lungo cammini aumentanti di lunghezza minima permette di eliminare la dipendenza da U, ottenendo i primi algoritmi polinomiali. In particolare, Dinitz propose un semplice algoritmo che richiede tempo $O(n^2 \cdot m)$. Come per altri problemi algoritmici classici come minimo albero ricoprente e cammini minimi che abbiamo discusso nei capitoli 12 e 13, anche nel caso del massimo flusso l'uso di strutture dati opportune si è rivelato cruciale per ottenere algoritmi efficienti. Un esempio di questo tipo è l'algoritmo di Zvi Galil e Amnon Naamad del 1980 [7], che richiede tempo $O(mn \cdot \log^2 n)$ usando particolari strutture dati per memorizzare informazioni su cammini.

La maggior parte degli algoritmi per reti con pesi reali progettati negli ultimi vent'anni hanno prestazioni che sono entro un fattore polilogaritmico da $O(mn)$ [8]. Ricordiamo fra questi il metodo di preflow-push sviluppato da Andrew Goldberg e Robert Tarjan [10], che permette di ottenere implementazioni molto efficienti in pratica [11]. È tutt'ora un problema aperto di fondamentale importanza capire se si possa infrangere la barriera $O(mn)$ nel caso generale di reti con capacità reali.

Come mostrato da Shimon Even e Robert Tarjan nel 1975 [5], nel caso in cui le capacità degli archi sono unitarie, l'algoritmo di Dinitz, che invia flusso su cammini minimi, richiede tempo $O(\min\{n^{2/3}, m^{1/2}\} \cdot m)$. Quasi trent'anni dopo, Andrew Goldberg e Satish Rao hanno mostrato come estendere questo risultato al caso di capacità intere, ottenendo un tempo di esecuzione $O(\min\{n^{2/3}, m^{1/2}\} \cdot m \cdot \log(n^2/m) \cdot \log U)$ [9]. Se sia possibile fare di meglio per reti con capacità unitarie o intere è un altro importante problema aperto.

Fra tutte le varianti del problema del massimo flusso, il caso di reti non orientate con capacità unitarie è quello che si sa risolvere nel modo più efficiente. Il migliore algoritmo noto è dovuto a David Karger e Matthew Levine [12] e richiede tempo $O(n^{7/6}m^{2/3})$.

Un'ottima trattazione di problemi di flusso appare nel testo di Ravindra Ahuja, Thomas Magnanti e James Orlin [1]. Una interessante panoramica dei risultati noti e dei problemi aperti nel calcolo del massimo flusso è riportata in una rassegna pubblicata nel 1998 da Andrew Goldberg [8].

Riferimenti bibliografici

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] G. B. Dantzig. *Activity Analysis of Production and Allocation*, chapter Programming of Interdependent Activities, II, Mathematical Models, 19–32. John Wiley and Sons Inc., New York, 1951.

[3] E. A. Dinitz. "Metod porazryadnogo sokrashcheniya nevyazok i transportnye zadachi". In *Issledovaniya po Diskretnoi Matematike*, Nauka, Moskva, 1973. In Russian. Title translation: Excess Scaling and Transportation Problems.

[4] J. Edmonds and R. M. Karp. "Theoretical Improvements in the Algorithmic Efficiency for Network Flow Problems", *Journal of the Association for Computing Machinery*, 19:248–264, 1972.

[5] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4:507–518, 1975.

[6] L. R. Ford, Jr., and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[7] Z. Galil and A. Naamad. "An $O(E \cdot V \cdot \log^2 V)$ Algorithm for the Maximal Flow Problem", *Journal of Computer and System Sciences*, 21:203–217, 1980.

[8] A. V. Goldberg. "Recent Developments in Maximum Flow Algorithms (Invited Lecture)", In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, 1–10, 1998.

[9] A. V. Goldberg and S. Rao. "Beyond the flow decomposition barrier", *Journal of the Association for Computing Machinery*, 45(5), 1998.

[10] A. V. Goldberg and R. E. Tarjan. "A new approach to the maximum flow problem", *Journal of the Association for Computing Machinery*, 35:921-940, 1988.

[11] B. V. Cherkassky and A. V. Goldberg. "On implementing the push-relabel method for the maximum flow problem", *Algorithmica*, 19:390–410, 1997.

[12] D. R. Karger and M. S. Levine. "Finding maximum flows in undirected graphs seems easier than bipartite matching", In *Proceedings of the 30th annual ACM symposium on Theory of computing*, 69–78, 1998.

15

Algoritmi geometrici

La filosofia è scritta nel grandissimo libro che è l'universo, in una lingua matematica, i cui caratteri sono triangoli, cerchi, ed altre figure geometriche. Senza queste è un aggirarsi per un oscuro labirinto.

(Galileo Galilei)

La geometria computazionale studia il progetto e l'analisi di algoritmi e strutture dati per risolvere problemi geometrici. Gli oggetti presi in considerazione sono tipicamente insiemi di punti nello spazio Euclideo e oggetti ottenibili dalla loro combinazione, quali linee, segmenti, poligoni. I tipici problemi affrontati richiedono di rispondere a interrogazioni sugli oggetti (ad esempio, due rettangoli si intersecano?) o di calcolare un altro oggetto geometrico con particolari proprietà (quale l'inviluppo convesso che definiremo a breve). Problemi di geometria computazionale trovano applicazione in innumerevoli aree, che includono la grafica, la progettazione di circuiti VLSI, l'ottimizzazione di reti, la robotica, le basi di dati, e per questo motivo sono stati ampiamente studiati. In questo capitolo, per offrire una panoramica delle tematiche affrontate in quest'area, esamineremo soprattutto i seguenti quattro problemi:

- *Inviluppo convesso*: dato un insieme di punti, l'inviluppo convesso è il più piccolo poligono convesso tale che ogni punto dell'insieme o giace al suo interno o sul suo perimetro. Intuitivamente, possiamo pensare ai punti come a chiodi che sporgono da un piano; immaginiamo anche di avere un elastico che allargheremo in modo tale da includere tutti i chiodi e che poi rilasceremo: l'inviluppo convesso è proprio la forma assunta dall'elastico!
- *Localizzazione di punti*: data una suddivisione planare del piano, ovvero un insieme di punti connessi tramite segmenti che si intersecano solo alle loro estremità, vorremmo rispondere a interrogazioni del tipo: "dato un punto p del piano, quale faccia della suddivisione planare contiene p?".
- *Ricerca multidimensionale*: dato un insieme di punti nel piano, quanti (e quali) punti giacciono all'interno di un dato rettangolo con lati paralleli agli assi? Il problema può naturalmente essere generalizzato a spazi d-dimensionali: ad esempio, per $d = 3$, avremo cubi invece di rettangoli e così via. Questo problema ha applicazioni particolarmente importanti: si pensi che i punti rappresentino un insieme di persone, e ogni dimensione sia un certo attributo della persona (luogo di nascita, data di nascita, titolo di studio). Una struttura dati

per la ricerca multidimensionale, ad esempio, può consentire di rispondere efficientemente a interrogazioni del tipo: quante persone nate a Palermo tra il 1970 e il 1980 hanno conseguito la laurea in Informatica?

- *Intersezione di rettangoli*: dato un insieme di rettangoli con lati paralleli agli assi, il problema è di restituire tutte le coppie di rettangoli che si intersecano. Sebbene ci concentreremo su rettangoli nel piano, potremmo anche pensare a intersezioni tra altri tipi di oggetti geometrici: segmenti, cerchi, poligoni. In genere le proprietà degli oggetti considerati, ad esempio la loro forma, sono molto importanti per trovare soluzioni algoritmiche efficienti. Ad esempio, nell'algoritmo che presenteremo in questo capitolo sfrutteremo pesantemente il fatto che i rettangoli in ingresso hanno lati paralleli agli assi. Il problema diventerebbe molto più difficile se ciò non fosse vero.

Analizzeremo ora ognuno di questi problemi in maggior dettaglio.

15.1 Inviluppo convesso

In questo paragrafo studieremo il problema di calcolare l'inviluppo convesso di un insieme di punti nel piano. Esistono generalizzazioni di questo problema in spazi d-dimensionali, di cui però non parleremo in questa sede.

Definizione 15.1 *L'inviluppo convesso di un insieme di n punti nel piano è il più piccolo poligono convesso che racchiude tutti i punti.*

Rappresenteremo l'inviluppo convesso come un poligono, ovvero elencando ciclicamente i punti che ne fanno parte. Osserviamo che tutti i punti dell'inviluppo convesso appartengono all'insieme, ed il loro numero può variare da un minimo di 3 a un massimo di n. Presenteremo ora vari approcci per il calcolo dell'inviluppo convesso in due dimensioni, ottenendo algoritmi via via più efficienti. Iniziamo con un semplice metodo induttivo che ci farà prendere familiarità con il problema. Per semplicità, non tratteremo casi particolari come, ad esempio, l'esistenza di punti allineati. Trattare casi speciali è tipicamente non banale in problemi geometrici, ma tutti gli algoritmi che presenteremo di seguito possono essere estesi senza particolari difficoltà.

15.1.1 Un approccio "induttivo"

Se i punti nell'insieme fossero soltanto tre, farebbero tutti parte dell'inviluppo convesso. Sembra quindi naturale applicare un ragionamento induttivo, simile a quello che abbiamo usato nel Paragrafo 4.2 del Capitolo 4 per cercare di estendere l'ordinamento da k a $k+1$ elementi. Nel caso dell'inviluppo convesso, l'induzione è sul numero di punti: avendo a disposizione l'inviluppo convesso $\mathcal{I}_k$ di un insieme di k punti, aggiungiamo all'insieme un nuovo punto e modifichiamo $\mathcal{I}_k$ in modo da trasformarlo nell'inviluppo convesso $\mathcal{I}_{k+1}$ del nuovo insieme. Ci aspettiamo che prendere in considerazione un nuovo punto p non debba essere troppo

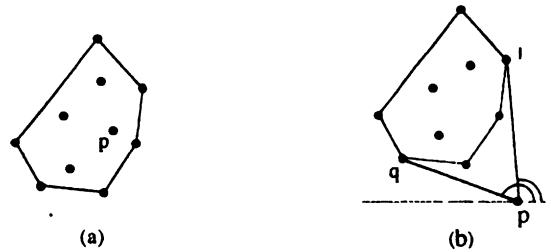


Figura 15.1 Variazione dell'inviluppo convesso $\mathcal{I}_k$ in seguito all'aggiunta del nuovo punto p.

difficile. Esaminiamo quindi quali cambiamenti comporta all'inviluppo convesso $\mathcal{I}_k$ l'aggiunta di p. Come mostrato in Figura 15.1, abbiamo due casi:

- se il punto p è interno a $\mathcal{I}_k$, l'inviluppo convesso non varia;
- altrimenti, p sarà incluso in $\mathcal{I}_{k+1}$, sostituendo una catena di punti che invece faceva parte di $\mathcal{I}_k$.

Dobbiamo quindi essere in grado di verificare se un punto è interno o esterno a un poligono convesso, ed eventualmente identificare i punti di $\mathcal{I}_k$ (ce ne potrebbero essere molti!) che diventano interni. La seguente osservazione ci permette di risparmiare facilmente la prima verifica:

Osservazione 15.1 *Se una delle coordinate del punto p fosse massima o minima rispetto a quella di tutti i primi k punti, sicuramente p apparterrebbe a $\mathcal{I}_{k+1}$.*

Possiamo quindi pensare di ordinare preliminarmente i punti, ad esempio, per coordinata y decrescente, e di esaminarli in quest'ordine per modificare l'inviluppo convesso. Resta ora il problema di identificare i punti che diventano interni. Immaginiamo di tracciare una linea orizzontale passante per il nuovo punto p, come mostrato in Figura 15.1(b). Siano t e q i due punti di $\mathcal{I}_k$ tali che le linee $\overline{pt}$ e $\overline{pq}$ formano rispettivamente un angolo minimo e massimo con la linea orizzontale passante per p. È facile vedere che proprio i punti q e t saranno i punti di raccordo con p nel nuovo inviluppo convesso $\mathcal{I}_{k+1}$. Basterà allora spostarsi da t in senso orario fino a q ed eliminare i punti incontrati. Per usare quest'idea, abbiamo solo bisogno di saper identificare i punti t e q. Osserviamo che il punto t (rispettivamente, q) è tale che, per ogni punto g, percorrendo il segmento $\overline{pt}$ ($\overline{pq}$) e poi il segmento $\overline{tg}$ ($\overline{qg}$) si compie una svolta a sinistra (destra). Ciò può essere verificato in tempo $O(1)$ come segue:

Lemma 15.1 *Dati tre punti p, t e g, percorrendo il segmento $\overline{pt}$ e poi il segmento $\overline{tg}$ si compie una svolta a sinistra se e solo se*

$$\frac{x_g - x_p}{y_g - y_p} < \frac{x_t - x_p}{y_t - y_p}$$

```
algoritmo inviluppoInduttivo(punti p1, p2, ... pn) → insieme di punti
1.  Ordina i punti per coordinata y decrescente
2.  I_n ← {p1, p2, p3}
3.  for k = 4 to n do
4.      l ← linea parallela all'asse x passante per p_k
5.      trova il punto q tale che l'angolo tra p_k q e l è minimo
6.      trova il punto t tale che l'angolo tra p_k t e l è massimo
7.      rimuovi da I_n i punti tra t e q in senso orario
8.      aggiungi p_k a I_n
9.  return I_n
```

Figura 15.2 Approccio induttivo per il calcolo dell'inviluppo convesso.

Dimostrazione. Siano φ_t e φ_g gli angoli formati dalla linea orizzontale passante per p e dai segmenti $\overline{pt}$ e $\overline{pg}$, rispettivamente. Dalla relazione tra le coordinate cartesiane e le coordinate polari di un punto sappiamo che

$$\tan \varphi_t = \frac{y_t - y_p}{x_t - x_p}$$

$$\tan \varphi_g = \frac{y_g - y_p}{x_g - x_p}$$

Percorrendo i segmenti come indicato si effettua quindi una svolta a sinistra se e solo se $\varphi_g > \varphi_t$. Deve pertanto risultare:

$$\frac{y_g - y_p}{x_g - x_p} > \frac{y_t - y_p}{x_t - x_p}$$

da cui segue la disuguaglianza nell'enunciato. □

Lo pseudocodice dell'algoritmo `inviluppoInduttivo` risultante da questo approccio è mostrato in Figura 15.2.

Teorema 15.1 *L'algoritmo* `inviluppoInduttivo` *calcola l'inviluppo convesso di n punti nel piano in tempo $O(n^2)$ nel caso peggiore.*

Dimostrazione. Utilizzando uno degli algoritmi ottimi basati su confronti descritti nel Capitolo 4 è possibile ordinare i punti per coordinata y decrescente in tempo $O(n \log n)$. Per aggiungere il punto $(k+1)$-esimo all'inviluppo $\mathcal{I}_k$ spendiamo tempo $O(k)$ nelle righe da 5 a 8: infatti l'inviluppo convesso corrente potrebbe contenere $\Theta(k)$ punti. Nel caso peggiore, il tempo di esecuzione dell'intero algoritmo `inviluppoInduttivo` è quindi

$$O(n \log n) + \sum_{i=4}^{n} O(k) = O(n^2)$$

utilizzando la serie aritmetica (vedi Paragrafo 17.2 in Appendice). □

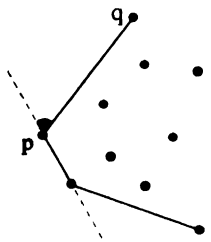


Figura 15.3 Correttezza del metodo di incartamento dei regali: q è il punto tale che l'angolo tra $\overline{pq}$ e ℓ è minimo.

15.1.2 Il metodo dell'incartamento dei regali

Il principale problema dell'algoritmo `inviluppoInduttivo` è che, estendendo l'inviluppo punto per punto, potremmo perdere molto tempo nel costruire inviluppi contenenti punti che poi alla fine saranno eliminati. Potremmo in qualche modo essere certi che, quando aggiungiamo un punto, quel punto farà certamente parte dell'inviluppo convesso finale? Questa è l'idea da cui muove l'approccio che presentiamo in questo paragrafo, che ricorda il modo con cui si incarta un regalo e che quindi chiameremo *algoritmo dell'incartamento dei regali* (in inglese, *gift wrapping*). Questo metodo, che fu originariamente proposto da Jarvis nel 1973 [7], ha il vantaggio di poter essere generalizzato a dimensioni superiori a due, ed in realtà l'analogia con l'impacchettamento dei regali diventa più chiara se lo si pensa in tre dimensioni.

L'algoritmo `marciaDiJarvis` mantiene un sottocammino dell'inviluppo convesso dell'intero insieme di punti e si basa su un approccio induttivo: l'induzione non è però sul numero di punti, bensì sulla lunghezza k del sottocammino individuato. Ad ogni passo, estendiamo il sottocammino con un nuovo segmento, senza mai disfare quello che abbiamo costruito. Il problema è dunque il seguente:

> dato un sottocammino dell'inviluppo finale di lunghezza k, come possiamo estenderlo a un sottocammino di lunghezza $k+1$?

Detto p l'ultimo punto del cammino di lunghezza k, vogliamo trovare un punto q tale che il segmento $\overline{pq}$ appartiene sicuramente all'inviluppo convesso dell'intero insieme di punti. Tracciamo una linea ℓ passante per p e per il punto che precede p nel cammino corrente; per ogni punto t tra gli $(n-k)$ punti rimanenti, calcoliamo l'angolo tra la linea ℓ e il segmento $\overline{pt}$, aggiungendo al cammino il punto q che minimizza tale angolo. La Figura 15.3 illustra perché questo procedimento è corretto: se q non appartenesse all'inviluppo convesso finale, dovrebbe infatti esistere un punto a sinistra della linea $\overline{pq}$, contraddicendo l'ipotesi che q formi l'angolo minimo con ℓ.

Rimane ancora da specificare il passo base: per $k = 1$, il sottocammino conterrà un unico punto, ad esempio quello con coordinata y minima (che sicuramente appartiene all'inviluppo) e la linea ℓ sarà la linea orizzontale passante

```
algoritmo marciaDiJarvis(punti p1, p2, ... pn) → insieme di punti
    sia p il punto con coordinata y minima
    I_n ← {p}
    ℓ ← linea orizzontale passante per p
    R ← insieme di tutti i punti tranne p
    while ( I_n non è completo ) do
        trova il punto q ∈ R tale che l'angolo tra pq̄ e ℓ è minimo
        rimuovi q da R
        aggiungi q a I_n
        ℓ ← pq̄
        p ← q
    return I_n
```

Figura 15.4 Il metodo dell'incartamento dei regali per il calcolo dell'inviluppo convesso.

per tale punto. Lo pseudocodice dell'algoritmo `marciaDiJarvis` è mostrato in Figura 15.4.

Teorema 15.2 *Sia dato un insieme di n punti nel piano il cui inviluppo convesso contiene h punti. L'algoritmo* `marciaDiJarvis` *calcola l'inviluppo convesso dell'insieme in tempo $O(n \cdot h)$ nel caso peggiore.*

Dimostrazione. La correttezza segue dalle argomentazioni che abbiamo discusso precedentemente. Concentriamoci quindi sul tempo di esecuzione. Il ciclo `while` viene eseguito un numero di volte pari al numero di punti nell'inviluppo, ovvero h volte. Poichè la k-esima iterazione richiede tempo $O(n-k)$ per esaminare l'insieme R, il tempo è dato dalla somma

$$\sum_{k=1}^{h} O(n-k) = \sum_{k=1}^{h} O(n) = O(n \cdot h)$$

nel caso peggiore. □

Asintoticamente, nel caso peggiore non abbiamo quindi guadagnato nulla rispetto all'algoritmo `inviluppoInduttivo`, poiché potrebbe essere $h = \Theta(n)$. Come vedremo nel prossimo paragrafo, possiamo però sfruttare le idee di entrambi gli algoritmi per progettare una soluzione più efficiente.

15.1.3 La scansione di Graham

Presenteremo in questo paragrafo un algoritmo che richiede tempo $O(n \log n)$ per trovare l'inviluppo convesso, ed è quindi considerevolmente più veloce di quelli visti nei paragrafi precedenti. L'algoritmo è dovuto a Graham [6] e, ancora una volta, si basa su un approccio induttivo. In modo simile all'algoritmo


```
algoritmo scansioneDiGraham(punti p1, p2, ... pn) → insieme di punti
    p ← punto con coordinata y minima
    l ← linea parallela all'asse x passante per p
    for each punto q ≠ p do calcola l'angolo tra pq̄ e l
    ordina i punti per angolo crescente: siano p1, p2, ... pn i punti ordinati
    for k = 1 to 3 do t_k = p_k
    m ← 3
    for k = 4 to n do
        while ( percorrendo t_{m-1}t_m̄ e poi t_m p_k̄ svoltiamo a sinistra )
        do m ← m − 1
        m ← m + 1
        t_m ← p_k
    return {t1, t2, ... tm}
```

Figura 15.5 La scansione di Graham per il calcolo dell'inviluppo convesso.

`marciaDiJarvis`, l'idea è di mantenere un sottocammino convesso che viene aggiornato ad ogni passo. Rilassiamo però il requisito che il sottocammino appartenga all'inviluppo finale, richiedendo solo che appartenga all'inviluppo convesso dei primi k punti considerati (in particolare, connettendo il primo e l'ultimo punto del sottocammino si otterrà l'inviluppo convesso dei k punti). Da questo punto di vista, l'algoritmo `scansioneDiGraham` è quindi simile all'algoritmo `inviluppoInduttivo`, e potrebbe dover scartare successivamente alcuni dei punti inseriti nella soluzione parziale in un istante precedente. La differenza cruciale è nell'ordine in cui i punti sono considerati. Siano p il punto di coordinata minima e ℓ la linea orizzontale passante per p. Per ogni altro punto q, calcoliamo l'angolo che il segmento $\overline{pq}$ forma con la linea ℓ ed ordiniamo i punti per angolo crescente. L'ipotesi induttiva che l'algoritmo di Graham usa è di conoscere il sottocammino convesso relativo ai primi k punti con angoli minimi. Questa ipotesi permetterà di estendere il sottocammino molto più velocemente di quanto facesse l'algoritmo `inviluppoInduttivo`, come spiegato di seguito.

Chiamiamo $t_1, t_2, \ldots t_m$ il sottocammino convesso relativo ai primi k punti. Sia inoltre p_{k+1} il $(k+1)$-esimo punto nell'ordinamento. Immaginiamo di percorrere il segmento $\overline{t_{m-1}t_m}$ e poi il segmento $\overline{t_m p_{k+1}}$:

- se "svoltiamo" a destra, aggiungiamo p_{k+1} al cammino, che infatti rimane convesso;
- se "svoltiamo" a sinistra, abbiamo invece trovato una concavità. Possiamo allora rimuovere t_m e ripetere il test sui segmenti $\overline{t_{m-2}t_{m-1}}$ e $\overline{t_{m-1}p_{k+1}}$. Continuiamo così finchè non arriviamo a un punto che non va eliminato.

Il test sulla svolta a sinistra o a destra può essere facilmente implementato in tempo $O(1)$, come discusso nel Lemma 15.1. Lo pseudocodice dell'algoritmo `scansioneDiGraham` è dato in Figura 15.5, e il tempo di esecuzione è analizzato nel Teorema 15.3. La Figura 15.6 illustra invece il generico passo della scansione.

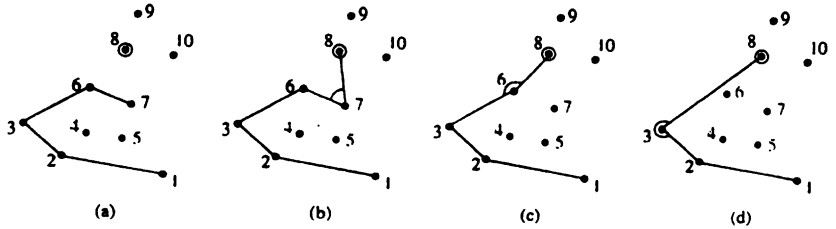


Figura 15.6 Passo generico della scansione di Graham: la numerazione dei punti è stata assegnata per angolo crescente.

Teorema 15.3 *L'algoritmo* `scansioneDiGraham` *calcola l'inviluppo convesso di n punti nel piano in tempo $O(n \log n)$ nel caso peggiore.*

Dimostrazione. Studiamo innanzitutto il tempo di esecuzione delle linee 7–11. Senza contare il tempo speso all'interno del ciclo `while`, queste richiedono tempo $O(n)$. Osserviamo che ogni punto viene aggiunto all'inviluppo convesso una sola volta, in corrispondenza della linea 11. Potrebbe poi essere rimosso in un ciclo `while` eseguito successivamente, ma una volta rimosso non sarà mai più considerato. Quindi, il tempo speso in totale all'interno del ciclo `while` è $O(n)$ e, una volta ordinati i punti, la scansione richiede tempo lineare. Il costo dell'algoritmo è pertanto dominato dall'ordinamento, e quindi è $O(n \log n)$. □

Prima di concludere questo paragrafo, osserviamo che la scansione di Graham utilizza una tecnica algoritmica di vasta applicabilità nota come *backtracking*. Questa tecnica, che è un raffinamento della ricerca esaustiva, permette di ricercare una soluzione per un certo problema esplorando sistematicamente lo spazio di tutte le possibili soluzioni, secondo un approccio "in profondità". Più formalmente, assumiamo che una soluzione sia rappresentata da un vettore $(v_1, ..., v_m)$ di valori, che nel caso della scansione di Graham coincide con l'insieme di punti contenuti nell'inviluppo convesso generato fino a quel momento. Partendo dal vettore vuoto, l'algoritmo ad ogni passo cerca di estendere il vettore parziale corrente con un nuovo valore. Se tale estensione produce una soluzione parziale non ammissibile, come ad esempio nel caso di Figura 15.6 (b), l'algoritmo "torna sui suoi passi" (da cui il nome *backtracking*): l'ultimo valore nel vettore viene rimosso, e si cerca di estendere il vettore con valori alternativi (vedi Figura 15.6 (c)). Nel caso della scansione di Graham, i passi di backtrack e di estensione sono realizzati rispettivamente dalle righe 9 e 10 nello pseudocodice di Figura 15.5.

15.2 Localizzazione di punti in suddivisioni planari

Una *suddivisione planare* del piano è un insieme di n punti connessi tramite segmenti che si intersecano solo alle loro estremità. Punti e segmenti individuano un insieme di regioni del piano racchiuse da segmenti, dette *facce*. Possiamo facil-

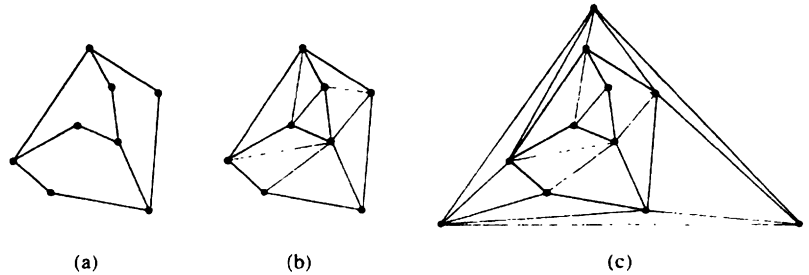


Figura 15.7 (a) Suddivisione planare; (b) triangolazione; (c) triangolazione con faccia esterna triangolare.

mente interpretare una suddivisione come la rappresentazione piana di un grafo planare, i cui vertici sono i punti e gli archi i segmenti che li connettono. Rimandiamo al Paragrafo 17.5 dell'Appendice per definizioni e proprietà basilari dei grafi planari, limitandoci a ricordare che un grafo è planare se può essere disegnato nel piano senza intersezioni di archi.

In questo paragrafo affronteremo il problema di rispondere a interrogazioni di localizzazione di punti in suddivisioni planari (in inglese, *point location*) della seguente forma:

dato un punto p, quale faccia della suddivisione planare contiene p?

Come avviene tipicamente in problemi geometrici, assumeremo che tutte le facce della suddivisione planare siano triangoli: parleremo in tal caso di *triangolazione*. Questa ipotesi non è restrittiva, poiché esiste un algoritmo che, in tempo $O(n)$, trasforma una qualunque suddivisione planare in una triangolazione [2]. La triangolazione prodotta può essere poi estesa in modo che anche la faccia esterna sia essa stessa un triangolo, come mostrato in Figura 15.7. Chiaramente, ogni triangolo appartiene ad una ed una sola faccia della suddivisione originaria, quindi se siamo in grado di restituire il triangolo cui il punto p appartiene, da esso possiamo ottenere la faccia della suddivisione planare in tempo costante. Grazie a tale riduzione, il problema diventa quello di rispondere a interrogazioni di localizzazione di punti in una triangolazione.

Una utile proprietà dei grafi planari è la formula di Eulero dimostrata nel Lemma 17.12 dell'Appendice, che lega i numeri n, m e f di vertici, archi e facce:

$$n - m + f = 2 \tag{15.1}$$

Come mostrato in Appendice, dalla formula di Eulero segue che una triangolazione corrisponde ad un grafo planare massimale, e quindi ha esattamente $3n - 6$ archi e $2n - 4$ facce. Trasformando una suddivisione planare su n punti in una triangolazione, l'occupazione di memoria rimane pertanto lineare in n.

Sia dunque G una triangolazione su n punti. Per rispondere a interrogazioni di localizzazione di punti, ci avvarremo di una struttura dati D che mantiene

una sequenza di triangolazioni su un numero di punti progressivamente inferiore (fino a tre soli punti, quelli sulla faccia esterna). Mostreremo ora come costruire la sequenza di triangolazioni (Paragrafo 15.2.1), come ottenere la struttura dati D (Paragrafo 15.2.2), ed infine come usarla per rispondere alle interrogazioni (Paragrafo 15.2.3).

15.2.1 La sequenza di triangolazioni

Per costruire la sequenza di triangolazioni abbiamo bisogno del concetto di insieme indipedente in un grafo, definito come segue.

Definizione 15.2 *Dato un grafo $G = (V, E)$. un* insieme indipendente *di G è un insieme $S \subseteq V$ di vertici non adiacenti, ovvero tali che. $\forall u, v \in S$, $(u, v) \notin E$. Un insieme indipendente è* massimale *se l'aggiunta di un qualunque altro vertice violerebbe la proprietà di indipendenza.*

Chiameremo $T_1, T_2, ...T_{h(n)}$ la sequenza di triangolazioni, in cui $T_1 = G$ e $T_{h(n)}$ consiste di un solo triangolo i cui estremi sono i tre vertici estemi (eventualmente aggiunti per triangolare G come mostrato in Figura 15.7c). T_{i+1} è ottenuta da T_i usando il seguente algoritmo:

- seleziona un sottoinsieme indipendente massimale V_i di vertici interni di T_i, e rimuovi i vertici e gli archi ad essi incidenti;
- ritriangola i poligoni che si sono formati in seguito alla rimozione dei vertici.

La scelta del sottoinsieme indipendente è di cruciale importanza per le prestazioni dell'algoritmo di interrogazione, poiché determina alcune importanti proprietà della struttura dati di supporto D. In particolare, il criterio di selezione che useremo è il seguente:

rimuovi un insieme indipendente massimale del sottografo indotto dai vertici aventi grado minore di 12 (escludendo i vertici esterni).

A prima vista, la scelta potrebbe sembrare strana: perché 12, e non 5 oppure 90? E perchè limitarsi a considerare vertici di grado limitato? La scelta è giustificata dal seguente lemma.

Lemma 15.2 *Sia T_i una triangolazione con n_i punti e sia n_{i+1} il numero di vertici della suddivisione planare ottenuta rimuovendo un insieme indipendente massimale del sottografo di T_i indotto dai vertici di grado minore di 12. Risulta $n_{i+1} \geq c \cdot n_i$ per una opportuna costante c.*

Dimostrazione. In un grafo planare con n_i vertici, almeno $n_i/2$ vertici hanno grado minore di 12 (vedi Corollario 17.2 in Appendice). Poiché non rimuoveremo mai i tre vertici esterni, che potrebbero far parte di questo insieme di grado basso, l'insieme da cui effettuare la scelta contiene almeno $n_i/2 - 3$ vertici. Per la proprietà di indipendenza, ciascun vertice scelto implica che nessuno dei suoi vicini può appartenere a V_{i+1}: avendo il vertice grado limitato, i vicini scartati saranno al più 11. Essendo l'insieme V_{i+1} massimale, ciò vuol dire che

$$n_{i+1} = |V_{i+1}| \geq \frac{1}{11}\left(\frac{n_i}{2} - 3\right)$$

ovvero la suddivisione planare ottenuta dalla rimozione contiene una frazione costante del numero di punti nella triangolazione T_i. □

Corollario 15.1 *Applicando ripetutamente rimozioni di vertici e ritriangolazioni, si ha $h(n) = O(\log n)$.*

Rimuovere vertici di grado limitato ha anche un'altra utile conseguenza. In base all'algoritmo, dobbiamo infatti ritriangolare i poligoni che si sono formati in seguito alla rimozione dei vertici. Se un vertice rimosso ha grado minore di 12, il poligono che ne risulta ha meno di 12 lati, e quindi possiamo triangolarlo facilmente in tempo $O(1)$. Osserviamo inoltre che, se d è il grado del vertice rimosso, la nuova triangolazione conterrà $(d - 2)$ triangoli.

15.2.2 La struttura dati

Vediamo ora come ottenere, dalla sequenza di triangolazioni, la struttura dati D che useremo nelle interrogazioni. Innanzitutto, D è un grafo diretto aciclico i cui nodi rappresentano triangoli. Per semplicità, come mostrato in Figura 15.8, partizioneremo i nodi in $h(n)$ livelli, a seconda dell'istante in cui il corrispondente triangolo è stato creato. Data una coppia di livelli S_i e S_{i+1}, e dati un triangolo t_i del livello S_i e un triangolo t_{i+1} del livello S_{i+1}, esiste un arco da t_{i+1} a t_i se:

1. il triangolo t_i scompare da S_i in seguito alla rimozione di uno dei suoi estremi;
2. il triangolo t_{i+1} compare in S_{i+1} in seguito alla ritriangolazione;
3. t_{i+1} e t_i hanno intersezione non nulla.

Grazie alle proprietà della sequenza di triangolazioni costruita, ogni cammino in D ha lunghezza $O(\log n)$, essendo logaritmico il numero di livelli in cui i nodi sono partizionati. Inoltre, come dimostrato nel lemma successivo, la struttura dati D ha un'occupazione di memoria lineare in n.

Lemma 15.3 *La struttura dati D ha $O(n)$ nodi ed archi.*

Dimostrazione. Chiamiamo f_i e n_i il numero di triangoli e di punti nella triangolazione T_i. Per la formula di Eulero, $f_i \leq 2n_i$ (vedi Corollario 17.1 in Appendice). Inoltre, abbiamo dimostrato nel Lemma 15.2 che $n_i \leq \alpha \cdot n_{i-1}$ con $\alpha \simeq 1 - 1/22$. Iterando il ragionamento, si ottiene $n_i \leq \alpha \cdot n_{i-1} \leq \alpha^2 \cdot n_{i-2} \leq ... \leq \alpha^{i-1} n_1 = \alpha^{i-1} n$. Il numero totale di nodi in D è pertanto limitato come segue:

$$\sum_{i=1}^{h(n)} f_i \leq 2 \sum_{i=1}^{h(n)} n_i \leq 2n \sum_{i=1}^{h(n)} \alpha^{i-1} \leq 2n \sum_{i=0}^{\infty} \alpha^i \leq \frac{2n}{1-\alpha}$$

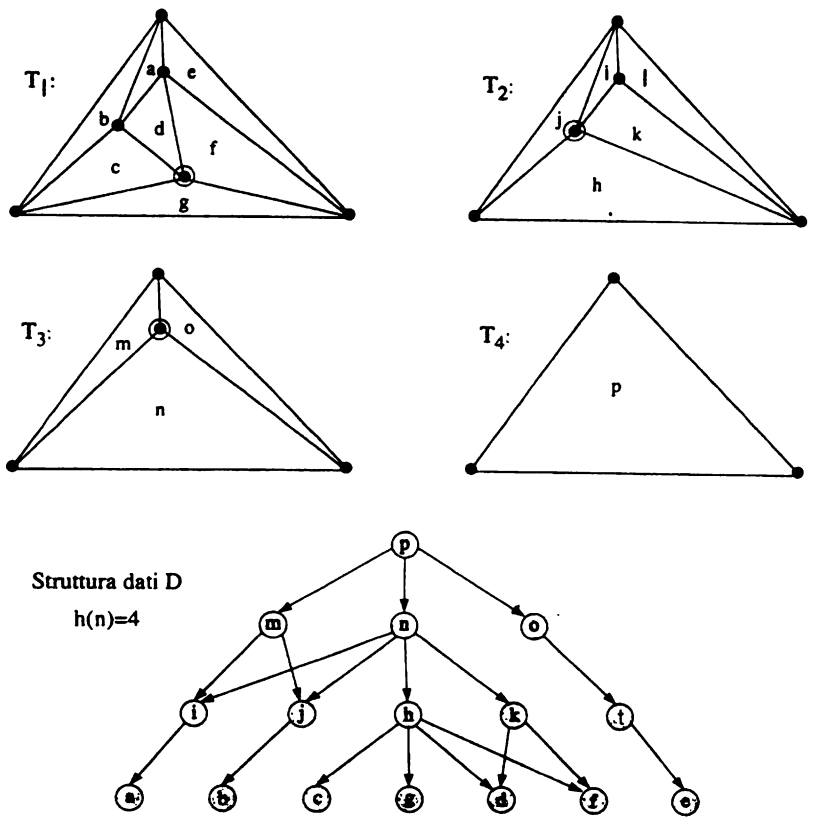


Figura 15.8 Una sequenza di triangolazioni e la corrispondente struttura dati D per la localizzazione di punti.

utilizzando la serie geometrica di ragione α, con $\alpha < 1$. Quindi il numero di nodi è lineare in n. Consideriamo ora il numero di archi. Poiché le nuove triangolazioni dei poligoni create ad ogni passo hanno un numero di triangoli pari al numero di lati del poligono meno 2, e poiché i poligoni hanno meno di 12 lati, il grado uscente di ogni vertice è costante. Anche il numero di archi è pertanto $O(n)$. □

Riassumendo, la struttura dati D ha un'occupazione di memoria $O(n)$, profondità $O(\log n)$, e grado dei vertici costante.

15.2.3 L'algoritmo di interrogazione

Sia p il punto specificato nell'interrogazione. L'algoritmo posiziona p nelle triangolazioni $T_{h(n)}, T_{h(n)-1}, \ldots T_1$, raffinando ad ogni passo la localizzazione. In mag-


```
    algoritmo localizzazioneDiPunti(puntop) → faccia
1.    v ← nodo di D senza archi entranti
2.    if ( p ∉ triangolo(v) ) then return faccia esterna
3.    while ( v ha archi uscenti ) do
4.       for each arco (v, w) do
5.          if ( p ∈ triangolo(w) ) then succ ← w
6.       v ← succ
7.    return triangolo associato al nodo v
```

Figura 15.9 Localizzazione di punti in suddivisioni planari del piano.

gior dettaglio, la struttura dati D viene visitata partendo dall'unico nodo senza archi entranti, corrispondente all'unico triangolo di $T_{h(n)}$. Su un generico nodo v, vengono esaminati tutti gli archi uscenti, identificando l'unico discendente di v che contiene il punto p. La visita prosegue da quel discendente e si ferma su un nodo del livello più basso, restituendo quel nodo in uscita. Lo pseudocodice è mostrato in Figura 15.9.

Teorema 15.4 *Data una suddivisione planare su n punti, possiamo costruire una struttura dati D con un'occupazione di memoria $O(n)$ che consente di rispondere a interrogazioni di localizzazione di punti in tempo $O(\log n)$.*

Dimostrazione. Per l'occupazione di memoria, rimandiamo al Lemma 15.3. Consideriamo ora il tempo per rispondere ad una interrogazione. Osserviamo che possiamo verificare se un punto appartiene a un triangolo in tempo costante. Inoltre, i discendenti di un generico nodo sono al più 11, ed esaminarli tutti nelle righe 4 e 5 costa quindi tempo $O(1)$. Poiché D ha un numero logaritmico di livelli (vedi Corollario 15.1) e ad ogni iterazione del ciclo `while` scendiamo di almeno un livello, il tempo di esecuzione totale è $O(\log n)$. □

15.3 Problemi di ricerca multidimensionali

Il problema della ricerca in spazi multidimensionali è di fondamentale importanza in numerosi contesti, tra cui basi di dati, sistemi informativi geografici, e applicazioni statistiche. Tipicamente, i dati sono tuple di d chiavi, che vengono interpretate come punti nello spazio cartesiano a d-dimensioni. Su questo insieme di punti, desideriamo eseguire diversi tipi di interrogazioni. In questo paragrafo ci concentreremo su *interrogazioni di range*: dato un certo dominio dello spazio d-dimensionale, il problema consiste nel restituire tutti i punti in esso contenuti. È chiaro che, nel caso la risposta contenga k punti, la ricerca costerà almeno tempo $O(k)$, poiché non potremo far di meglio che spendere tempo costante per ciascun punto restituito.

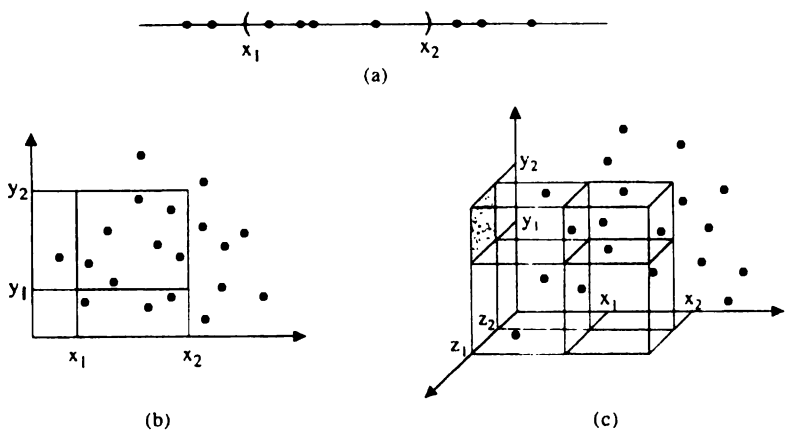


Figura 15.10 Esempi di interrogazioni ortogonali in 1, 2 e 3 dimensioni.

Nel seguito presenteremo strutture dati per il problema della ricerca in due dimensioni che richiedono tempo $O(\log n + k)$ oppure $O(\log^2 n + k)$ per interrogazione nel caso peggiore, dove n è il numero di punti nell'insieme e k è la dimensione del risultato di una specifica interrogazione. Come vedremo, il tempo dipende dal tipo di interrogazioni che la struttura è in grado di supportare. Ci concentreremo solo su interrogazioni cosiddette *ortogonali*, ovvero interrogazioni in cui il dominio di ricerca è definito da un rettangolo con lati paralleli agli assi, possibilmente con uno o più lati assenti. Esempi di domini ortogonali in 1, 2 e 3 dimensioni sono mostrati in Figura 15.10. Inoltre, poiché rendere dinamiche strutture multidimensionali è tipicamente molto più difficile che non nel caso di una sola dimensione, ci limiteremo a presentare una versione statica delle strutture dati, descrivendo solo come esse supportino vari tipi di interrogazioni. Iniziamo, per semplicità, con una descrizione del caso di una sola dimensione.

15.3.1 Il caso di una dimensione

Supponiamo di avere n punti su una retta e di voler rispondere a domande del tipo: "quali sono tutti i punti nell'intervallo $[a, b]$?". Un istante di riflessione ci fa intuire che gli alberi di ricerca studiati nel Capitolo 6 possano essere utili ai nostri scopi: una visita simmetrica, come descritta nel Paragrafo 3.3.3 del Capitolo 3, ci dà infatti le chiavi contenute nell'albero in ordine crescente.

Per semplicità, associamo i punti della retta alle foglie dell'albero in ordine crescente. Definiamo chiave di una foglia la coordinata del punto ad essa associato, e chiave di un nodo interno la massima chiave nel suo sottoalbero sinistro. In pratica, stiamo usando un'idea simile a quella vista per gli alberi 2–3, in

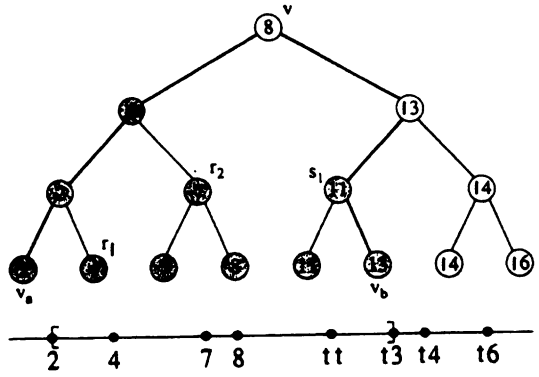


Figura 15.11 Esempio di interrogazione ortogonale in una dimensione con $a = 2$ e $b = 13$.

modo da poter effettuare ricerche facilmente. L'algoritmo di interrogazione, che chiameremo `query1D`, lavora come descritto di seguito:

Passo 1: tramite due ricerche di a e b, identifichiamo i nodi v_a e v_b contenenti rispettivamente la più piccola chiave $\geq a$ e la più grande chiave $\leq b$. Identifichiamo anche il più profondo antenato comune di v_a e v_b, ovvero il primo nodo v tale che $a \leq key(v) \leq b$;

Passo 2: restituiamo i punti associati alle foglie v_a e v_b;

Passo 3: per ogni nodo w nel cammino da v a v_a (v escluso) tale che v_a è nel sottoalbero sinistro, sia r il figlio destro di w: visitiamo il sottoalbero radicato in r e restituiamo tutti i punti associati alle sue foglie;

Passo 4: per ogni nodo w nel cammino da v a v_b (v escluso) tale che v_b è nel sottoalbero destro, sia s il figlio sinistro di w: visitiamo il sottoalbero radicato in s e restituiamo tutti i punti associati alle sue foglie.

Teorema 15.5 *L'algoritmo* `query1D` *supporta interrogazioni ortogonali in una dimensione in tempo $O(\log n + k)$, dove n è il numero di punti e k è la dimensione del risultato dell'interrogazione.*

Dimostrazione. La correttezza dell'algoritmo è garantita dal fatto che i punti sono associati alle foglie dell'albero di ricerca in ordine crescente e dalla proprietà di ricerca. Come suggerito dalla Figura 15.11, i nodi denominati r e s nell'algoritmo partizionano l'intervallo di ricerca $[a, b]$ (estremi esclusi) in intervalli disgiunti. I punti contenuti in ciascun intervallo sono restituiti separatamente. Inoltre, poiché gli intervalli sono disgiunti e la loro unione copre $[a, b]$, ogni punto è restituito esattamente una volta. Ricordiamo dal Paragrafo 3.3.3 del Capitolo 3 che una visita completa di un sottoalbero con t nodi richiede tempo $\Theta(t)$, e osserviamo

che nel nostro contesto una siffatta visita permette di restituire $\Theta(t)$ nuovi punti. Quindi il tempo speso per tutte le visite è $O(k)$, dove k è il numero totale di punti nell'intervallo $[a, b]$. A ciò si aggiunge il tempo $O(\log n)$ necessario per scendere dalla radice alle foglie v_a e v_b. L'intero algoritmo di interrogazione ha dunque tempo di esecuzione $O(\log n + k)$. □

15.3.2 Il range tree

Il range tree rappresenta una estensione naturale degli alberi binari di ricerca a dimensioni superiori. Lo descriveremo per semplicità nel caso del piano, ovvero per $d = 2$. Assumiamo quindi di avere un insieme S di n punti del piano e di volerli memorizzare in modo da poter rispondere efficientemente a interrogazioni del tipo: "quali punti sono compresi nel rettangolo $[x_1, x_2] \times [y_1, y_2]$?". Il range tree è una struttura a due livelli definita come segue:

- La *struttura primaria* è un albero di ricerca bilanciato T le cui foglie sono associate ai punti dell'insieme S ordinati per coordinata x crescente (se ci sono più punti con una stessa coordinata x, li associamo alla stessa foglia). Per ogni nodo interno v, denotiamo con $\mu(v)$ l'insieme dei punti associati alle foglie del sottoalbero radicato in v, che chiameremo per brevità *punti propri* di v.
- La *chiave* di una foglia è la coordinata x ad essa corrispondente. La chiave di un nodo interno è pari alla massima chiave nel sottoalbero sinistro.
- A ciascun nodo v è inoltre associata una *struttura secondaria* $Y(v)$ che contiene tutti i punti propri di v in ordine di coordinata y crescente: $Y(v)$ è implementata come una struttura di ricerca mono-dimensionale sulle coordinate y, come descritto nel Paragrafo 15.3.1.

Lemma 15.4 *L'occupazione di memoria di un range tree associato a n punti del piano è $O(n \log n)$.*

Dimostrazione. Basta osservare che ogni punto appare in tutte le strutture secondarie di un intero cammino radice-foglia della struttura primaria T. □

La Figura 15.12 rappresenta un insieme di punti del piano e un range tree ad essi associato, evidenziando quali siano i punti propri di ciascun nodo. Sia $R = [x_1, x_2] \times [y_1, y_2]$ il rettangolo specificato nell'interrogazione. Per enumerare i punti in R, apporteremo alcune modifiche all'algoritmo `query1D` per la ricerca in una dimensione. L'intuizione suggerisce di usare la struttura primaria per identificare i punti le cui coordinate x sono comprese in $[x_1, x_2]$. Di questi punti, però, vanno restituiti solo quelli la cui coordinata y è in $[y_1, y_2]$, spendendo tempo proporzionale al loro numero: per far ciò, interrogheremo le strutture secondarie. L'algoritmo, che chiameremo `query2D`, lavora come descritto di seguito:

Passo 1: tramite una ricerca di x_1 e x_2 nella struttura primaria, identifichiamo le foglie v_1 e v_2 associate alle coordinate orizzontali x_1 e x_2. Identifichiamo anche il più profondo antenato comune di v_1 e v_2, ovvero il primo nodo v tale che $x_1 \leq key(v) \leq x_2$;

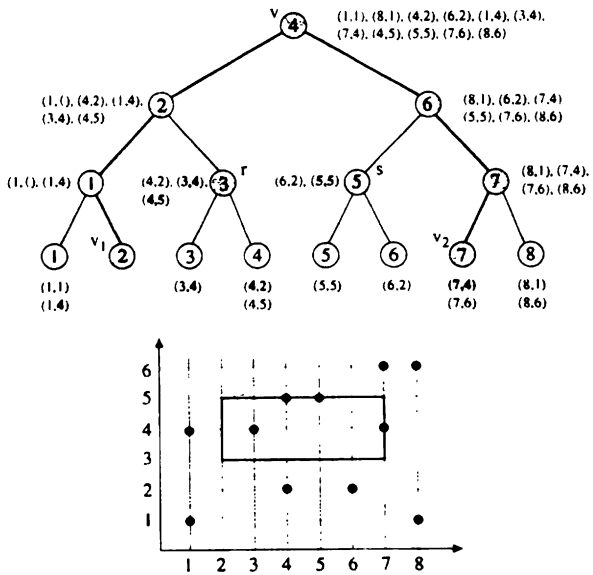


Figura 15.12 Range tree in due dimensioni: per semplicità, le strutture secondarie sono rappresentate come liste di punti, ordinati per coordinata y crescente. Evidenziamo in grassetto i punti restituiti in seguito all'interrogazione ortogonale $[2, 7] \times [3, 5]$.

Passo 2: restituiamo i punti in $Y(v_1)$ e $Y(v_2)$ aventi coordinata y in $[y_1, y_2]$;

Passo 3: per ogni nodo w nel cammino da v a v_1 (v escluso) tale che v_1 è nel sottoalbero sinistro, sia r il figlio destro di w: restituiamo i punti in $Y(r)$ con coordinata y in $[y_1, y_2]$;

Passo 4: per ogni nodo w nel cammino da v a v_2 (v escluso) tale che v_2 è nel sottoalbero destro, sia s il figlio sinistro di w: restituiamo i punti in $Y(s)$ con coordinata y in $[y_1, y_2]$.

Esaminiamo ora la correttezza e il tempo di esecuzione dell'algoritmo di interrogazione.

Teorema 15.6 *L'algoritmo* `query2D` *supporta interrogazioni ortogonali in due dimensioni in tempo $O(\log^2 n + k)$, dove n è il numero di punti e k è la dimensione del risultato dell'interrogazione.*

Dimostrazione. I nodi di tipo r e s partizionano i punti dell'insieme aventi coordinata x in $[x_1, x_2]$ in strisce verticali disgiunte di altezza infinita. Non tutti i punti in tali strisce devono essere restituiti, ma solo quelli con coordinata y in $[y_1, y_2]$:

per questo motivo viene effettuata una interrogazione di range in una dimensione sulle strutture secondarie dei nodi di tipo r e s. Ciò implica la correttezza.

Consideriamo ora il tempo di esecuzione. Nel caso peggiore, come segue dal Teorema 15.5, una interrogazione ad una struttura secondaria richiede tempo $O(\log n + t)$, dove t è il numero di punti da essa restituiti. Quante interrogazioni su strutture secondarie possiamo effettuare? Osserviamo che ogni livello della struttura primaria contiene al più un nodo di tipo r e al più un nodo di tipo s: quindi il numero di interrogazioni secondarie è $O(\log n)$, essendo logaritmico il numero di livelli della struttura primaria. Poiché ogni punto è restituito una sola volta, ciò implica tempo di esecuzione $O(\log^2 n + k)$ per l'intero algoritmo, dove k è il numero di punti nel rettangolo corrispondente all'interrogazione. □

Con poche modifiche, il range tree può essere esteso a spazi d-dimensionali con $d > 2$. Per far ciò, basterà usare una definizione ricorsiva, con la struttura dati del Paragrafo 15.3.1 come passo base. In pratica, la struttura primaria permetterà di effettuare ricerche nella prima dimensione, e le strutture secondarie di ogni nodo saranno strutture di ricerca $(d-1)$-dimensionali. Si può dimostrare che l'occupazione di memoria in tal caso è $O(n \log^{d-1} n)$ e il tempo di interrogazione è $O(\log^d n + k)$.

15.3.3 Il priority search tree

Vedremo ora che se il rettangolo di interrogazione è privo di almeno un lato, possiamo usare meno spazio e rispondere alle interrogazioni più velocemente. Ciò non è sorprendente. Consideriamo, ad esempio, un rettangolo senza due lati paralleli, ad esempio $[x_1, x_2] \times [-\infty, \infty]$: una interrogazione bidimensionale diventa in tal caso una interrogazione mono-dimensionale, poiché le ordinate y dei punti non ci interessano! Non è così immediato applicare un ragionamento del genere nel caso di interrogazioni del tipo $[x_1, x_2] \times [y, \infty]$ oppure $[-\infty, x] \times [y, \infty]$, che studieremo in questo paragrafo. Presenteremo una struttura dati, nota come *priority search tree* e dovuta a McCreight [9], che supporta entrambe queste interrogazioni in tempo e spazio ottimali.

La struttura dati. Un priority search tree è un albero binario bilanciato ai cui nodi sono associati un punto dell'insieme e una chiave in base al seguente criterio:

- la radice r contiene un punto (x^*, y^*) di coordinata y massima; per brevità, chiamiamo S' l'insieme $S \setminus \{(x^*, y^*)\}$;
- la chiave $key(r)$ della radice è il mediano delle coordinate x dei punti in S';
- il sottoalbero sinistro è un priority search tree sui punti dell'insieme S' aventi coordinata $x \leq key(r)$;
- il sottoalbero destro è un priority search tree sui punti dell'insieme S' aventi coordinata $x > key(r)$.

Un esempio di priority search tree è mostrato in Figura 15.13. Intuitivamente, un priority search tree è un ibrido tra un heap rispetto alle coordinate y (vedi i Capitoli 4 e 8) e un albero binario di ricerca rispetto alle coordinate x (vedi il

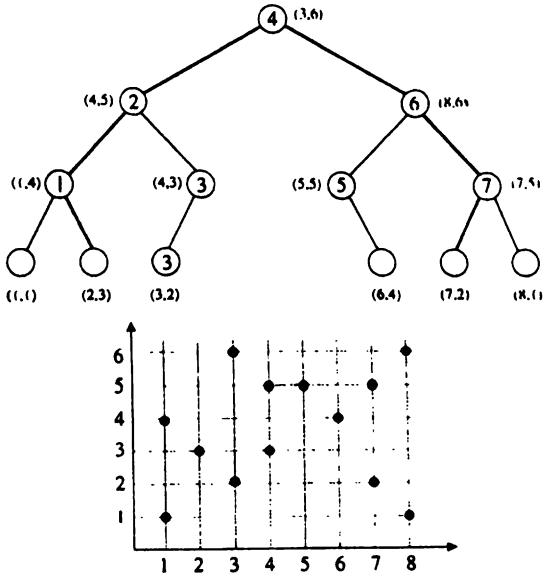


Figura 15.13 Priority search tree: il valore all'interno di ciascun nodo rappresenta la chiave, mentre il punto associato al nodo è restituito esternamente.

Capitolo 6). Poiché ogni nodo contiene una quantità costante di informazione e ogni punto è associato a uno ed un solo nodo, l'occupazione di memoria è $O(n)$.

L'algoritmo di interrogazione. Vediamo ora come usare un priority search tree per rispondere a interrogazioni ortogonali limitate al più su tre lati. Come esempio, useremo l'interrogazione $[x_1, x_2] \times [y_1, \infty]$. Lo pseudocodice dell'algoritmo `queryPriority` è mostrato in Figura 15.14.

L'algoritmo visita un sottoalbero radicato in r. Per capire quando fermarsi lungo un ramo, si sfrutta la proprietà dell'ordinamento a heap (riga 4): se il punto associato al nodo r ha coordinata verticale minore della base y_1 del rettangolo di query R, nessun punto nel suo sottoalbero può appartenere a R, e in tal caso non si esegue nessuna chiamata ricorsiva. Altrimenti, per capire se scendere nel ramo sinistro o destro, si sfrutta la proprietà di ricerca (righe 5 e 6): se l'estremo destro del rettangolo di query è minore della chiave di r, tutti i punti nel sottoalbero destro di r non possono appartenere al rettangolo R e quindi basta ricorrere sul sottoalbero sinistro. Il caso in cui l'estremo sinistro del rettangolo di query è maggiore della chiave di r è simmetrico. In ogni altro caso, ricorriamo su entrambi i figli, e questo garantisce la correttezza dell'algoritmo.

Ad una prima analisi, sembrerebbe che effettuare due chiamate ricorsive possa avere conseguenze catastrofiche sul tempo di esecuzione. In particolare, se visitassimo tanti nodi senza però restituire i punti in essi contenuti, nel caso peggiore

```
algoritmo queryPriority(priority search tree T,
                        R = [x1, x2] × [y1, ∞]) → insieme di punti
1.  r ← radice di T
2.  (x*, y*) ← punto assegnato alla radice
3.  if ( (x*, y*) ∈ [x1, x2] × [y1, ∞] ) then restituisci (x*, y*) e prosegui
4.  if ( y* < y1 ) then return
5.  else if ( x2 ≤ key(r) ) then queryPriority(sottoalbero sinistro di T,R)
6.  else if ( x1 > key(r) ) then queryPriority(sottoalbero destro di T,R)
7.  else
8.     queryPriority(sottoalbero sinistro di T,R)
9.     queryPriority(sottoalbero destro di T,R)
```

Figura 15.14 Ricerca in un priority search tree.

potremmo anche aspettarci un tempo di esecuzione proporzionale a n, anziché alla dimensione k del risultato! Come vedremo, fortunatamente questo è impossibile: durante la ricerca possiamo visitare alcuni nodi in più, ma il loro numero è basso (precisamente, logaritmico in n), come dimostriamo nel Lemma 15.5.

Lemma 15.5 *L'algoritmo* `queryPriority` *visita* $O(\log n + k)$ *nodi del priority search tree, dove* n *è il numero totale di punti e* k *è il numero di punti nel rettangolo di query.*

Dimostrazione. Sia v un nodo dell'albero cui è assegnato un punto (x, y), e siano s_v e d_v i suoi figli sinistro e destro, se esistono. Ai fini della dimostrazione, associamo al nodo un intervallo $[\ell(v), u(v)]$ come segue:

- se v è una foglia, $\ell(v) = u(v) = x$;
- se v è un nodo interno, $\ell(v) = \min\{x, \ell(s_v)\}$ e $u(v) = \max\{x, u(d_v)\}$.

Intuitivamente, $[\ell(v), u(v)]$ è l'intervallo di coordinate orizzontali "coperto" dal nodo v. È facile vedere che $\ell(v) \leq key(v) \leq u(v)$ e che, per definizione di priority search tree, gli intervalli $[\ell, u]$ coperti da nodi sullo stesso livello dell'albero sono disgiunti. Inoltre:

$$[\ell(s_v), u(s_v)] = [\ell(v), key(v)]$$

$$[\ell(r_v), u(r_v)] = [key(v), u(v)]$$

Sia $R = [x_1, x_2] \times [y_1, \infty]$ il rettangolo di query. Poiché $x_1 \leq x_2$ e $\ell(v) \leq u(v)$, rimangono solo sei possibili ordinamenti:

1. $x_1 \leq x_2 < \ell(v) \leq u(v)$: questo implica che $x_2 < key(v)$ e quindi viene effettuata la chiamata ricorsiva sul solo sottoalbero sinistro (riga 5). Possono esistere molti nodi che soddisfano questa condizione, nel caso peggiore anche in numero lineare: sia w il più alto antenato comune di tutti questi nodi. Quando w viene visitato, per quanto osservato sopra l'algoritmo prosegue esclusivamente sul suo figlio sinistro, sul figlio sinistro del figlio sinistro, e così via. In totale, potremo pertanto visitare al più $O(\log n)$ nodi di questo tipo senza restituire punti.
2. $\ell(v) \leq u(v) < x_1 \leq x_2$: il caso è simmetrico al precedente, con $key(v) < x_1$ e chiamata sul solo sottoalbero destro (riga 6).
3. $x_1 \leq \ell(v) \leq x_2 \leq u(v)$: poichè gli intervalli $[\ell, u]$ coperti da nodi sullo stesso livello dell'albero sono disgiunti, su ogni livello al più un nodo è tale che $\ell(v) \leq x_2 \leq u(v)$. Quindi, il numero totale di nodi di questo tipo è $O(\log n)$, e visitarli senza restituire punti non inficia il tempo di esecuzione.
4. $\ell(v) \leq x_1 \leq u(v) \leq x_2$: come il caso (3) ragionando su x_1;
5. $\ell(v) \leq x_1 \leq x_2 \leq u(v)$: come il caso (3);
6. $x_1 \leq \ell(v) \leq u(v) \leq x_2$: in questo caso effettuiamo sicuramente due chiamate ricorsive. Inoltre, diversamente dal caso (3), il numero di nodi di questo tipo potrebbe essere molto grande. Un'analisi più raffinata e la proprietà di ordinamento a heap sulle coordinate y ci verranno in aiuto. I nodi di tipo 6 possono essere classificati in due gruppi: figli di nodi di tipo 3 e 4, oppure figli di nodi di tipo 6.
   In base ai ragionamenti nel caso (3), solo $O(\log n)$ nodi di tipo 6 possono appartenere al primo gruppo, poiché il numero dei loro genitori è $O(\log n)$. Visitarli senza restituire punti non inficia quindi il tempo di esecuzione.
   Assumiamo ora che v sia un nodo del secondo gruppo, e chiamiamo (x, y) il punto assegnato a v. Se i figli di v vengono visitati, in base all'algoritmo y deve essere maggiore o uguale alla base del rettangolo R, altrimenti ci saremmo fermati nella riga 4. Quindi, poichè per definizione $\ell(v) \leq x \leq u(v)$, il punto (x, y) è certamente stato restituito. In conclusione, non appena troviamo un nodo di tipo 6 il cui punto associato non viene restituito, la ricerca non prosegue. Ciò implica che il numero di nodi di tipo 6 visitati è al più il doppio del numero di punti che essi restituiscono, e quindi $O(k)$.

L'algoritmo `queryPriority` visita pertanto $O(\log n + k)$ nodi del priority search tree, dove n è il numero totale di punti e k è il numero di punti nel rettangolo di query. □

Poiché la visita di ogni nodo richiede tempo $O(1)$, possiamo concludere che il tempo di esecuzione dell'algoritmo di interrogazione è $O(\log n + k)$ nel caso peggiore. L'algoritmo funziona anche nel caso in cui più di un lato sia assente e quindi, ad esempio, supporta efficientemente anche interrogazioni del tipo $[-\infty, x] \times [y, \infty]$. Riassumiamo le prestazioni dei priority search tree nel seguente teorema.

Teorema 15.7 *L'algoritmo* `queryPriority` *supporta interrogazioni ortogonali in due dimensioni del tipo* $[x_1, x_2] \times [y, \infty]$ *in tempo* $O(\log n + k)$, *dove* n *è il numero di punti e* k *è la dimensione del risultato dell'interrogazione.*

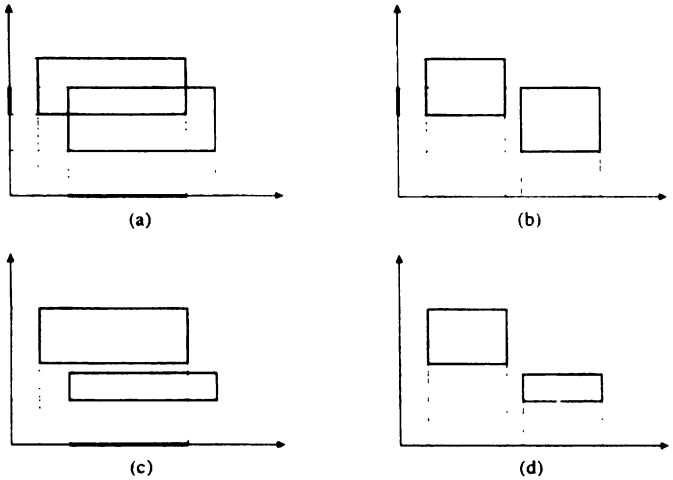


Figura 15.15 Due rettangoli isotetici si intersecano se e solo se le loro proiezioni sugli assi hanno entrambe intersezione non nulla.

15.4 Intersezione di rettangoli

In questo paragrafo studieremo il seguente problema: dato un insieme di n rettangoli isotetici, ovvero con i lati paralleli agli assi, restituire tutte le coppie di rettangoli che si intersecano. Affronteremo questo problema nel piano (cioè per $d = 2$), riducendolo opportunamente al problema in una sola dimensione. Osservando la Figura 15.15, è infatti facile convincersi della seguente proprietà:

Osservazione 15.2 *Due rettangoli isotetici si intersecano se e solo se sia le loro proiezioni lungo l'asse x che le proiezioni lungo l'asse y hanno intersezione non nulla.*

Essendo le proiezioni segmenti in uno spazio mono-dimensionale, proprio questa osservazione ci permetterà di passare da due ad una sola dimensione. Osserviamo inoltre che due segmenti $[a, a']$ e $[b, b']$ si intersecano se e solo se l'estremo iniziale dell'uno è contenuto nell'altro. Ciò può essere formalizzato come segue:

Osservazione 15.3 *Due segmenti $[a, a']$ e $[b, b']$ si intersecano se e solo se vale una delle due seguenti condizioni, che sono mutuamente esclusive:*

$$b \leq a \leq b' \tag{15.2}$$

$$a \leq b \leq a' \tag{15.3}$$

Per passare da due ad una dimensione ci avvaliamo di una tecnica molto usata in geometria, nota come *scansione del piano* (in inglese, *plane sweep*). L'uso

di questa tecnica, combinato con una potente struttura dati per risolvere il problema dell'intersezione di segmenti, ci permetterà di risolvere efficientemente il problema dell'intersezione di rettangoli.

15.4.1 Il metodo di plane sweep

La tecnica di plane sweep si basa sull'idea di trattare una delle dimensioni come "dimensione temporale". Essa fornisce un metodo per ordinare oggetti geometrici, tipicamente inserendoli in una struttura dati dinamica man mano che vengono incontrati ed esaminando opportune relazioni tra essi. Nel nostro caso, immaginiamo di "spazzare" il piano con una linea immaginaria parallela all'asse delle ordinate, da sinistra verso destra. In ogni istante, la linea interseca un insieme di rettangoli, che chiameremo *attivi*. Questo insieme varia solo in corrispondenza delle ascisse dei lati verticali dei rettangoli. In particolare, se la linea di scansione incontra una coordinata x associata al lato sinistro di un rettangolo, il rettangolo viene aggiunto all'insieme dei rettangoli attivi; se la coordinata x è invece associata al lato destro di un rettangolo, il rettangolo viene eliminato.

Quando un rettangolo diventa attivo, restituiamo i rettangoli attivi con cui si interseca. È facile convincersi della seguente proprietà, che garantisce che la correttezza del metodo di plane sweep:

Proprietà 15.1 *Se due rettangoli hanno intersezione non nulla, esiste un instante durante la scansione del piano in cui sono entrambi attivi.*

Questo implica che restituiremo tutte e sole le intersezioni volute. Come possiamo trovare i rettangoli attivi che intersecano il rettangolo R? L'intersezione delle proiezioni lungo l'asse x di tutti i rettangoli attivi è sicuramente non nulla, perché tutte le loro proiezioni condividono almeno l'ascissa della linea di scansione. Basta quindi verificare l'intersezione delle proiezioni lungo l'asse delle ordinate. In particolare, se $[y, y']$ è la proiezione di R lungo tale asse, in base all'Osservazione 15.3 occorre trovare:

1. tutte le proiezioni (ovvero, i segmenti) che contengono y;
2. tutte le proiezioni il cui estremo iniziale è contenuto nel segmento $[y, y']$.

La struttura dati che descriveremo nel prossimo paragrafo supporta efficientemente interrogazioni di intersezione di intervalli.

15.4.2 L'albero degli intervalli

L'albero degli intervalli (in inglese, *interval tree*) fu introdotto da Edelsbrunner nel 1980. Descriveremo qui una versione di questa struttura dati solo parzialmente dinamica, poiché permette di aggiungere e togliere intervalli i cui estremi appartengo a un insieme di punti fissato e noto *a priori*. Sebbene questa versione sia sufficiente per i nostri scopi, sottolineiamo che ne esiste una versione molto più sofisticata del tutto dinamica, in cui anche l'insieme degli estremi può variare.

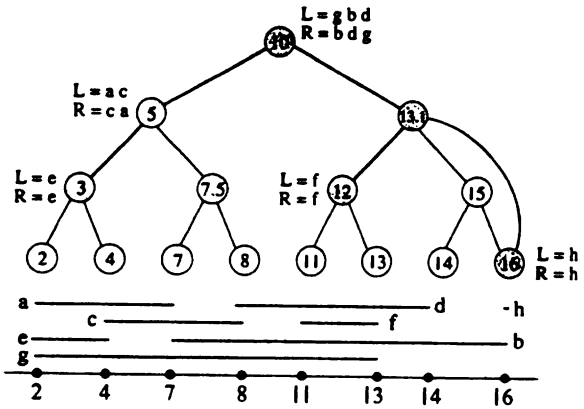


Figura 15.16 Esempio di interval tree.

Nel resto di questo paragrafo chiameremo $y_1, y_2, \ldots, y_n$ gli n punti che sono possibili estremi di intervalli, ordinati in modo crescente. Un interval tree è una struttura dati a tre livelli, nel senso che consiste di una *struttura primaria* (statica), una *struttura secondaria* e una *struttura terziaria* (dinamiche). È tipico, in geometria computazionale, aumentare strutture dati basilari, come alberi di ricerca, associando a ciascun nodo strutture secondarie contenenti informazioni utili. Gli alberi degli intervalli illustrano bene questo approccio.

- La struttura primaria è un albero binario di altezza $O(\log n)$ le cui foglie sono i punti ordinati $y_1, y_2, \ldots, y_n$. A ciascun nodo v della struttura primaria è associato un *valore discriminante* $\delta(v)$, compreso tra il massimo punto nel sottoalbero sinistro e il minimo punto nel sottoalbero destro.
- A ciascun nodo v della struttura primaria sono associate due strutture secondarie, chiamate $\mathcal{L}(v)$ e $\mathcal{R}(v)$. Supponiamo che le foglie dell'albero radicato in v siano i punti compresi tra y_i e y_k, con $i \leq k$. Le due strutture secondarie contengono gli intervalli con estremo sinistro $\geq y_i$ ed estremo destro $\leq y_k$ che attraversano il discriminante $\delta(v)$. Questi intervalli sono ordinati per estremo sinistro crescente in $\mathcal{L}(v)$ e per estremo destro decrescente in $\mathcal{R}(v)$.
- Ciascun nodo della struttura primaria è classificato come *attivo* o *inattivo*: è attivo se la sua struttura secondaria è non vuota oppure se entrambi i suoi sottoalberi contengono nodi attivi. È facile vedere che i nodi attivi possono essere connessi a formare un albero binario $\mathcal{T}$, che costituisce la struttura terziaria.

In Figura 15.16 mostriamo un esempio di interval tree. Osserviamo che al più metà dei nodi attivi può avere strutture secondarie vuote: sia le strutture secondarie delle foglie di $\mathcal{T}$ sia quelle dei nodi con un unico figlio sono infatti non vuote. Vediamo ora come gestire interrogazioni, inserimenti e cancellazioni di intervalli

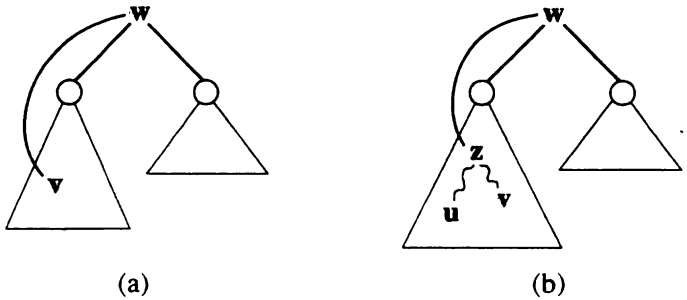


Figura 15.17 Modifica della struttura terziaria a fronte dell'inserimento di un intervallo: (a) il nodo w non aveva figlio sinistro prima dell'inserimento; (b) il nodo u era figlio sinistro di w prima dell'inserimento.

assumendo che le strutture secondarie siano implementate come alberi binari di ricerca bilanciati (ad esempio, tramite gli alberi AVL che abbiamo descritto nel Paragrafo 6.2 del Capitolo 6).

Inserimenti e cancellazioni. Per inserire un intervallo $[a, b]$ effettuiamo una ricerca sulla struttura primaria per trovare il primo nodo v tale che $a \leq \delta(v) \leq b$. Inseriamo poi l'intervallo nelle strutture secondarie di v. Se v era inattivo, lo rendiamo attivo ed aggiorniamo opportunamente l'albero $\mathcal{T}$ come descritto di seguito. Consideriamo il più profondo nodo attivo, w, nel cammino dalla radice (che assumiamo essere sempre attiva) al nodo v. Supponiamo che v sia nel sottoalbero sinistro di w (l'altro caso è simmetrico). Abbiamo due casi, come illustrato in Figura 15.17:

- se w non aveva un figlio sinistro nella struttura terziaria, rendiamo v figlio sinistro di w;
- se w aveva già un figlio sinistro u nella struttura terziaria, sia z il minimo antenato comune tra i nodi u e v. Rendiamo z figlio sinistro di w, aggiornandone opportunamente i figli ed eventualmente attivandolo.

La cancellazione procede in modo del tutto simmetrico all'inserimento e ne omettiamo la descrizione. Possiamo quindi concludere:

Lemma 15.6 *Dato un albero degli intervalli con n nodi definito su insieme di punti P, possiamo inserire un nuovo intervallo i cui estremi siano in P o cancellare un intervallo esistente in tempo $O(\log n)$ nel caso peggiore.*

Dimostrazione. Segue dal fatto che sia la struttura primaria che quelle secondarie sono alberi di ricerca bilanciati con $O(n)$ nodi, e che solo $O(1)$ nodi della struttura terziaria sono modificati. □

Interrogazioni. Vediamo ora come restituire tutti gli intervalli che intersecano un dato intervallo $[a, b]$. Come nell'inserimento, tracciamo un cammino nella struttura primaria dalla radice al primo nodo v tale che $a \leq \delta(v) \leq b$. Sia π tale cammino, che potrebbe anche essere vuoto. Da v, siano π_a e π_b i cammini divergenti che conducono alle foglie a e b. Sia u un qualunque nodo in $\pi \cup \pi_a \cup \pi_b$. Interroghiamo una delle strutture secondarie di u come segue.

- $u \in \pi$: l'intervallo $[a, b]$ sta o interamente a sinistra o interamente a destra di $\delta(u)$. Consideriamo il primo caso, ovvero $b \leq \delta(u)$: per verificare se $[a, b]$ interseca qualcuno degli intervalli associati a u, è sufficiente scorrere $\mathcal{L}(u)$ (e restituire gli intervalli che vi si trovano) finché non troviamo il primo nodo con estremo sinistro $> b$. Nell'esempio di Figura 15.16, se $[a, b] = [2, 7]$, la radice è un nodo del cammino π, e l'esame della sua struttura secondaria $\mathcal{L}$ ci porta a restituire ordinatamente gli intervalli g e b. Il caso $a \geq \delta(u)$ è simmetrico: scorreremo $\mathcal{R}(u)$ fino a trovare il primo nodo con estremo destro $< a$.
- $u \in \pi_a$: se $\delta(u) \leq a$, per verificare se $[a, b]$ interseca qualcuno degli intervalli associati a u, è sufficiente scorrere $\mathcal{R}(u)$ e restituire gli intervalli che vi si trovano, finché non troviamo il primo nodo con estremo destro $< a$. Nell'esempio di Figura 15.16, se $[a, b] = [8, 11]$, il nodo con discriminante 5 è un nodo del cammino π_a, e l'esame della sua struttura secondaria $\mathcal{R}$ ci porta a restituire l'intervalli c.
  Il caso $a \leq \delta(u)$ è invece più delicato e coinvolge l'uso della struttura terziaria. Osserviamo innanzitutto che il punto b non è coperto dal sottoalbero radicato in u. Quindi, il segmento $[a, b]$ attraversa $\delta(u)$ e di conseguenza tutti gli intervallo associati al nodo u. Inoltre, $[a, b]$ interseca anche tutti gli intervalli assegnati a nodi nel sottoalbero destro di u: per restituire questi intervalli, visitiamo i nodi attivi nel sottoalbero destro di u restituendo tutti gli intervalli nelle loro strutture secondarie. Poiché almeno la metà dei nodi attivi ha a suo carico almeno un intervallo, il tempo speso sarà proporzionale al numero di intervalli restituiti.
  Nell'esempio di Figura 15.16, se $[a, b] = [4, 11]$, il nodo con discriminante 5 è un nodo del cammino π_a, e tutti gli intervalli nelle sue strutture secondarie intersecano $[4, 11]$ e sono restituiti. Dovremmo inoltre restituire tutti gli intervalli nel sottoalbero radicato nel nodo con discriminante 7.5: in questo esempio non esistono però intervalli di questo tipo, come è facile verificare dal fatto che tale nodo non è attivo nella struttura terziaria.
- $u \in \pi_b$: con gli appropriati cambiamenti, questo caso è simmetrico al caso $u \in \pi_a$. Lasciamo quindi la sua implementazione per esercizio.

Lemma 15.7 *Dato un albero degli intervalli con n nodi definito su insieme di punti P, possiamo restituire tutti e soli gli intervalli che intersecano un dato intervallo $[a, b]$ in tempo $O(\log n + f)$, dove f è il numero totale di tali intervalli.*

Dimostrazione. Il tempo speso su ciascuna struttura secondaria è limitato dal numero di intervalli in essa contenuti che intersecano $[a, b]$ più 1. Poichè le due discese sulla struttura primaria comportano $O(\log n)$ passi ciascuna, e l'analisi della struttura terziaria richiede tempo proporzionale al numero di intervalli $O(f)$ effettivamente restituito, il tempo di esecuzione totale è $O(\log n + f)$. Inoltre, ogni intervallo è restituito una sola volta, poiché è associato ad un unico nodo della struttura primaria dell'interval tree. □

Torniamo ora al problema di intersezione dei rettangoli. Analizzando l'algoritmo risultante dal metodo di plane sweep e dall'uso dell'albero degli intervalli possiamo affermare che:

Teorema 15.8 *È possibile risolvere il problema dell'intersezione dei rettangoli in tempo $O(s \log s + r)$ nel caso peggiore, dove s è il numero totale di rettangoli dell'insieme in ingresso e r è il numero di quelli che si intersecano.*

Dimostrazione. Il metodo di plane sweep richiede innanzitutto di ordinare al più $2s$ ascisse dei lati verticali dei rettangoli: ciò può essere fatto in tempo $O(s \log s)$ utilizzando uno degli algoritmi ottimi che abbiamo descritto nel Capitolo 4. Nella fase di scansione, ogni qualvolta eseguiamo una interrogazione sull'interval tree, il segmento verticale $[a, b]$ corrisponde al lato sinistro di un diverso rettangolo dell'insieme. Quindi, se due rettangoli R_1 e R_2 si intersecano e R_1 viene incontrato prima di R_2, l'intersezione verrà restituita una sola volta, ovvero non appena R_2 diventa attivo. In totale si eseguono $O(s)$ interrogazioni, ciascuna delle quali, in base al Lemma 15.7, richiede tempo $O(\log s + r_i)$, dove r_i è il numero di intersezioni restituite e $\sum_i r_i = r$. Si eseguono inoltre $O(s)$ inserimenti e cancellazioni nell'albero degli intervalli, ciascuno dei quali richiede tempo $O(\log s)$ per il Lemma 15.6. Il tempo totale risulta essere $O(s \log s + r)$. □

È possibile dimostrare che questo tempo di esecuzione è ottimo, a meno di costanti moltiplicative.

15.5 Problemi

Problema 15.1 (*) Dimostrare una delimitazione inferiore di $\Omega(n \log n)$ al problema del calcolo dell'inviluppo convesso, tramite una riduzione dal problema dell'ordinamento. In particolare, dimostrare che possiamo trasformare in tempo lineare un'istanza del problema dell'ordinamento in un'istanza del problema dell'inviluppo convesso in mode tale che, se sapessimo calcolare l'inviluppo convesso in tempo inferiore a $O(n \log n)$, sapremmo anche ordinare n elementi in tempo inferiore a $O(n \log n)$ nel modello dei confronti.

Problema 15.2 (**) Dati n punti nel piano a coordinate intere in $[1, n] \times [1, n]$, progettare un algoritmo per calcolare l'inviluppo convesso in tempo lineare. Estendere poi l'approccio in modo da ordinare in tempo e spazio $O(n)$ anche punti le cui coordinate siano interi in $[1, n^k] \times [1, n^k]$, dove k è una costante.

Problema 15.3 Dati n panti nel piano, progettare an algoritmo per calcolare, in tempo $O(n \log n)$, un poligono semplice che abbia i punti come vertici. Un poligono si dice semplice se nessuna coppia di lati non consecutivi ha un punto in comune. Si noti che un poligono semplice può presentare concavità.

Problema 15.4 Dato un poligono con n vertici, mostrare un semplice algoritmo per triangolare il poligono in tempo $O(n^3)$ (il poligono può presentare concavità). Il problema sembra essere più semplice se il poligono è convesso?

Problema 15.5 Considera l'algoritmo di ricerca in una dimensione proposto nel Paragrafo 15.3.1.

- Perché tempo $O(\log n)$ è inevitabile anche quando i punti da restituire sono in numero costante? Dare un esempio di interrogazione in cui non esistono né nodi di tipo r né nodi di tipo s.
- Come rimuoveresti l'assunzione che gli estremi dell'intervallo $[a, b]$ siano punti dell'insieme?
- Come modificheresti l'algoritmo in modo da restituire i punti in ordine crescente?
- Assumi di modificare la struttura dati in modo che i punti siano associati non solo alle foglie dell'albero, ma anche ai nodi interni. Come definiresti la chiave di un nodo? La nuova struttura dati sarebbe più vantaggiosa in termini di occupazione di memoria rispetto a quella proposta? Come modificheresti l'algoritmo per supportare interrogazioni di range in una struttura così definita?

Problema 15.6 Il priority search tree supporta query del tipo $[x_1, x_2] \times [-\infty, y]$? Se no, come lo modificheresti per supportarle?

Problema 15.7 (*) Dato un array non ordinato A di n numeri, una interrogazione di minimo nell'intervallo $[i, j]$, con $1 \leq i \leq j \leq n$, chiede di restituire il minimo tra i numeri $A[i]$, $A[i+1]$, ... $A[j]$. Dato un albero radicato T, una interrogazione di minimo antenato comune su due nodi u e v chiede di restituire il più profondo nodo che è antenato sia di u che di v. Assumendo che esista una struttara dati che permette di calcolare il minimo antenato comune in tempo $O(1)$, per ogni coppia di nodi, mostra come sfruttare l'idea del priority search tree in modo da poter rispondere ad interrogazioni di minimo nell'intervallo in tempo $O(1)$.

Problema 15.8 (**) Una interrogazione diagonale chiede di restituire i punti interni all'area $[-\infty, x] \times [x, \infty]$. Una interrogazione di intersezione di intervalli chiede di restituire tutti gli intervalli di un certo insieme che intersecano un dato intervallo $[a, b]$. Dimostrare che è possibile risolvere il problema dell'intersezione di intervalli mediante interrogazioni diagonali. Mostrare poi come usare un interval tree per rispondere a una interrogazione diagonale.

Problema 15.9 Sia S un insieme di n intervalli, e sia X l'insieme di intervalli di S che *non* intersecano un dato intervallo $[\ell, r]$. È possibile usare un interval tree per restituire gli intervalli in X in tempo $O(\log n + |X|)$? Se no, è possibile modificare l'interval tree per supportare questo tipo di interrogazione?

Problema 15.10 Sia S un insieme di n intervalli e sia p un punto. Una *stabbing query* richiede di restituire tutti gli intervalli contenenti p.

a) Mostrare come usare un interval tree per rispondere a una stabbing query e discutere il tempo di esecuzione nel caso peggiore.

b) Se nell'interval tree non fossero memorizzate le identità degli intervalli nelle strutture secondarie, ma solo il loro numero totale, si potrebbe rispondere a stabbing query di conteggio, che richiedono di restituire il numero di intervalli che includono un dato punto?

Problema 15.11 Sia S un insieme di n intervalli, sia $[\ell, r]$ un dato intervallo, e sia f la dimensione del risultato di una certa interrogazione. Mostrare come usare un interval tree su S per restituire:

1. gli intervalli di S che contengono $[\ell, r]$;
2. gli intervalli di S che sono contenuti in $[\ell, r]$;
3. gli intervalli di S che intersecano $[\ell, r]$ ma non sono contenuti in esso;
4. gli intervalli di S che intersecano $[\ell, r]$ ma non lo contengono.

Discutere i tempi di esecuzione degli algoritmi di interrogazione, sottolineando quando è possibile ottenere tempo di esecuzione $O(\log n + f)$, ovvero dipendente linearmente dalla dimensione del risultato. Nel caso il tempo di esecuzione sia superiore, discutere se è possibile modificare l'interval tree in modo da supportare la specifica interrogazione nel tempo suddetto.

15.6 Sommario

In questo capitolo abbiamo affrontato, tramite quattro esempi salienti, il progetto di algoritmi per problemi geometrici. Le istanze di tali problemi consistono tipicamente in insiemi di punti nello spazio Euclideo e in oggetti geometrici ottenibili dalla composizione di punti, quali segmenti o poligoni. Lo spazio può avere dimensione $d \geq 1$, sebbene in questo capitolo ci siamo principalmente soffermati sui casi $d = 1$ e $d = 2$. Abbiamo affrontato due principali categorie di problemi:

- nel caso dell'inviluppo convesso che abbiamo studiato nel Paragrafo 15.1, si richiede di costruire un oggetto geometrico con particolari proprietà a partire dagli oggetti ricevuti in ingresso;
- nei problemi che abbiamo trattato nei rimanenti paragrafi, si richiede invece di realizzare strutture dati per poter rispondere efficientemente ad interrogazioni sulle proprietà degli oggetti in ingresso. L'interrogazione può avere una risposta binaria o può richiedere di enumerare tutti gli oggetti (o sottoinsiemi di oggetti, come nel caso dell'intersezione di rettangoli) che soddisfano una certa proprietà: in questo caso, il tempo di esecuzione dell'algoritmo di interrogazione deve essere almeno proporzionale al numero di oggetti restituiti.

Nello studio dei problemi affrontati abbiamo avuto modo di presentare delle tecniche abbastanza generali che trovano applicazione in innumerevoli altri contesti, come ad esempio la tecnica del *backtracking* e il metodo di scansione del piano.

Osserviamo inoltre che i problemi di ricerca multidimensionali studiati nel Paragrafo 15.3 sono una naturale generalizzazione a dimensioni $d \geq 2$ del problema del dizionario, che abbiamo trattato nei Capitoli 6 e 7 di questo libro.

15.7 Note bibliografiche

La geometria computazionale è un'area estremamente vasta: rimandiamo pertanto a libri di testo specifici, quali [3, 4, 11], per una descrizione dettagliata ed esaustiva degli innumerevoli problemi in essa affrontati. Il progetto di algoritmi efficienti per calcolare l'inviluppo convesso di un insieme di punti del piano fu affrontato con successo nei primi anni '70: a questo periodo risalgono infatti gli algoritmi di Jarvis [7] e di Graham [6], che abbiamo descritto nel Paragrafo 15.1. In seguito, Yao [14] ha dimostrato una delimitazione inferiore di $\Omega(n \log n)$ per questo problema, da cui segue che l'algoritmo di Graham è ottimo nel caso peggiore. La struttura dati per la localizzazione di punti in suddivisioni planari che abbiamo descritto nel Paragrafo 15.2 è dovuta a Kirkpatrick [8]. Questo problema è stato successivamente molto studiato, e Snoeyink in [13] dà una esauriente panoramica sui principali risultati ottenuti. Esiste anche una vastissima letteratura relativa a problemi di ricerca multidimensionali, come evidenziato ad esempio in [10, 11, 12]: il range tree ed il priority search tree che abbiamo descritto nel Paragrafo 15.3 sono dovuti rispettivamente a Bentley [1] e McCreight [9]. Infine, la soluzione al problema dell'intersezione di rettangoli che abbiamo proposto nel Paragrafo 15.4 è dovuta ad Edelsbrunner [5].

Riferimenti bibliografici

[1] J. L. Bentley, "Multidimensional divide and conquer", *Communications of the ACM*, 23:214–229, 1980.

[2] B. Chazelle, "Triangulating a simple polygon in linear time", *Discrete Computational Geometry*, 6(5):485–524, 1991.

[3] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational geometry: algorithms and applications*, seconda edizione, Springer Verlag, 2000.

[4] H. Edelsbrunner, *Algorithms in combinatorial geometry*, volume 10 delle EATCS Monographs in Theoretical Computer Science, Springer Verlag, 1987.

[5] H. Edelsbrunner, "A new approach to rectangle intersections", *International Journal Computational Mathematics*, 13:209–219 e 221–229, 1983.

[6] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", *Information Processing Letters*, 1(4): 132–133, 1972.

[7] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane", *Information Processing Letters*, 2(1):18–21, 1973.

[8] D. G. Kirkpatrick, "Optimal search in planar subdivisions", *SIAM Journal on Computing*, 12(1):28–35, 1983.

[9] E. McCreight, "Priority search trees", *SIAM Journal on Computing*, 14(2):257–276, 1985.

[10] K. Mehlhorn, *Data structures and algorithms 3: multi-dimensional searching and computational geometry*, Springer-Verlag, 1984.

[11] F. P. Preparata e M. I. Shamos, *Computational geometry: an introduction*, Springer Verlag, 1985.

[12] H. Samet, *The design and analysis of spatial data structures*, Addison Wesley, 1990.

[13] J. Snoeyink, "Point location", *Handbook of discrete and computational geometry*, CRC Press, 559–574, 1997.

[14] A. C.-C. Yao, "A lower bound to finding convex hulls", *Journal of the Association for Computing Machinery*, 28(4):780–787, 1981.

16

Teoria della NP-completezza

Avessimo almeno spazio sufficiente, e tempo...

(Andrew Marvell)

Tutti i problemi che abbiamo finora affrontato ammettono algoritmi risolutivi il cui tempo di esecuzione è polinomiale nella dimensione n dell'istanza di ingresso. Ad esempio, abbiamo visto algoritmi che richiedono tempo lineare o quadratico, e difficilmente ci siamo imbattuti in tempi superiori a $O(n^3)$. Anche tempi di esecuzione in cui compaiono fattori logaritmici rientrano in questa classificazione, poiché possono sempre essere limitati superiormente da un polinomio in n: ad esempio, $O(n \log n) = O(n^{1+\epsilon})$ per ogni $\epsilon > 0$. Informalmente, la classe dei problemi che ammettono algoritmi polinomiali è nota come classe P, e questi problemi sono ritenuti trattabili.

Sfortunatamente, per molti problemi di grande interesse pratico algoritmi polinomiali non solo non sono stati ancora progettati, ma si presume che non esistano affatto! Per molti di questi problemi, gli unici algoritmi risolutivi attualmente noti richiedono tempo almeno esponenziale in n. Ciò può essere proibitivo per un loro effettivo impiego su istanze reali di grandi dimensioni: per avere una soluzione, dovremmo attendere mesi, anni, o magari secoli, anche usando i calcolatori più potenti e veloci disponibili sul mercato!

In questo capitolo studieremo molti problemi ritenuti "difficili", e li classificheremo in una gerarchia di complessità via via crescente. Introdurremo il concetto di modello di calcolo non deterministico, in cui una computazione può diramarsi in un albero delle computazioni che si sviluppano indipendentemente, e la classe NP dei problemi risolvibili in tempo polinomiale non deterministico. In particolare, definiremo la classe dei problemi cosiddetti NP-completi mostrando che, se fossimo in grado di progettare un algoritmo polinomiale per uno solo di essi, allora avremmo algoritmi polinomiali per tutti i problemi in NP, e la gerarchia collasserebbe! Rispondere alla domanda

$$\mathrm{P} = \mathrm{NP}?$$

ovvero comprendere il potere di calcolo del non determinismo, attualmente è uno dei problemi aperti di maggiore importanza nelle discipline informatiche. Poiché la maggior parte dei ricercatori congettura che la risposta sia negativa, vedremo

che spesso ci si limita a risolvere i problemi ritenuti difficili in maniera approssimata (anziché pretendere una soluzione esatta), purché il tempo di esecuzione rimanga polinomiale e la soluzione ottenuta non si discosti troppo da quella ottima.

16.1 Complessità di problemi decisionali

Nella teoria della *complessità computazionale* desideriamo caratterizzare i problemi in termini delle risorse di calcolo richieste per la loro soluzione, quali tempo ed occupazione di memoria: informalmente, una *classe di complessità* è un insieme di problemi che possono essere risolti usando le stesse risorse di calcolo. In questo paragrafo introdurremo i concetti fondamentali per definire più formalmente una classe di complessità, e presenteremo alcune importanti classi.

16.1.1 Decidere, ricercare, ottimizzare

Poiché i problemi algoritmici possono avere una natura molto diversa, tentiamo innanzitutto di darne almeno una prima caratterizzazione. In generale, possiamo pensare a un problema P come a una relazione $P \subseteq I \times S$, dove I è l'insieme delle *istanze di ingresso* e S quello delle *soluzioni*. Possiamo anche immaginare di avere un predicato che, presa un'istanza $x \in I$ ed una soluzione $s \in S$, restituisce 1 se $(x, s) \in P$ (ovvero se s è una soluzione del problema P sull'istanza x), e 0 altrimenti. Basandoci su questa terminologia, classifichiamo innanzitutto i problemi in problemi di decisione, di ricerca o di ottimizzazione:

Problemi di decisione: si tratta dei problemi che richiedono una risposta binaria, e quindi $S = \{0, 1\}$. In particolare, essi richiedono di verificare se l'istanza x soddisfa una certa proprietà, ad esempio, se una grafo è connesso, se un numero è primo, se un elemento è contenuto in un dizionario. Possiamo partizionare le istanze di un problema di decisione P in istanze *positive*, ovvero istanze x tali che $(x, 1) \in P$, ed istanze *negative*, ovvero istanze x tali che $(x, 0) \in P$.

Problemi di ricerca: data una certa istanza x, questi problemi richiedono di restituire una soluzione s tale che $(x, s) \in P$. Ad esempio, trovare un albero ricoprente di un grafo, un cammino tra due nodi, o il mediano di un insieme di elementi sono tipici problemi di ricerca.

Problemi di ottimizzazione: in questo caso, data un'istanza x, si vuole trovare la migliore soluzione s^* tra tutte le possibili soluzioni s per cui $(x, s) \in P$. La bontà di una soluzione viene valutata secondo un criterio specificato nel problema stesso. Un tipico problema di ottimizzazione è la ricerca del minimo albero ricoprente in un grafo pesato, che abbiamo affrontato nel Capitolo 12: in questo caso un albero ricoprente è considerato migliore di un altro se ha peso inferiore, dove il peso è la somma dei costi degli archi in esso contenuti. Altri tipici problemi di ottimizzazione sono il calcolo del massimo flusso in una rete (Capitolo 14), di un cammino semplice di lunghezza minima o massima tra una coppia di nodi

(Capitolo 13), del miglior ordine in cui eseguire alcuni job in un computer (Capitolo 10). In generale, i problemi di ottimizzazione sono ulteriormente distinti in problemi di *massimizzazione* o *minimizzazione*.

I principali concetti della teoria della complessità computazionale sono stati definiti in termini di problemi di decisione. Per tali problemi, infatti, essendo la risposta binaria, non dobbiamo preoccuparci del tempo richiesto per restituire la soluzione, e tutto il tempo è quindi speso esclusivamente per il calcolo. Un simile ragionamento vale per l'occupazione di memoria. Si potrebbe obiettare che molti problemi di interesse pratico sono dati in forma di ottimizzazione. È però possibile esprimere anche un problema di ottimizzazione in forma decisionale. Considerando, ad esempio, il calcolo del minimo albero ricoprente, potremmo esprimere il problema come segue: "dato un grafo pesato G ed un intero k, esiste un albero ricoprente di G di costo al più k?". Intuitivamente, questo problema non sembra più difficile di quello di ottimizzazione: se siamo in grado di trovare la soluzione ottima (ovvero l'albero ricoprente minimo), siamo anche in grado di rispondere alla domanda precedente per ogni possibile valore di k, semplicemente confrontando k con il valore dell'ottimo! In altre parole, il problema di minimizzazione è almeno tanto difficile quanto il corrispondente problema decisionale, e caratterizzare la complessità di quest'ultimo permette quindi di dare almeno una limitazione inferiore alla complessità del primo.

16.1.2 Classi di complessità

Dati un problema di decisione P ed un algoritmo $\mathcal{A}$, diciamo che $\mathcal{A}$ risolve il problema P se $\mathcal{A}$ restituisce 1 su un'istanza x se e solo se $(x, 1) \in P$. Inoltre, diciamo che $\mathcal{A}$ risolve P in tempo $t(n)$ e spazio $s(n)$ se il tempo di esecuzione e l'occupazione di memoria di $\mathcal{A}$ sono rispettivamente $t(n)$ e $s(n)$. Possiamo ora definire alcune importanti classi di complessità.

Definizione 16.1 *Data una qualunque funzione $f(n)$, chiamiamo* $\textsc{Time}(f(n))$ *e* $\textsc{Space}(f(n))$ *gli insiemi dei problemi decisionali che possono essere risolti rispettivamente in tempo e spazio $O(f(n))$.*

Ad esempio, il problema di verificare se un certo elemento è presente in un dizionario contenente n elementi appartiene alle classi $\textsc{Time}(\log n)$ e $\textsc{Space}(n)$, come segue dalle strutture dati presentate nel Capitolo 6. Poiché questa classificazione è fin troppo dettagliata per i nostri scopi, presentiamo ora alcune classi più ampie, ottenute dall'unione delle classi introdotte nella Definizione 16.1 e pertanto note come *classi unione*.

Definizione 16.2 *La classe* P *è la classe dei problemi risolvibili in tempo polinomiale nella dimensione n dell'istanza di ingresso:*

$$\mathrm{P} = \bigcup_{c=0}^{\infty} \textsc{Time}(n^c)$$

La classe PSPACE *è la classe dei problemi risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso:*

$$\text{PSPACE} = \bigcup_{c=0}^{\infty} \text{SPACE}(n^c)$$

La classe EXPTIME *è la classe dei problemi risolvibili in tempo esponenziale nella dimensione n dell'istanza di ingresso:*

$$\text{EXPTIME} = \bigcup_{c=0}^{\infty} \text{TIME}(2^{n^c})$$

Poiché un algoritmo che richiede tempo polinomiale può avere accesso al più a un numero polinomiale di diverse locazioni di memoria, è chiaro che P⊆PSPACE. Inoltre, risulta PSPACE⊆EXPTIME: intuitivamente, se assumiamo per semplicità che le locazioni di memoria siano binarie, n^c diverse locazioni di memoria possono trovarsi in al più 2^{n^c} stati diversi (una dimostrazione formale di questa inclusione richiederebbe l'utilizzo del modello di calcolo noto come *macchina di Turing*, cui abbiamo accennato nel Paragrafo 2.1 del Capitolo 2, ma è al di là dello scopo di questo libro).

Non è noto se le inclusioni siano proprie, ovvero se P⊂PSPACE e/o PSPACE ⊂EXPTIME. Entrambi sono problemi di teoria della complessità ancora aperti! L'unico risultato di separazione dimostrato finora mostra che P⊂EXPTIME, e quindi esiste un problema che può essere risolto in tempo esponenziale, ma per cui tempo polinomiale non è sufficiente. Abbiamo già visto, fin dal Capitolo 1 relativamente al calcolo dei numeri di Fibonacci, la sostanziale differenza tra un tempo di esecuzione esponenziale ed uno polinomiale. Tipicamente, la classe P è pertanto considerata come una classe soglia tra problemi trattabili e problemi intrattabili.

Esempi. Mentre abbiamo incontrato numerosi problemi in P nel corso di questo libro, diamo ora alcuni esempi di problemi che appartengono a PSPACE o EXPTIME ma per cui algoritmi polinomiali non sono noti (e probabilmente non esistono affatto). I nostri esempi sono varianti di problemi di *soddisfacibilità* di formule Booleane: come vedremo nel Paragrafo 16.3, lo studio di problemi di soddisfacibilità si è rivelato molto importante nello sviluppo della teoria della complessità.

Innanzitutto, richiamiamo il concetto di *forma normale congiuntiva*. Dato un insieme V di variabili Booleane, chiamiamo *letterale* una variabile o la sua negazione, e *clausola* una disgiunzione di letterali. Una espressione Booleana su V si dice in forma normale congiuntiva se è espressa come congiunzione di clausole. Ad esempio, posto $V = \{x, y, z, w\}$, se denotiamo con $\wedge$ l'operatore "and" Booleano e con $\vee$ l'operatore "or" Booleano, l'espressione $(x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$ è in forma normale congiuntiva, mentre l'espressione $(y \wedge (\overline{w} \vee z)) \vee (\overline{x} \wedge y)$ non lo è. Una *formula Booleana quantificata* è una espressione in forma normale congiuntiva preceduta da una sequenza arbitrariamente lunga di quantificatori universali ($\forall$) ed esistenziali ($\exists$) che legano tutte le variabili. Ad esempio, $\exists x\ \forall y\ \exists z\ \forall w :\ (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$ è una formula Booleana quantificata. I due problemi che considereremo sono i seguenti.

Definizione 16.3 *Data una espressione in forma normale congiuntiva, il problema della* soddisfacibilità *richiede di verificare se esiste una assegnazione di valori di verità alle variabili che rende l'espressione vera. Il problema delle* formule


```
algoritmo formulaQuantificata(f = ξ_1 x_1 ... ξ_n x_n : e(x_1, ..., x_n),
                    assegnazione z_1, ..., z_{i-1} alle variabili x_1, ..., x_{i-1},
                    indice i ) → booleano
1.   if ( i = n + 1 ) then return e(z_1, ..., z_n)
2.   else
3.      v_0 ← formulaQuantificata(ξ_{i+1} x_{i+1} ... ξ_n x_n : e(x_1, ..., x_n),
4.                                z_1, ..., z_{i-1}, 0, i + 1)
5.      v_1 ← formulaQuantificata(ξ_{i+1} x_{i+1} ... ξ_n x_n : e(x_1, ..., x_n),
6.                                z_1, ..., z_{i-1}, 1, i + 1)
7.      if ( ξ_i = ∃ ) then return v_0 ∨ v_1
8.      else return v_0 ∧ v_1
```

Figura 16.1 Algoritmo esponenziale per risolvere il problema delle formule Booleane quantificate. Nello pseudocodice $e(x_1, \ldots, x_n)$ denota una espressione Booleana in forma normale congiuntiva.

Booleane quantificate *richiede invece di verificare se una certa formula Booleana quantificata è vera.*

Ad esempio, l'espressione $(x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$ è soddisfatta dall'assegnazione $x = 1, y = 1, z = 0, w = 1$, mentre è facile verificare che l'espressione $(x \vee \overline{y}) \wedge \overline{x} \wedge y$ non è soddisfacibile. La formula Booleana quantificata $\exists x\ \forall y\ \exists z\ \forall w :\ (x \vee \overline{w} \vee z) \wedge (\overline{x} \vee w) \wedge y$ non è vera (l'ultima clausola non può essere verificata da $y = 0$, ma y è preceduta dal quantificatore universale), mentre diventa vera cambiando l'ordine delle variabili quantificate: $\exists y\ \forall x\ \exists w\ \forall z :\ (x \vee \overline{w} \vee z) \wedge (\overline{x} \vee w) \wedge y$.

Entrambi i problemi possono essere facilmente risolti in tempo esponenziale. Consideriamo il caso delle formule Booleane quantificate. Un possibile algoritmo ricorsivo esamina la formula da sinistra verso destra assegnando valori di verità alle variabili nell'ordine in cui esse sono quantificate. Assumiamo che tutte le variabili da x_1 a x_{i-1} siano state assegnate, e sia x_i l'i-esima variabile quantificata. Sia f_{i+1} la sotto-formula a destra della quantificazione di x_i, e siano v_0 e v_1 i valori di f_{i+1} ottenuti assegnando a x_i i valori 0 e 1, rispettivamente. Se x_i è quantificata esistenzialmente, il valore della formula $\exists x_i :\ f_{i+1}$ è $v_1 \vee v_0$, altrimenti il valore della formula $\forall x_i :\ f_{i+1}$ è $v_1 \wedge v_0$. Lo pseudocodice per questo algoritmo è mostrato in Figura 16.1.

Un algoritmo per soddisfacibilità è persino più semplice: possiamo infatti immaginare che le variabili siano tutte quantificate esistenzialmente ed omettere il test nella riga 7, restituendo sempre $v_1 \vee v_0$. Se n è il numero delle variabili, gli algoritmi esaminano 2^n diverse configurazioni, dimostrando che entrambi i problemi sono in EXPTIME. Possiamo essere anche più precisi: infatti, poiché la memoria usata durante l'esecuzione è lineare, possiamo affermare che entrambi i problemi sono in PSPACE⊆EXPTIME.

16.2 La classe NP

Come abbiamo già osservato, in un problema di decisione siamo interessati a verificare se una istanza x del problema soddisfa una certa proprietà. Spesso, in caso di risposta affermativa, oltre alla semplice risposta binaria si richiede anche di fornire un qualche oggetto y, dipendente dall'istanza x e dal problema specifico, che possa certificare il fatto che x soddisfa la proprietà, giustificando quindi la risposta affermativa. Tale oggetto è noto come *certificato*, ed intuitivamente è una soluzione costruttiva per il problema. Ad esempio, un certificato per soddisfacibilità è un'assegnazione di verità alle variabili che rende vera l'espressione. Analogamente, un certificato per la connettività di un grafo è un suo albero ricoprente. Proviamo ora a pensare a un certificato per il problema delle formule Booleane quantificate: un po' di riflessione ci farà intuire che, per via dei quantificatori universali, non basta esibire un'assegnazione di valori di verità alle variabili, ma ne servono un numero addirittura esponenziale nel caso peggiore! Quindi, per questo problema, perfino esprimere un certificato sembra difficile.

Ciò suggerisce che il costo di verificare che un certo oggetto matematico sia una soluzione per un dato problema può essere usato come discriminante per caratterizzare la complessità del problema stesso. Formalmente, immaginiamo che l'istanza di ingresso e la soluzione di un problema siano descritti con una stringa di simboli su un qualche alfabeto finito, in base a precise convenzioni di codifica. Una qualunque stringa di caratteri può essere pensata come un certificato per un'istanza del problema. Verificare che la stringa sia effettivamente un certificato include verificare che sia sintatticamente corretta e che descriva costruttivamente una soluzione del problema.

Informalmente, NP *è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale.*

Ad esempio, poiché possiamo verificare in tempo lineare che un'assegnazione di verità a un insieme variabili rende vera una certa espressione, o che un albero è un albero ricoprente di un certo grafo, possiamo dedurre che sia soddisfacibilità sia connettività appartengono alla classe NP. Non possiamo però dedurre altrettanto per il problema delle formule Booleane quantificate: attualmente, non è infatti noto se tale problema sia in NP, sebbene si congetturi che non lo sia!

Riflettiamo ora sull'origine dell'acronimo NP. L'intuizione ci suggerisce (correttamente) che P stia per "polinomiale", ma il significato di N non è affatto chiaro. Per capirlo, definiamo la classe NP più formalmente introducendo il concetto di *algoritmo non deterministico*, da cui N.

16.2.1 Non determinismo

Negli algoritmi che abbiamo visto finora il passo successivo è sempre univocamente determinato dallo stato della computazione: essi sono infatti detti *deterministici*. Un algoritmo non deterministico, invece, oltre alle normali istruzioni, può eseguire istruzioni del tipo *indovina* $z \in \{0,1\}$, ovvero può "indovinare"

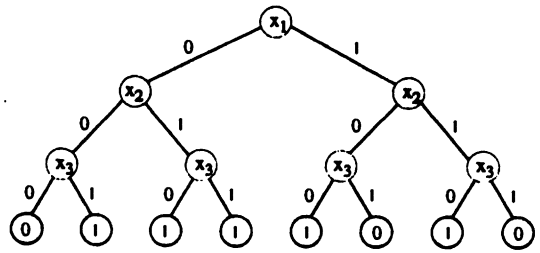


Figura 16.2 Albero corrispondente ad una computazione non-deterministica per verificare la soddisfacibilità dell'espressione Booleana $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$.

un valore binario per z facendo proseguire la computazione in una "giusta" direzione. Quindi, mentre una una computazione deterministica è descritta tramite una catena, una computazione non deterministica è descritta tramite un albero, e le ramificazioni corrispondono ai punti in cui vengono eseguite istruzioni di tipo *indovina*.

Ad esempio, un algoritmo non deterministico per il problema della soddisfacibilità di una espressione Booleana in forma normale congiuntiva potrà indovinare i valori da assegnare alle variabili, e poi verificare che l'espressione sia soddisfatta dalla particolare assegnazione. Un albero corrispondente ad una espressione con quattro variabili è mostrato in Figura 16.2: nell'esempio, l'espressione è soddisfacibile, poiché esiste almeno un'assegnazione di verità (ovvero un ramo dell'albero) che porta ad una foglia con etichetta `successo`.

Proviamo ora ad immaginare un algoritmo non deterministico per il problema delle formule Booleane quantificate: non sorprendentemente, incontreremo varie difficoltà. Per semplicità, supponiamo prima di indovinare il valore di tutte le variabili quantificate esistenzialmente: questo è facile, come nel caso di soddisfacibilità. Le variabili quantificate universalmente, però, devono essere trattate in modo diverso: tutti i valori di verità assegnati a quelle variabili devono infatti soddisfare l'espressione. Quindi non è più possibile sfruttare il non-determinismo fornito dall'istruzione *indovina* per incanalare la computazione sul giusto ramo: occorrerebbe infatti verificare che *tutte* le foglie di un intero sottoalbero (di dimensione esponenziale) abbiano etichetta `successo`. Ci imbattiamo quindi nella stessa difficoltà incontrata nella ricerca di un certificato per questo problema: non riusciamo neppure ad immnginare a un certificato di dimensione polinomiale nella dimensione dell'istanza di ingresso.

Definizione 16.4 *Data una qualunque funzione $f(n)$, chiamiamo* NTIME$(f(n))$ *l'insieme dei problemi decisionali che possono essere risolti da un algoritmo non deterministico in tempo $O(f(n))$. La classe* NP *è la classe dei problemi risolvibili in tempo polinomiale non deterministico nella dimensione n dell'istanza di ingresso:*

$$\mathrm{NP} = \bigcup_{c=0}^{\infty} \mathrm{NTIME}(n^c)$$

Che rapporto intercorre tra la precedente definizione della classe NP e il problema di verificare in tempo polinomiale (deterministico) che una stringa sia un certificato per una certo problema? I due punti di vista sono solo due diverse facce della stessa medaglia. Non è infatti restrittivo dividere un algoritmo non deterministico in due fasi, in modo tale che tutte le istruzioni *indovina* siano concentrate nella prima fase:

Fase non deterministica di costruzione: sfruttando l'istruzione *indovina*, l'algoritmo costruisce un certificato per l'istanza del problema, ovvero genera una stringa di caratteri che descrive una soluzione costruttiva del problema su una specifica istanza.

Fase deterministica di verifica: l'algoritmo, deterministicamente (cioè senza usare l'istruzione *indovina*), verifica che la stringa di caratteri sia effettivamente un certificato per l'istanza del problema, restituendo 1 se ciò è vero, e 0 altrimenti.

16.2.2 Uno sguardo alla gerarchia

È facile vedere che P⊆NP, poiché un algoritmo deterministico è un caso particolare di uno non deterministico, in cui l'istruzione *indovina* non viene mai usata. Inoltre, la fase deterministica di verifica può ovviamente essere condotta in tempo polinomiale solo se il certificato ha dimensione polinomiale: ciò suggerisce che NP⊆PSPACE. Si congettura inoltre che entrambe le inclusioni siano proprie, ovvero:

$$\mathrm{P} \subset \mathrm{NP} \subset \mathrm{PSPACE} \subset \mathrm{EXPTIME}$$

Nessuno finora è però riuscito a dimostrarlo. Caratterizzeremo nei prossimi paragrafi molti problemi che si ritiene appartengano a NP ma non a P, i cosiddetti problemi NP-*completi*. Per quanto riguarda la seconda inclusione, si ritiene che proprio il problema delle formule Booleane quantificate, che appartiene a PSPACE come abbiamo visto nel paragrafo 16.1.2, non appartenga invece a NP, suggerendo che NP⊂PSPACE. La Figura 16.3 riassume le osservazioni sulla gerarchia di complessità che abbiamo fatto finora.

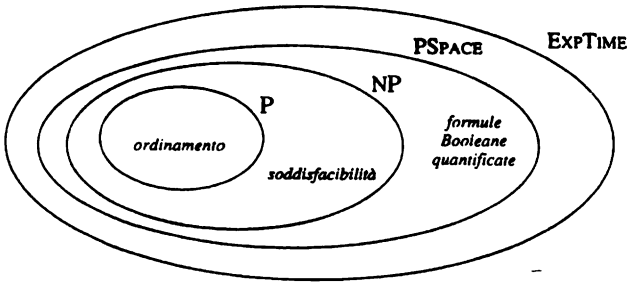


Figura 16.3 Gerarchia di complessità: si congettura che tutte le inclusioni siano proprie, ma il solo risultato di separazione dimostrato finora è P⊂EXPTIME.

La gerarchia che abbiamo abbozzato è in realtà ben più complessa e vasta. Esistono perfino problemi *indecidibili*, ovvero problemi che nessun algoritmo è in grado di risolvere, indipendentemente dalla quantità di tempo e spazio a disposizione. Un classico esempio è il *problema della fermata*: immaginiamo di aver scritto un programma e di eseguirlo su una certa istanza di ingresso. Dopo cinque minuti, il programma ancora non ha prodotto una soluzione: è entrato in un ciclo infinito o ci aspettiamo che prima o poi dia un risultato? I compilatori ed i debugger che abbiamo a disposizione purtroppo non sono (e non saranno mai) in grado di distinguere le due possibilità, perché si può dimostrare che il problema della fermata è indecidibile: non esiste alcun programma che possa sempre correttamente identificare cicli infiniti.

Per concludere, osserviamo che, sebbene definite in maniera teorica, molte classi di complessità hanno forti implicazioni pratiche. Ad esempio, la classe P rappresenta molto bene la classe dei problemi che possono essere risolti velocemente in pratica su istanze di dimensioni non banali: di solito, se un problema ammette soluzione veloce, siamo in grado di dimostrare che l'algoritmo che lo risolve ha un tempo di esecuzione polinomiale. D'altra parte, i tempi di esecuzione polinomiali che possiamo dimostrare tipicamente corrispondono a polinomi di grado basso, e quindi ad algoritmi che sono effettivamente utilizzabili: nel corso di questo libro, raramente ci siamo imbattuti in tempi di esecuzione superiori a $O(n^3)$. Nel seguito di questo capitolo studieremo in maggior dettaglio la classe NP, che sembra quindi essere una classe di problemi per cui, anche in pratica, è difficile trovare una soluzione esatta su istanze arbitrariamente grandi. Molti problemi algoritmici di grande rilevanza applicativa appartengono proprio a questa classe.

16.3 Problemi NP-completi

In questo paragrafo introdurremo la nozione di NP-*completezza*, che ci permetterà di caratterizzare i problemi più difficili all'interno della classe NP. L'espressione "più difficili" va interpretata nel seguente modo: se esistesse un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti i problemi in NP potrebbero essere risolti in tempo polinomiale e l'annosa questione $P = NP$? avrebbe risposta affermativa. Seguendo Harel [7] e parafrasando una nota filastrocca sul duca di York e sugli spostamenti delle sue truppe, potremmo scrivere a proposito dei problemi NP-completi:

E quando son su, son su
e quando son giù, son giù
a metà non posson star:
o sono su, o sono giù.

Ovvero, o i problemi NP-completi sono tutti risolvibili deterministicamente in tempo polinomiale, o nessuno di essi lo è.

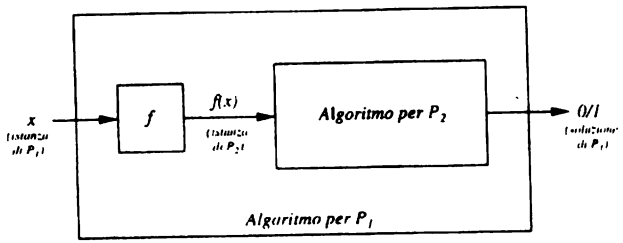


Figura 16.4 Se $\mathcal{P}_1 \prec \mathcal{P}_2$, l'uso combinato della funzione f e di un algoritmo per $\mathcal{P}_2$ permette di risolvere $\mathcal{P}_1$.

16.3.1 Riducibilità polinomiale

Al fine di caratterizzare i problemi più difficili in NP, introduciamo il concetto di *riducibilità*, che consente di stabilire formalmente quando un problema è più difficile di un altro. Esistono varie definizioni di riducibilità: quella che studieremo in questo paragrafo è stata proposta da Karp [10] e, sebbene non sia la più potente, è del tutto sufficiente per i nostri scopi.

Definizione 16.5 *Dati due problemi decisionali $\mathcal{P}_1 \subseteq I_1 \times \{0,1\}$ e $\mathcal{P}_2 \subseteq I_2 \times \{0,1\}$, diremo che $\mathcal{P}_1$ è riducibile polinomialmente a $\mathcal{P}_2$ (e scriveremo $\mathcal{P}_1 \prec \mathcal{P}_2$), se esiste una funzione $f : I_1 \rightarrow I_2$ con le seguenti proprietà:*

- *f è calcolabile in tempo polinomiale;*
- *per ogni istanza x del problema $\mathcal{P}_1$ ed ogni soluzione $s \in \{0,1\}$, $(x,s) \in \mathcal{P}_1$ se e solo se $(f(x),s) \in \mathcal{P}_2$.*

Intuitivamente, la funzione f trasforma un'istanza di $\mathcal{P}_1$ in un'istanza di $\mathcal{P}_2$ in modo tale che istanze positive di $\mathcal{P}_1$ risultino in istanze positive di $\mathcal{P}_2$ ed istanze negative di $\mathcal{P}_1$ risultino in istanze negative di $\mathcal{P}_2$. Quindi, se esistesse un algoritmo per risolvere $\mathcal{P}_2$, potremmo utilizzarlo per risolvere $\mathcal{P}_1$ come illustrato in Figura 16.4: $\mathcal{P}_2$ è dunque almeno tanto difficile quanto $\mathcal{P}_1$. Ad esempio, il problema di soddisfacibilità è polinomialmente riducibile a quello delle formule Booleane quantificate. Sia $e(x_1,...,x_n)$ l'espressione di cui vogliamo verificare la soddisfacibilità. La funzione f introduce n variabili fittizie, chiamate $y_1,...,y_n$, trasformando l'espressione e nella formula quantificata $\exists x_1 \forall y_1\ \exists x_2 \forall y_2 ... \exists x_n \forall y_n\ \ e(x_1,...,x_n)$. Poiché $y_1,...,y_n$ non sono usate in e, la scelta di una valore `true` o `false` per queste variabili non inficierà la validità dell'intera formula. La seguente proprietà è una facile conseguenza della Definizione 16.5:

Proprietà 16.1 *Se $\mathcal{P}_1 \prec \mathcal{P}_2$ e $\mathcal{P}_2 \in$P, allora $\mathcal{P}_1 \in$P.*

Osserviamo che l'algoritmo per risolvere $\mathcal{P}_1$ suggerito dalla riduzione potrebbe richiedere un tempo superiore (anche asintoticamente) a quello richiesto per risolvere $\mathcal{P}_2$, ma in questo contesto siamo solo interessati ad ottenere tempi di esecuzione polinomiali.

Ci sono alcune variazioni minori della Definizione 16.5, che possono però risultare utili. In generale, un problema $\mathcal{P}_1$ è più facile di un problema $\mathcal{P}_2$ se possiamo trovare un algoritmo per risolvere $\mathcal{P}_1$ che usa un "piccolo" namero di chiamate ad una sottoprocedura per risolvere $\mathcal{P}_2$. "Piccolo" non significa necessariamente pari a 1, come nel caso della Definizione 16.5 e della conseguente Figura 16.4: un numero polinomiale di chiamate va infatti ancora hene per poter parlare di riducibilità polinomiale. Questa osservazione può semplificare molte riduzioni. Ad esempio, consideriamo i due seguenti problemi su grafi.

Definizione 16.6 *Dati un grafo G, due nodi s e t ed un intero k, il problema del* cammino semplice lungo *richiede di verificare se esiste un cammino semplice da s a t contenente almeno k archi. Dato un grafo G, il problema del* ciclo Hamiltoniano *richiede di verificare se esiste un ciclo che visita ciascun nodo esattamente una volta.*

È facile vedere che il problema del ciclo Hamiltoniano è riducibile polinomialmente a quello del cammino semplice lungo: per verificare se un grafo $G(V,E)$ ha un ciclo Hamiltoniano, è infatti sufficiente verificare che esista un arco $(u,v) \in E$ tale che u e v sono connessi da un cammino semplice di lunghezza $k = (n-1)$. La sottoprocedura per risolvere il cammino semplice lungo è in questo caso richiamata al più $|E|$ volte, ma la riduzione rimane polinomiale. Possiamo infine definire i problemi NP-ardui e NP-completi.

Definizione 16.7 *Un problema decisionale $\mathcal{P}$ si dice* NP-*arduo se ogni problema $\mathcal{Q} \in$NP è riducibile polinomialmente a $\mathcal{P}$. Un problema decisionale $\mathcal{P}$ si dice* NP-*completo se appartiene alla classe* NP *ed è* NP-*arduo.*

Sottolineiamo che possono esistere problemi NP-ardui, ma non appartenenti alla classe NP. Concludiamo con un teorema che dà un importante contributo allo studio della domanda "P=NP?".

Teorema 16.1 *Se un qualunque problema decisionale* NP-*completo appartenesse alla classe* P, *allora risulterebbe* P=NP.

Dimostrazione. Il teorema segue dalla Definizione 16.7 e dalla Proprietà 16.1. □

Torniamo ora alla filastrocca all'inizio di questo capitolo: l'interpretazione in termini di teoria della complessità a questo punto dovrebbe essere più chiura!

16.3.2 Il teorema di Cook

L'NP-completezza va pensata come uno strumento che permette di confrontare le classi P e NP limitandosi a studiare la difficoltà di singoli problemi. Purtroppo, mentre dimostrare che un problema è in NP sembra ragionevole (abbiamo infatti visto che possiamo farlo esihendo un algoritmo polinomiale non deterministico o, ancor più semplicemente, un certificato polinomiale), dimostrare che un problema

è NP-arduo non sembra altrettanto facile. Applicare la Definizione 16.7 implicherebbe infatti dimostrare riduzioni polinomiali a partire da tutti i problemi in NP, anche quelli cbe magari non conosciamo e cbe potrebbero essere in numero infinito! La prima dimostrazione di NP-completezza aggira il problema basandosi su una diversa idea. Essa ha portato al seguente teorema, dovuto a Steven Cook [4], che rappresenta una delle pietre miliari della teoria della NP-completezza:

Teorema 16.2 (Teorema di Cook) *Il problema della soddisfacibilità di espressioni Booleane in forma normale congiuntiva è* NP-*completo.*

Una dimostrazione formale richiederebbe l'introduzione di alcuni nuovi concetti che esulano dallo scopo di questo libro. Ci limitiamo quindi alla seguente osservazione: Cook ha mostrato un algoritmo che, dati un qualunque problema $\mathcal{P}$ (che vogliamo ridurre a soddisfacibilità) ed una qualunque istanza x per $\mathcal{P}$, costruisce una espressione Booleana in forma normale congiuntiva che descrive il calcolo di un algoritmo non deterministico per risolvere $\mathcal{P}$ su x. L'espressione, molto lunga e complessa, è vera se e solo se l'algoritmo restituisce 1. A titolo di esempio, mostreremo ora l'esistenza di un problema NP-completo in modo non del tutto formale, ma più semplice della dimostrazione di Cook. Il problema che prenderemo in considerazione è il problema della fermata limitata, definito nel seguente modo:

Definizione 16.8 *Dati un programma X ed un intero k, il problema della* fermata limitata *richiede di verificare se esistono dei dati che, passati in ingresso a X, ne causano la fermata dopo al più k passi.*

Notiamo che questo problema è una variazione del problema della fermata di cui abbiamo parlato nel Paragrafo 16.2.2, che risulta però indecidibile: l'unica – enorme – differenza è nella specifica della limitazione k al numero di passi entro cui si richiede la convergenza. Mostreremo ora che la fermata limitata è un problema NP-completo. Innanzitutto, osserviamo che l'appartenenza a NP è facile da dimostrare: un certificato può verificare che certi dati rappresentano una soluzione al problema della fermata limitata simulando il programma X su quei dati per k passi. Ciò richiede tempo polinomiale in k e nella lunghezza del programma: al fine di poter affermare che il certificato richiede tempo polinomiale nella dimensione totale dell'istanza di ingresso, dobbiamo quindi assumere che k sia specificato in notazione unaria, così che la lunghezza della parte dell'istanza che rappresenta k sia k stesso, e non $\log k$. Si tratta però di una limitazione tecnica, che non inficia la verità del risultato. Dobbiamo ora dimostrare che la fermata limitata è un problema NP-arduo, ovvero che ogni problema $\mathcal{P} \in$NP è riducibile polinomialmente ad esso. Se $\mathcal{P} \in$NP, allora esiste un certificato polinomiale C per $\mathcal{P}$: C verifica che una certa stringa di lunghezza n rappresenti una soluzione costruttiva per $\mathcal{P}$ in tempo $p(n)$. È facile modificare C in un programma C' che entra in un ciclo infinito ogni qualvolta la risposta di C sia negativa. Ma allora, se sapessimo risolvere la fermata limitata, potremmo risolvere $\mathcal{P}$ richiamando come subroutine l'algoritmo per la fermata limitata con parametri $X = C'$ e $k = p(n)$.

La risposta della subroutine sarà positiva se e solo se esistono dei dati su cui C' converge in al più $p(n)$, e ciò può avvenire se e solo se questi dati sono una soluzione costruttiva per $\mathcal{P}$, ovvero se l'istanza di $\mathcal{P}$ ammette soluzione! Poiché $\mathcal{P} \prec$ fermata limitata, quest'ultimo problema risulta NP-arduo.

Fortunatamente, la maggior parte delle dimostrazioni di NP-completezza non si basano su questo tipo di ragionamento, che sarebbe troppo complicato per caratterizzare la complessità di altri problemi. Piuttosto, esse sfruttano la transitività delle riduzioni polinomiali:

Proprietà 16.2 *Se $\mathcal{P}_1 \prec \mathcal{P}_2$ e $\mathcal{P}_2 \prec \mathcal{P}_3$, allora $\mathcal{P}_1 \prec \mathcal{P}_3$.*

Vedremo nel prossimo paragrafo alcune dimostrazioni di NP-completezza basate su questa idea.

16.3.3 Dimostrazioni di NP-completezza

Dopo il teorema di Cook, un secondo lavoro fondamentale per la teoria della NP-completezza fu dovuto a Richard Karp, che mostrò ingegnose riduzioni tra problemi apparentemente scorrelati, dimostrandone l'NP-completezza per transitività. Innanzitutto, definiamo il seguente problema:

Definizione 16.9 *Data una espressione in forma normale congiuntiva in cui ogni clausola consiste di 3 letterali, il problema della* 3-soddisfacibilità *richiede di verificare se esiste una assegnazione di valori di verità alle variabili che rende l'espressione vera.*

Si tratta di un caso particolare del problema di soddisfacibilità, in cui il numero di letterali in ogni clausola è limitato a tre. Nonostante questa limitazione, è possibile dimostrare che il problema non diventa più semplice, ovvero che rimane NP-completo. Perché interessarsi di questa variazione? Grazie al teorema di Cook, molte fondamentali dimostrazioni di NP-completezza si basano su riduzioni che partono dal problema della soddisfacibiltà. Tali riduzioni sono tanto più semplici quanto più strutturate sono le clausole iniziali, e dunque ridurre a partire da 3-soddisfacibilità potrebbe essere molto più facile che non partendo da clausole lunghe quanto si voglia. Prima di mostrare che 3-soddisfacibilità è un problema NP-completo, diamo un risultato tecnico preliminare.

Lemma 16.1 *Siano σ, ρ, ed τ tre espressioni Booleane, e sia $\phi = (\sigma \Leftrightarrow \rho \vee \tau)$. Allora ϕ può essere riscritta come*

$$(\sigma \vee \rho \vee \overline{\tau}) \wedge (\sigma \vee \overline{\rho} \vee \tau) \wedge (\sigma \vee \overline{\rho} \vee \overline{\tau}) \wedge (\overline{\sigma} \vee \rho \vee \tau)$$

dove $\overline{e}$ denota la negazione dell'espressione e.

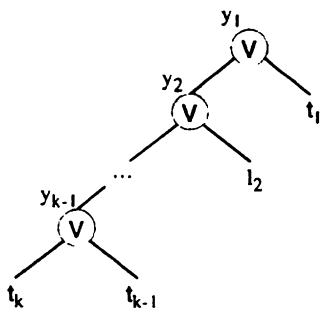


Figura 16.5 Trasformazione di una clausola di lunghezza k in una congiunzione di clausole di lunghezza 3.

Dimostrazione. Ricordando che $a \Rightarrow b$ equivale a $\overline{a} \vee b$, disegnamo una tabella di verità che mostra il valore di ϕ per ogni possibile valore di σ, ρ e τ:

σ	ρ	τ	$\phi = (\sigma \Leftrightarrow \rho \vee \tau)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Osservando le configurazioni per cui $\phi = 0$, possiamo scrivere

$$\overline{\phi} = (\overline{\sigma} \wedge \overline{\rho} \wedge \tau) \vee (\overline{\sigma} \wedge \rho \wedge \overline{\tau}) \vee (\overline{\sigma} \wedge \rho \wedge \tau) \vee (\sigma \wedge \overline{\rho} \wedge \overline{\tau})$$

Per concludere la dimostrazione basta ora negare l'espressione ed applicare le Leggi di De Morgan. □

Teorema 16.3 *Il problema della 3-soddisfacibilità di espressioni Booleane in forma normale congiuntiva è* NP-*completo.*

Dimostrazione. L'appartenenza a NP può essere dimostrata come nel caso di soddisfacibilità (vedi Paragrafo 16.1.2). Per dimostrare che 3-soddisfacibilità è NP-arduo, usiamo una riduzione da soddisfacibilità. L'unico problema deriva dalla lunghezza delle clausole nell'espressione di partenza, che potrebbero contenere strettamente più o meno di 3 letterali. Considereremo quindi le clausole una alla volta, mostrando come trasformarle in congiunzioni di clausole di lunghezza 3.

Sia $C = l_1 \vee l_2 \vee l_3 ... \vee l_k$ una clausola di lunghezza $k \geq 4$, in cui $l_1 ... l_k$ rappresentano letterali. La clausola può essere rappresentata come in Figura 16.5. Ogni operatore $\vee$ corrisponde ad un nodo interno dell'albero, le cui foglie sono i letterali. Associando ad ogni nodo interno una variabile ausiliaria y_i, è facile convincersi che ϕ può essere riscritta come

$$\phi = y_1 \wedge (y_1 \Leftrightarrow y_2 \vee l_1) \wedge (y_2 \Leftrightarrow y_3 \vee l_2) \wedge ... \wedge (y_{k-1} \Leftrightarrow l_k \vee l_{k-1})$$

Usando il Lemma 16.1, possiamo ulteriormente riscrivere ϕ come congiunzione di clausole contenenti al più tre letterali.

Una qualunque clausola $C = l_1 \vee l_2$ di lunghezza 2 può essere riscritta come $(l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \overline{x})$, dove x è una variabile ausiliaria. Una qualunque clausola $C = l_1$ di lunghezza 1 può essere riscritta come $(l_1 \vee x \vee y) \wedge (l_1 \vee x \vee \overline{y}) \wedge (l_1 \vee \overline{x} \vee y) \wedge (l_1 \vee \overline{x} \vee \overline{y})$, dove x e y sono variabili ausiliarie.

L'espressione ottenuta è in forma normale congiuntiva ed è algebricamente equivalente a quella originaria. Inoltre, il numero di clausole di lunghezza 3 ottenute dall'espressione originaria è polinomiale nella lunghezza dell'espressione, e pertanto l'intera trasformazione richiede tempo polinomiale. □

Usando il problema della 3-soddisfacibilità, Karp dimostrò che numerosi altri problemi sono NP-completi. Consideriamo, ad esempio, un classico problema su grafi.

Definizione 16.10 *Una* clique *di dimensione t è un grafo di t nodi che sono tutti connessi da coppie da archi. Il problema della clique richiede di verificare, dati un grafo G ed un intero k, se G contiene una clique di dimensione almeno k.*

Teorema 16.4 *Il problema della clique è* NP-*completo.*

Dimostrazione. Un certificato per il problema della clique è un sottografo i cui vertici sono tutti adiacenti tra loro: è facile verificare tale certificato in tempo polinomiale, e questo dimostra che il problema appartiene a NP.

Per dimostrare che il problema è NP-arduo, mostriamo una riduzione da 3-soddisfacibilità, che abbiamo visto essere NP-completo in base al Teorema 16.3. Consideriamo una espressione Booleana e in forma normale congiuntiva: sia k il numero di clausole e siano l_1^i, l_2^i e l_3^i i 3 letterali nella i-esima clausola c_i. Costruiamo da e un grafo G come segue. Per ogni clausola, G contiene tre nodi corrispondenti ai tre letterali nella clausola. Due nodi sono connessi da un arco se i corrispondenti letterali sono in clausole diverse e non sono l'uno la negazione dell'altro. Un esempio è mostrato in Figura 16.6.

Mostriamo ora che e è soddisfacibile se e solo se G contiene una clique di k nodi. Consideriamo prima la condizione sufficiente. Se e è soddisfacibile, almeno un letterale in ciascuna clausola è assegnato valore `true`. Consideriamo esattamente un letterale vero per clausola ed osserviamo i vertici di G corrispondenti ai letterali scelti. Tali vertici sono tutti connessi a coppie, poichè corrispondono a letterali in clausole diverse e che non sono l'uno la negazione dell'altro: altrimenti non potrebbero essere contemporaneamnte veri! Abbiamo quindi individuato

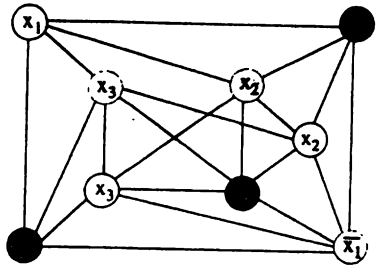


Figura 16.6 NP-completezza del problema della clique: grafo corrispondente all'espressione Booleana $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$. Il colore di un vertice denota a quale clausola corrisponde: bianco per la prima clausola, grigio scuro per la secoda e grigio chiaro per la terza.

una clique in G di dimensione k. La dimostrazione della condizione necessaria è simile ed è lasciata come esercizio per il lettore. □

Il problema della clique è utile a sua volta per dimostrare l'NP-completezza di molti altri problemi su grafi, come, ad esempio, l'insieme indipendente.

Definizione 16.11 *Dato un grafo G, un* insieme indipendente *di dimensione t è un insieme di t nodi di G nessuno dei quali è connesso con gli altri tramite un arco. Il problema dell'insieme indipendente richiede di verificare, dati un grafo G ed un intero k, se G contiene un insieme indipendente di dimensione almeno k.*

Teorema 16.5 *Il problema dell'insieme indipendente è* NP-*completo.*

Dimostrazione. Verificare l'appartenenza a NP è un semplice esercizio. Per dimostrare che insieme indipendente è NP-arduo, usiamo una riduzione dal problema della clique, NP-completo in base al Teorema 16.4. Dato un grafo G, costruiamo un grafo $\overline{G}$ con gli stessi nodi e l'insieme complementare di archi: ovvero $\overline{G}$ contiene un arco (u, v) se e solo se (u, v) non è un arco in G. È facile vedere che un insieme di nodi definisce una clique in G se e solo se definisce un insieme indipendente in $\overline{G}$. Quindi G ammette una clique di dimensione k se e solo se $\overline{G}$ ammette un insieme indipendente di dimensione k. È inoltre immediato verificare che $\overline{G}$ può essere costruito in tempo polinomiale. □

I problemi NP-completi ammontano ormai a varie centinaia: tra essi ritroviamo, ad esempio, i problemi del ciclo Hamiltoniano e del cammino semplice lungo introdotti nel Paragrafo 16.3.1. Di seguito ne citiamo altri tra i più importanti, rimandando al libro di Garey e Johnson [6], il testo di riferimento sull'NP-completezza, per un elenco molto più dettagliato.

- *Copertura di vertici:* una copertura di vertici (in inglese, *vertex cover*) di un grafo $G = (V, E)$ è un insieme di nodi $C \subseteq V$ tale che, per ogni $(u, v) \in E$, almeno uno tra u e v appartiene a C. Il problema della copertura di vertici richiede di verificare se, dati un grafo G ed un intero k, esiste una copertura di vertici di G di dimensione al più k.
- *Feedback arc set:* un feedback arc set di un grafo orientato $G = (V, E)$ è un insieme di archi $F \subseteq E$ tale che il grafo $G' = (V, E \setminus F)$ non contiene cicli. Il problema del feedback arc set richiede di verificare se, dati un grafo orientato G ed un intero k, esiste un feedback arc set di G di dimensione al più k.
- *Commesso viaggiatore:* dati un grafo completo $G = (V, E)$ con pesi w sugli archi ed un intero k, il problema del commesso viaggiatore richiede di verificare se esiste un ciclo di peso al più k che attraversa ogni nodo una ed una sola volta.
- *Colorazione:* dati un grafo $G = (V, E)$ ed un intero k, il problema della colorazione richiede di verificare se esiste una k-colorazione dei nodi di G, ovvero una funzione $c : V \to \{1, ..., k\}$ tale che $c(u) \neq c(v)$ se $(u, v) \in E$.
- *Somme di sottoinsiemi:* dati un insieme S di numeri naturali ed un intero t, il problema delle somme di sottoinsiemi richiede di verificare se esiste un sottoinsieme di S i cui elementi sommano esattamente a t.
- *Zaino:* dati un numero intero k, uno zaino di capacità c, e n oggetti di dimensioni $s_1 ... s_n$ cui sono associati profitti $p_1, ... p_n$, il problema dello zaino richiede di verificare se esiste un sottoinsieme degli oggetti di dimensione $\leq c$ che garantisca un profitto $\geq k$.
- *Sequenziamento di lavori con penalità:* siano dati un intero k ed n lavori, con rispettivi tempi di esecuzione $t_1, ... t_n$, scadenze $d_1, ... d_n$, e penalità $p_1, ... p_n$. I lavori devono essere tutti eseguiti, ma uno alla volta. Un sequenziamento è un ordine di esecuzione (ovvero una permutazione) per i lavori. Se l'i-esimo lavoro nella permutazione supera la scadenza, si incorre nella relativa penalità. Il problema del sequenziamento di lavori richiede di verificare se esiste un sequenziamento dei lavori con penalità totale $\leq k$.

16.4 Algoritmi di approssimazione

In questo paragrafo ci soffermeremo sulla soluzione di problemi di ottimizzazione. Abbiamo visto nel Paragrafo 16.1.1 che possiamo associare a un problema di ottimizzazione una corrispondente versione decisionale che non è più difficile da risolvere del problema stesso. Ad esempio, un tipico problema di ottimizzazione richiede di trovare una copertura di vertici di minima cardinalità. La corrispondente versione decisionale è stata introdotta nel Paragrafo 16.3.3 ed è formulata come segue: dati un grafo G ed un intero k, esiste una copertura di vertici di cardinalità al più k?

Una interessante domanda è la seguente: cosa fare nel caso si renda necessario risolvere un problema di ottimizzazione la cui versione decisionale sia NP-completa? Sperare di poter trovare un algoritmo polinomiale non sembra ragionevole, poiché l'evidenza suggerisce che P≠NP. A volte, avere una soluzione esatta non è però strettamente necessario: una soluzione che non si discosti troppo da

quella ottima potrebbe comunque risultare utile, e magari potremmo anche sperare che sia più facile da calcolare! La teoria dell'approssimazione formalizza questa semplice idea.
Dato un problema di ottimizzazione, esistono in generale numerose *soluzioni ammissibili* per quel problema, ma tipicamente non tutte sono ottime. Ad esempio, consideriamo il problema della colorazione di un grafo: una soluzione ammissibile consiste nell'assegnare un colore diverso ad ogni nodo, usando pertanto n colori. È facile vedere che questa soluzione è ottima se e solo se il grafo è completo: se infatti mancasse almeno un arco, diciamo (u, v), i nodi u e v potrebbero essere assegnati con lo stesso colore, usando pertanto meno di n colori. Vediamo innanzitutto come misurare la distanza di una qualunque soluzione ammissibile da una soluzione ottima.

Definizione 16.12 *Sia $\mathcal{P}$ un problema di ottimizzazione. Diciamo che un algoritmo di approssimazione per $\mathcal{P}$ ha fattore di approssimazione $\rho(n)$ se il costo C della soluzione prodotta dall'algoritmo su una qualunque istanza di dimensione n differisce di un fattore moltiplicativo $\leq \rho(n)$ dal costo C^* di una soluzione ottima:*

$$max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

Osserviamo che la Definizione 16.12 definisce il fattore di approssimazione per problemi sia di minimizzazione, come la colorazione, che di massimizzazione, come la clique. Infatti, nel caso di problemi di minimizzazione, avremo $C \geq C^*$ e $max(C/C^*, C^*/C) = C/C^*$. La disuguaglianza diventa quindi $C \leq C^* \cdot \rho(n)$: chiaramente, se $\rho(n) = 1$, C è una soluzione ottima, mentre in generale C sarà tanto peggiore quanto più $\rho(n)$ si allontana dal valore 1. Nel caso di problemi di massimizzazione si ha una situazione analoga. Poichè non esistono soluzioni ammissibili con valore migliore delle soluzioni ottime, possiamo dedurne che il fattore di approssimazione è sempre ≥ 1 e che gli algoritmi esatti sono quelli con fattore di approssimazione pari a 1.

Osserviamo inoltre che il fattore di approssimazione $\rho(n)$ è in generale una funzione della dimensione n dell'istanza di ingresso. Esistono tuttavia problemi per cui è possibile ottenere fattori di approssimazione costanti: vedremo un semplice esempio relativo al problema della copertura di vertici. Nonostante l'algoritmo che presenteremo sia quasi banale, dimostreremo che esso garantisce sempre di trovare una soluzione che contiene al più il doppio dei nodi in una soluzione ottima.

Ricordiamo che il problema della copertura di vertici richiede di ricoprire tutti gli archi di un grafo tramite nodi: per ogni arco (u, v), almeno uno tra i nodi u e v deve essere nella copertura di vertici. Potremmo allora scegliere un qualunque arco (u, v) ed aggiungere entrambi gli estremi all'insieme: in questo modo copriremmo non solo (u, v), ma anche tutti gli archi incidenti a u ed a v. Possimo quindi cancellare questi archi dal grafo, ed iterare il ragionamento finché il grafo non diventa vuoto. Lo pseudocodice è dato in Figura 16.4. Se il grafo è rappresentato tramite liste di adiacenza, l'intero algoritmo richiede tempo


```
   algoritmo vertexCover(grafo G(V,E)) → insieme di vertici
1.   C = ∅
2.   while ( E ≠ ∅ ) do
3.     sia (u,v) un arco di G
4.     C = C ∪ {u,v}
5.     for each arco e = (u,w) o e = (v,w) do
6.       E = E \ {e}
7.   return C
```

Figura 16.7 Un semplice algoritmo di approssimazione per il problema della copertura di vertici.

$O(n+m)$, dove n ed m sono il numero di nodi e di archi, rispettivamente. Il fatto che C sia una copertura di vertici è garantito dal fatto che l'algoritmo non termina finché E non diventa vuoto e dal fatto che un arco è rimosso dal grafo solo se è coperto. Dimostriamo ora il fattore di approssimazione.

Teorema 16.6 *L'algoritmo* `vertexCover` *ha un fattore di approssimazione* 2.

Dimostrazione. Sia R l'insieme di archi scelti dall'algoritmo alla riga 3 durante l'intera esecazione dell'algoritmo: la scelta di (u, v) implica che tutti gli archi incidenti a u e v sono rimossi dal grafo, e quindi nessuna coppia di archi in R può avere un estremo in comune. Per coprire gli archi in R tramite nodi, sono pertanto necessari almeno $|R|$ nodi distinti. Poiché entrambi gli estremi degli archi selezionati nella linea 3 sono aggiunti a C, risulta $|C| = 2|R|$, da cui segue il fattore di approssimazione 2. □

Esistono naturalmente algoritmi di approssimazione molto più sofisticati e basati su tecniche algoritmiche molto più complesse, la cui trattazione esula dallo scopo di questo libro. Ci limitiamo ad osservare in questa sede che dal punto di vista del fattore di approssimazione ottenuto lo scenario è estremamente vario. Non solo, come nel caso della copertura di vertici, è possibile dimostrare fattori di approssimazione costanti, ma esistono anche problemi che possono essere approssimati ottenendo un qualunque fattore di approssimazione $\epsilon > 0$, al prezzo di tempi di esecuzione più alti (tipicamente, inversamente proporzionali ad ϵ). Sfortunatamente, esistono anche problemi che sono difficili persino da approssimare, come la colorazione: è stato infatti dimostrato che se esistesse un algoritmo di approssimazione per il problema della colorazione con fattore di approssimazione $\leq 4/3$, allora risulterebbe P=NP.

16.5 Problemi

Problema 16.1 Dimostrare che i problemi elencati al termine del Paragrafo 16.3.3

sono in NP, progettando algoritmi non deterministici polinomiali per risolverli o algoritmi deterministici polinomiali per verificarne un certificato.

Problema 16.2 (*) Dimostrare che il problema della copertura di vertici definito nel Paragrafo 16.3.3 è NP-completo. Per dimostrare che è NP-arduo, ti consigliamo di usare una riduzione polinomiale dal problema della clique e di studiare il nesso tra una clique in un grafo G e una copertura di vertici nel grafo complementare $\overline{G}$. Ricordiamo che $\overline{G}$ ha lo stesso insieme di vertici di G, ma contiene un arco (a, b) se e solo se $(a, b) \notin G$. Dimostrare inoltre che, nel caso in cui il grafo sia un albero, la minima copertura di vertici può essere trovata in tempo polinomiale.

Problema 16.3 (*) Il problema della 2-soddisfacibilità è così definito: data una espressione in forma normale congiuntiva in cui ogni clausola consiste di 2 letterali, esiste una assegnazione di valori di verità alle variabili che rende l'espressione vera? Diversamente da 3-soddisfacibilità, tale problema non è NP-arduo: progettare un algoritmo *polinomiale* per risolverlo.

Problema 16.4 Dimostrare che il problema del ciclo Hamiltoniano su grafi orientati è riducibile polinomialmente al problema del ciclo Hamiltoniano su grafi non orientati.

Problema 16.5 Dimostrare che il problema del ciclo Hamiltoniano su grafi orientati è riducibile polinomialmente al problema del commesso viaggiatore definito nel Paragrafo 16.3.3.

Problema 16.6 Il problema della colorazione definito nel Paragrafo 16.3.3 è NP-completo. Dimostrare che è però possibile verificare in tempo polinomiale se una grafo può essere colorato usando solo 2 colori. In particolare, progettare un algoritmo che, nel caso di risposta affermativa, sia in grado di esibire una 2-colorazione. L'algoritmo deve avere tempo di esecuzione $O(n + m)$, dove n e m sono rispettivamente il numero di nodi e di archi del grafo.

Problema 16.7 Mostrare che l'analisi del fattore di approssimazione dell'algoritmo per la copertura di vertici descritto nel Paragrafo 16.4 è stretta, esibendo una famiglia infinita di grafi per cui l'algoritmo restituisce una soluzione con valore doppio rispetto a quella ottima.

16.6 Sommario

In questo capitolo abbiamo introdotto il concetto di classe di complessità, che permette di classificare diversi problemi di calcolo in base alla quantità di tempo o spazio sufficiente per risolverli. Abbiamo poi introdotto varie classi di complessità, derivando una intera gerarchia:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

In particolare, ci siamo soffermati sulla classe NP dei problemi risolubili in tempo polinomiale non deterministico, ed abbiamo individuato un sottoinsieme di questi problemi, detti NP-completi, almeno tanto difficili quanto tutti gli altri problemi in NP. Se sapessimo risolvere in tempo polinomiale deterministico uno solo dei problemi NP-completi, allora sapremmo risolvere tutti i problemi in NP. Si congettura però che ciò non sia possibile, ovvero che non esistano algoritmi polinomiali deterministici per i problemi NP-completi.

Sottolineiamo che la classe P dei problemi che ammettono algoritmi polinomiali deterministici è ritenuta come la classe dei problemi trattabili. Quindi, non potendo presumibilmente sperare di risolvere in maniera esatta istanze di grandi dimensioni di problemi NP-ardui, spesso ci si accontenta di ottenere soluzioni approssimate, ovvero soluzioni non ottime ma che non si discostino troppo dall'ottimo.

16.7 Note bibliografiche

Basandosi su uno specifico modello di calcolo, la macchina di Turing di cui abbiamo parlato nel Paragrafo 2.1 del Capitolo 2, Hartmanis e Stearns [8] definirono il concetto di classe di complessità. Successivamente, Blum [2] svincolò la teoria della complessità dal modello. La classe P fu introdotta da Cobham [3] e, indipendentemente, da Edmonds [5]. Edmonds, già nel 1965, presentò anche la classe NP, congetturando che P$\neq$ NP: vale la pena sottolineare nuovamente che, dopo tanti anni di intense ricerche, questo problema non è stato ancora risolto.

Il concetto di NP-completezza fu introdotto da Cook [4], che dimostrò che il problema della soddisfacibilità di una formula Booleana in forma normale congiuntiva è NP-completo, ed indipendentemente da Levin [11], che diede una dimostrazione di NP-completezza per un problema di tassellazione del piano. Karp descrisse in [10] l'idea di riduzione polinomiale, che usò per dimostrare la NP-completezza di alcuni problemi fondamentali, tra cui clique e copertura di vertici. Il libro di Garey e Johnson [6] è il testo di riferimento fondamentale riguardante la teoria della NP-completezza, e presenta un lungo elenco di problemi NP-completi.

La teoria e le tecniche di progettazione degli algoritmi di approssimazione sono discussi, ad esempio, in [1, 9, 12]. Papadimitriou Steiglitz attribuiscono l'algoritmo di approssimazione per il problema della copertura di vertici che abbiamo descritto nel Paragrafo 16.4 a F. Gavril e M. Yannakakis.

Riferimenti bibliografici

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, *Complexity and approximation: combinatorial optimization problems and their approximability properties*, Springer, 1999.

[2] M. Blum, "Machine independent theory of the complexity of recursive functions", *Journal of the Association for Computing Machinery*, 14, 1967.

[3] A. Cobham, "The intrinsic computational difficulty of functions", *Proceedings 1964 Congress for Logic, Methodology, and the Philosophy of Science*, 24–30, North-Holland, 1964.
[4] S. Cook, "The complexity of theorem proving procedures", *Proceedings 3rd Annual ACM Symposium on Theory of Computing*, 151–158, 1971.
[5] J. Edmonds, "Paths, trees and flowers", *Canadian Journal of Mathematics*, 17:449–467, 1965.
[6] M. R. Garey, D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, Freeman and Company, 1979.
[7] David Harel, *Computer a responsabilità limitata: dove le macchine non riescono ad arrivare*, Giulio Einaudi editore (2002), Torino.
[8] J. Hartmanis e R. E. Stearns, "On the computational complexity of algorithms", *Transactions of the American Mathematical Society*, 117: 285–306, 1965.
[9] D. S. Hochbaum, *Approximation algorithms for NP-hard problems*, PWS Publishing Company, Boston, MA, 1997.
[10] R. M. Karp, "Reducibility among combinatorial problems", *Complexity of Computer Computations*, R. Miller and J. Thatcher, eds., Plenum Press, New York, 85–104, 1972.
[11] L. A. Levin, "Universal sorting problems", *Problemy Peredachi Informatsii*, in russo, 9(3):265–266, 1973.
[12] C. Papadimitriou e K. Steiglitz, *Combinatorial optimization: algorithms and complexity*, Prentice Hall, 1982.

17

Appendice

Quando chel cubo con le cose appresso
se eguaglia a qualche numero discreto
trovan due altri differenti in esso.
Dappoi terrai questo per consueto
che il lor produtto sia sempre eguale
al terzo cubo delle cose neto.
El residuo poi suo generale
delli lor cubi ben sottratti
varrà la tua cosa principale.

(Nicolò Tartaglia[1])

Come abbiamo visto nel corso del libro, l'analisi di algoritmi si avvale continuamente di strumenti matematici. In questa appendice, che non ha alcuna pretesa di completezza e di generalità, richiameremo quindi quelle nozioni di analisi, calcolo della probabilità, calcolo combinatorio e teoria dei grafi che abbiamo utilizzato nel testo. Cercheremo di dimostrare alcuni dei principali risultati in maniera autocontenuta, assumendo comunque che il lettore abbia familiarità con elementi basilari di analisi matematica e calcolo, quali semplici regole di derivazione e di integrazione. Rimandiamo invece al Capitolo 2 per una trattazione accurata delle metodologie e degli strumenti più specificamente legati all'analisi di algoritmi, come le notazioni O, Ω e Θ ed i metodi per risolvere equazioni di ricorrenza derivanti da algoritmi ricorsivi.

17.1 Logaritmi e numero di Nepero

Per ogni $b > 1$ e $x > 0$, $\log_b x$ è la potenza cui la base b deve essere elevata per ottenere x, ovvero $\log_b x$ è il numero reale ℓ tale che $b^\ell = x$. Tipicamente, le notazioni $\log x$ e $\ln x$ (logaritmo naturale di x) sottintendono che la base sia

[1] Questa terza rima, che risale al 12 febbraio 1535, descrive la soluzione dell'equazione $x^3 + px = q$: il *cubo*, le *cose appresso*, e il *numero discreto* sono rispettivamente x^3, px e q.

uguale a 10 e al numero di Nepero e, rispettivamente. Il logaritmo è una funzione strettamente crescente e, per ogni base, risulta $\log_b 1 = 0$. Le seguenti proprietà seguono facilmente dalla definizione di logaritmo.

Proprietà 17.1 (Cambiamento di base) *Per ogni $x > 0$ e $b, c > 1$ risulta:*

$$\log_b a = \frac{\log_c a}{\log_c b} \tag{17.1}$$

Proprietà 17.2 (Logaritmo di un prodotto) *Il logaritmo di un prodotto può essere espresso come somma dei logaritmi, ovvero per ogni $x, y > 0$ e $b > 1$ risulta:*

$$\log_b(x \cdot y) = \log_b x + \log_b y \tag{17.2}$$

Proprietà 17.3 (Logaritmo di una potenza) *Per ogni numero reale a, $x > 0$ e $b > 1$ risulta:*

$$\log_b(x^a) = a \log_b x \tag{17.3}$$

Dall'Equazione 17.3 segue che $\log_b(b^a) = a$. Notiamo inoltre che se $\log_b x = \log_b y$ allora $x = y$, da cui si ottiene la seguente proprietà:

Proprietà 17.4 *Per ogni $x, y > 0$ e $b > 1$ risulta:*

$$x^{\log_b y} = y^{\log_b x} \tag{17.4}$$

Dimostrazione. Applicando il logaritmo in base b al primo membro dell'uguaglianza da dimostrare ed usando l'Equazione 17.3 si ottiene:

$$\log_b(x^{\log_b y}) = \log_b y \log_b x = \log_b x \log_b y = \log_b(y^{\log_b x})$$

L'enunciato segue dal fatto che $\log_b w = \log_b z$ implica $w = z$. □

Esiste infine uno stretto legame tra le funzioni $\ln x$ e $1/x$.

Proprietà 17.5 (Derivata) *La derivata di $\ln x$ è $1/x$.*

Il logaritmo naturale di un valore c è quindi l'integrale definito nell'intervallo $[1, c]$ della funzione $1/x$, ed è interpretabile come l'area sottostante il grafico di $1/x$ come mostrato in Figura 17.1. Usando la regola del cambiamento di base, la Proprietà 17.5 e semplici regole di derivazione, è anche facile verificare che la derivata di $\log_b x$ è $\log_b e/x$. La base del logaritmo naturale, ovvero il numero di Nepero $e = 2.71828...$ è inoltre il valore approssimato di $(1 + 1/n)^n$, per n molto grande. Più precisamente, per ogni $x \in \mathbb{R}$, vale il seguente limite notevole:

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x \tag{17.5}$$

dove $(1 + x/n)^n$ converge a e^x per difetto. Il numero di Nepero e interviene anche in una formula che risulta di grande utilità in numerose questioni sia teoriche che pratiche, ovvero la *formula di De Moivre-Stirling*, che vedremo nel Paragrafo 17.4.

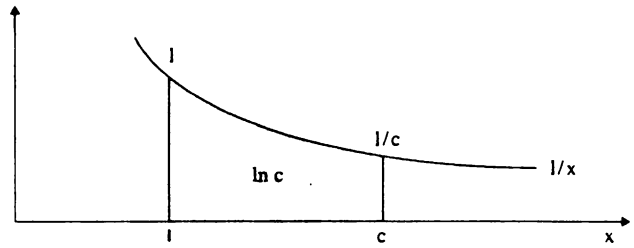


Figura 17.1 Il logaritmo naturale di un valore c è l'integrale definito nell'intervallo $[1, c]$ della funzione $1/x$.

17.2 Serie e successioni

Le serie e successioni numeriche che descriveremo in questo paragrafo si incontrano frequentemente nell'analisi di algoritmi.

17.2.1 Serie aritmetica e geometrica

Lemma 17.1 (Serie aritmetica) *La somma dei primi n numeri consecutivi, detta serie aritmetica, ha il seguente valore:*

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{17.6}$$

Dimostrazione. Sia

$$S_n = \sum_{i=1}^{n} i$$

Scriviamo:

$$\begin{aligned} S_n &= 1 + 2 + \ldots + n \\ S_n &= n + (n-1) + \ldots + 1 \end{aligned}$$

Sommando membro a membro otteniamo $2S_n = (n+1)n$ e quindi $S_n = n(n+1)/2$. □

Si narra che Gauss abbia derivato la precedente dimostrazione all'età di sei anni: per tenere impegnati gli alunni, sembra infatti che la maestra avesse assegnato il compito di calcolare il valore della somma dei primi cento numeri. Mentre i compagni erano ancora impegnati in laboriosi calcoli (sommando progressivamente l'i-esimo numero alla somma dei primi $i - 1$), Gauss consegnò il risultato quasi subito: immaginiamo che la maestra pensò che si trattasse di uno scherzo, ma fu poi costretta ad arrendersi davanti alla correttezza di un procedimento così elegante!

Lemma 17.2 (Serie geometrica) *La serie geometrica di ragione q, i cui addendi sono caratterizzati dall'avere una base costante $q \neq 1$ ed un esponente variabile, ha il seguente valore:*

$$\sum_{i=0}^{k} q^i = \frac{q^{k+1} - 1}{q - 1} \tag{17.7}$$

Dimostrazione. Sia

$$S_k(q) = \sum_{i=0}^{k} q^i$$

Moltiplicando entrambi i membri per q si ha:

$$q\, S_k(q) = \sum_{i=l}^{k+l} q^i$$

Sottraendo poi $S_k(q)$ ad entrambi i membri otteniamo

$$(q-1)\, S_k(q) = \sum_{i=l}^{k+1} q^i - \sum_{i=0}^{k} q^i = q^{k+l} - 1$$

da cui

$$S_k(q) = (q^{k+1} - 1)/(q - 1)$$

come volevamo dimostrare. □

Quando la sommatoria è infinita e $|q| < 1$, si ha la seguente utile uguaglianza:

$$\sum_{i=0}^{\infty} q^i = \frac{1}{q-1} \tag{17.8}$$

La serie geometrica è anche utile per calcolare il valore di altre sommatorie, come mostrato nel seguente esempio in cui useremo una tecnica basata sulla *divisione della sommatoria* in parti calcolabili in modo più semplice.

Esempio 17.1 Vale la seguente uguaglianza:

$$\sum_{i=1}^{k} i\, 2^{i-1} = (k-1)\, 2^k + 1 \tag{17.9}$$

Per dimostrarlo, osserviamo che

$$\sum_{i=1}^{k} i\, 2^{i-1} = \frac{1}{2} \sum_{i=1}^{k} i\, 2^i$$

e calcoliamo il valore di quest'ultima sommatoria, nota come *serie aritmetica-geometrica*. Il metodo che utilizzeremo consiste nel dividere la sommatoria in diverse parti calcolabili più facilmente. Poiché $2^i = 2^{i+1} - 2^i$, abbiamo:

$$\sum_{i=1}^{k} i\, 2^i = \sum_{i=1}^{k} i\, (2^{i+1} - 2^i) = \sum_{i=1}^{k} i\, 2^{i+1} - \sum_{i=1}^{k} i\, 2^i$$

Con un cambio di variabile $j = i + 1$ e con semplici manipolazioni algebriche possiamo ora scrivere:

$$\sum_{i=1}^{k} i\, 2^i = \sum_{i=1}^{k} i\, 2^{i+1} - \sum_{j=0}^{k-1} (j+1)\, 2^{j+1} =$$

$$= \sum_{i=1}^{k} i\, 2^{i+1} - \sum_{j=0}^{k-1} j\, 2^{j+1} - \sum_{j=0}^{k-1} 2^{j+1}$$

Osserviamo ora che l'ultima sommatoria può essere calcolata usando la serie geometrica di ragione 2, mentre le prime due somme si annullano, eccezion fatta per due addendi. Abbiamo quindi:

$$\sum_{i=1}^{k} i\, 2^i = k\, 2^{k+1} - 0 - \sum_{j=1}^{k} 2^j = k\, 2^{k+1} - \left(\frac{2^{k+1} - 1}{2 - 1} - 1 \right)$$

da cui

$$\sum_{i=1}^{k} i\, 2^i = (k-1)\, 2^{k+1} + 2$$

Dividendo ambo i membri per 2 si ottiene infine l'uguaglianza (17.9). □

17.2.2 Calcolo di somme per integrazione

Una utile tecnica per calcolare il valore di alcune sommatorie consiste nell'approssimarle tramite integrali. Valgono infatti le seguenti disuguaglianze:

- se $f(x)$ è una funzione non decrescente:

$$\int_{a-1}^{b} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)dx \tag{17.10}$$

- se $f(x)$ è una funzione non crescente:

$$\int_{a}^{b+1} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)dx \tag{17.11}$$

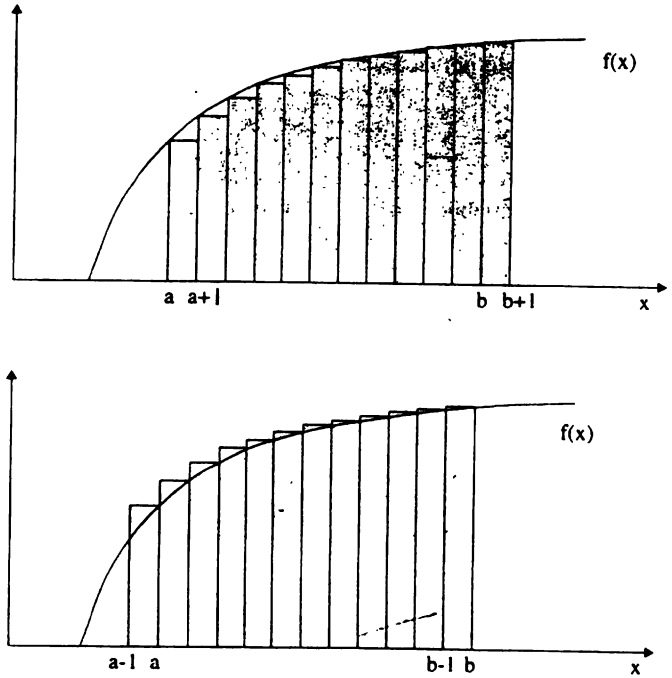


Figura 17.2 Approssimazione di sommatorie con integrali nel caso in cui la funzione $f(x)$ sia non decrescente.

La situazione nel caso in cui $f(x)$ sia non decrescente è illustrata in Figura 17.2. Vediamo ora una utile applicazione di questa tecnica nel calcolo del valore della serie armonica.

Lemma 17.3 (Serie armonica) *La serie armonica, i cui addendi sono caratterizzati dall'avere una base variabile ed un esponente costante pari a -1, soddisfa le seguenti delimitazioni:*

$$\ln(n+1) \leq \sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1 \tag{17.12}$$

Dimostrazione. Dimostriamo innanzitutto la delimitazione superiore. Poiché $f(i) = 1/i$ è una funzione decrescente, utilizzando la disuguaglianza (17.11) si ha:

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i} \leq 1 + \int_1^n \frac{1}{x} dx = 1 + \ln n - \ln 1 = \ln n + 1$$

Osserviamo che il valore dell'integrale è stato ottenuto utilizzando la Proprietà 17.5 sulla derivata dei logaritmi. Consideriamo ora la delimitazione inferiore. Con tecniche simili si ottiene:

$$\sum_{i=1}^{n} \frac{1}{i} \geq \int_1^{n+1} \frac{1}{x} dx = \ln(n+1) - \ln 1 = \ln(n+1)$$

che conclude la dimostrazione. □

17.2.3 La successione di Fibonacci

Siano $F_1 = 1$ e $F_2 = 1$. L'n-esimo numero di Fibonacci, per $n > 2$, è definito come la somma dei due precedenti numeri di Fibonacci, ovvero

$$F_n = F_{n-1} + F_{n-2}$$

Abbiamo dimostrato nel Paragrafo 1.2 del Capitolo 1 lo stretto nesso tra la successione di Fibonacci e la sezione aurea ϕ, che riassumiamo nel seguente lemma.

Lemma 17.4 *La relazione di Fibonacci ha la seguente soluzione:*

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right), \quad \text{dove} \quad \begin{cases} \phi = \frac{1+\sqrt{5}}{2} \approx +1.618 \\ \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618 \end{cases} \tag{17.13}$$

Dimostriamo ora altre due utili proprietà dei numeri di Fibonacci che abbiamo utilizzato nel corso del libro.

Lemma 17.5 *I numeri di Fibonacci soddisfano la seguente uguaglianza:*

$$F_{k+2} = 1 + \sum_{i=1}^{k} F_i$$

Dimostrazione. Per induzione su k. Per $k = 1$, la proprietà è banalmente vera, poiché

$$F_3 = F_1 + F_2 = 1 + F_1 = 1 + \sum_{i=1}^{1} F_i.$$

Assumiamo ora che

$$F_{k+2} = 1 + \sum_{i=1}^{k} F_i$$

per ogni $k \leq n$. Allora, per $k = n+1$ si ha:

$$F_{n+3} = F_{n+2} + F_{n+1} = 1 + \sum_{i=1}^{n} F_i + F_{n+1} = 1 + \sum_{i=1}^{n+1} F_i$$

come volevamo dimostrare. □

Lemma 17.6 *Risulta* $F_{k+2} \geq \phi^k$.

Dimostrazione. Per induzione su k. Per $k = 0$, $F_2 = 1 \geq \phi^0$. Per $k = 1$, $F_2 = 2 \geq \phi^1 = 1.618....$ Assumiamo ora che $F_{k+2} \geq \phi^k$ per ogni $k < n$. Allora, per $k = n$ otteniamo:

$$F_{n+2} = F_{n+1} + F_n \geq \phi^{n-1} + \phi^{n-2} = \phi^n \frac{1+\phi}{\phi^2} \geq \phi^n$$

poiché $\phi \approx 1.618$. □

17.3 Elementi di calcolo della probabilità

Nell'analisi nel caso medio e nel caso atteso (introdotte rispettivamente nei Paragrafi 2.4 e 2.6 del Capitolo 2), abbiamo usato alcuni semplici strumenti di calcolo della probabilità che richiameremo in questo paragrafo. Poiché tutti gli esempi trattati in questo libro si basano su distribuzioni di probabilità discrete su insiemi di eventi finiti, restringeremo la nostra trattazione a questo caso.

Nello studio di un fenomeno casuale dobbiamo innanzitutto definire un *universo* $\mathcal{U}$ di *eventi elementari*. Ad esempio, se il fenomeno è il lancio di un dado a sei facce, avremo sei eventi elementari corrispondenti all'uscita della i-esima faccia, con $1 \leq i \leq 6$. Come abbiamo osservato sopra, assumeremo che $\mathcal{U}$ abbia cardinalità finita. Per comporre eventi, è possibile usare le operazioni di:

- complementazione: "`non esce 1`" è il complementare dell'evento "`esce 1`";
- unione: "`esce 2 o 3`" è l'unione degli eventi "`esce 2`" e "`esce 3`";
- intersezione: "`(non esce 1) e (esce 1 o 3)`" è l'intersezione degli eventi "`non esce 1`" e "`esce 1 o 3`".

Come mostrato dall'ultimo esempio, complementazione, unione ed intersezione possono essere applicate sia ad eventi elementari che ad eventi già composti. Più formalmente, un evento è quindi un qualunque sottoinsieme dell'universo $\mathcal{U}$, ovvero un qualunque elemento dell'insieme delle parti di $\mathcal{U}$, che indicheremo con $2^{\mathcal{U}}$. L'insieme vuoto $\emptyset$ corrisponde all'*evento nullo*, mentre l'insieme $\mathcal{U}$ stesso corrisponde all'*evento certo*. Diremo inoltre che due eventi sono *mutuamente esclusivi* se la loro intersezione è nulla.

Definizione 17.1 *Sia* $\mathcal{U}$ *un universo di eventi elementari. Una* probabilità $\mathcal{P}$ *su* $\mathcal{U}$ *è una funzione* $\mathcal{P} : 2^{\mathcal{U}} \to \mathbb{R}$ *tale che:*

- $\mathcal{P}(\mathcal{U}) = 1$;
- $\mathcal{P}(\omega) \geq 0$ *per ogni evento* $\omega \in 2^{\mathcal{U}}$;
- *per ogni insieme di eventi* $\omega_1 \ldots \omega_n$ *mutuamente esclusivi a coppie:*

$$\mathcal{P}\left(\bigcup_{i=1}^{n} \omega_i\right) = \sum_{i=1}^{n} \mathcal{P}(\omega_i)$$

Dalla Definizione 17.1 segue che la probabilità di un qualunque evento è sempre un numero reale in $[0, 1]$. Inoltre, poiché l'evento nullo $\emptyset$ è il complementare dell'evento certo $\mathcal{U}$, che ha probabilità 1, risulta $\mathcal{P}(\emptyset) = 0$. Dalla definizione insiemistica di evento e dalla Definizione 17.1 è inoltre facile vedere che

$$\mathcal{P}(A \cup B) = \mathcal{P}(A) + \mathcal{P}(B) - \mathcal{P}(A \cap B)$$

Diremo che due eventi sono *indipendenti* se il verificarsi dell'uno non influenza in alcun modo il verificarsi dell'altro: formalmente, la probabilità della loro intersezione è uguale al prodotto delle loro probabilità, ovvero

$$\mathcal{P}(A \cap B) = \mathcal{P}(A) \cdot \mathcal{P}(B)$$

Esempio 17.2 Riprendiamo l'esempio relativo al lancio del dado. Supponendo che il dado non sia truccato, ogni evento elementare avrà la stessa probabilità di capitare, e quindi

$$\mathcal{P}(\texttt{esce il numero } i) = 1/6$$

per ogni i. Parleremo in tal caso di *distribuzione di probabilità uniforme*. Applicando la Definizione 17.1, è facile vedere che $\mathcal{P}(\texttt{non esce 1}) = 1 - \mathcal{P}(\texttt{esce 1}) = 5/6$. Analogamente, $\mathcal{P}(\texttt{esce 2 o 3}) = \mathcal{P}(\texttt{esce 2}) + \mathcal{P}(\texttt{esce 3}) - \mathcal{P}(\texttt{esce 2 e 3}) = 1/6 + 1/6 + 0 = 2/6$. Osserviamo che gli eventi "`esce 2`" e "`esce 3`" non sono indipendenti: il prodotto delle loro probabilità è infatti $1/36$, che è diverso dalla probabilità della loro intersezione, pari a 0. Ciò è intuitivo, se si pensa che il verificarsi di uno di questi due eventi implica che l'altro non potrà accadere. □

Definizione 17.2 *Una* variabile aleatoria *(discreta) è una funzione dall'universo* $\mathcal{U}$ *degli eventi elementari all'insieme dei numeri reali:*

$$X : \mathcal{U} \to \mathbb{R}$$

X associa un numero reale a ciascun possibile risultato di un esperimento. L'evento $X = x$ è l'unione degli eventi elementari ω per cui $X(\omega) = x$, e la sua probabilità è la somma delle probabilità degli eventi elementari:

$$\mathcal{P}\{X = x\} = \sum_{\omega \in \mathcal{U}:\ X(\omega) = x} \mathcal{P}\{\omega\}$$

Una utile sintesi della distribuzione dei valori assunti da una variabile aleatoria è fornita dal suo *valore atteso*, anche noto in letteratura come *media*, *valore medio*, o *speranza matematica*.

Definizione 17.3 *Il* valore atteso *di una variabile aleatoria discreta* X*, che indicheremo con* $E[X]$*, è la somma dei valori che la variabile può assumere, ciascuno pesato con la probabilità che venga assunto:*

$$E[X] = \sum_{x} x\, \mathcal{P}\{X = x\}$$

Esempio 17.3 Riprendiamo l'esempio del lancio del dado, e definiamo una variabile aleatoria X come segue:

$$X(\omega) = \begin{cases} 9 & \text{se esce 1 o 2} \\ 20 & \text{se esce 3, 4 o 5} \\ 30 & \text{se esce 6} \end{cases}$$

È facile verificare che $\mathcal{P}\{X = 9\} = 1/3$, $\mathcal{P}\{X = 20\} = 1/2$ e $\mathcal{P}\{X = 30\} = 1/6$. Il valore atteso di X è quindi dato da:

$$E[X] = 9\frac{1}{3} + 20\frac{1}{2} + 30\frac{1}{6} = 18$$

Ciò significa che il valore medio della variabile X è 18: notiamo che questo valore non è realmente assunto in corrispondenza di nessun evento elementare. □

Concludiamo osservando che i concetti di variabile aleatoria discreta e di valore atteso sono molto importanti nell'analisi nel caso medio e nell'analisi di algoritmi randomizzati. Pensiamo, ad esempio, all'algoritmo `quickSort` descritto nel Paragrafo 4.5 del Capitolo 4: il numero di confronti eseguito dall'algoritmo è una variabile aleatoria, di cui abbiamo calcolato il valore atteso.

17.4 Elementi di calcolo combinatorio

Il calcolo combinatorio si occupa di calcolare la cardinalità di insiemi finiti. Tipiche domande hanno la seguente forma: "in quanti modi possono essere ordinati n elementi?", oppure "quanti sottoinsiemi di dimensione k ammette un insieme contenente n elementi?". Le formule fornite dal calcolo combinatorio sono tipicamente utili quando si vuole analizzare un algoritmo nel caso medio o atteso, assumendo una distribuzione uniforme delle istanze di ingresso o dei numeri casuali. Le formule che descriveremo hanno però anche un'utilità più generale, indipendente dalle analisi probabilistiche.

Dati due insiemi A e B di cardinalità n_1 e n_2, denotiamo con $A \times B$ il loro *prodotto cartesiano*, cha ha cardinalità $n_1 \cdot n_2$. Generalizzando questa osservazione possiamo facilmente calcolare il numero di possibili *stringhe* di lunghezza n su un alfaheto Σ di k simboli:

Lemma 17.7 *Il numero di stringhe di lunghezza n su un alfabeto di cardinalità k è pari a n^k.*

Dimostrazione. Segue dal fatto che ogni simbolo può essere scelto in k possibili modi. □

Ad esempio, avendo a disposizione $\log_2 n$ bit, potremo rappresentare $2^{\log_2 n} = n$ numeri binari distinti.

Una *permutazione* di un insieme di n elementi è definita come una sequenza ordinata in cui ciascun elemento appare esattamente una volta. Più formalmente, una permutazione di un insieme A è una applicazione iniettiva di A in se stesso (ricordiamo che una funzione è iniettiva se ad elementi distinti del dominio corrispondono elementi distinti del codominio). Poiché uno stesso elemento appare una ed sola volta in una permutazione, possiamo facilmente calcolare il numero di permutazioni.

Lemma 17.8 *Il numero di permutazioni distinte di un insieme di n elementi è pari a $n!$.*

Dimostrazione. Il primo elemento può essere scelto in n modi distinti; il secondo elemento potrà essere scelto in $n - 1$ modi, poiché non potremo ripetere il primo; analogamente, il terzo elemento potrà essere scelto in $n - 2$ modi, e via dicendo. La scelta dell'ultimo elemento sarà obbligata. Ricordiamo che $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = n!$, da cui l'enunciato. □

Una utile approssimazione del numero $n!$ di permutazioni è data dalla *formula di De Moivre-Stirling*:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n-1}\right) \tag{17.14}$$

Questa disuguaglianza ci fornisce un valore approssimato per $n!$ con un errore che tende a 0 per n tendente all'infinito. Possiamo quindi scrivere:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{17.15}$$

Ricordiamo che abbiamo usato la formula di De Moivre-Stirling nella dimostrazione della delimitazione inferiore per il problema dell'ordinamento nel Capitolo 4.

Consideriamo infine la domanda "quanti sottoinsiemi di dimensione k ammette un insieme contenente n elementi?". Ad esempio, qual è il massimo numero di archi tra vertici distinti in un grafo non orientato? (In questo caso $k = 2$.) Tali sottoinsiemi sono noti come *combinazioni* di k elementi.

Lemma 17.9 *Il numero di combinazioni di k elementi su un insieme di dimensione n è pari al coefficiente binomiale*

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Dimostrazione. Calcoliamo dapprima il numero di modi in cui possiamo scegliere *ordinatamente* k elementi tra n. Il primo elemento può essere scelto in n modi distinti; il secondo elemento potrà essere scelto in $n - 1$ modi, poiché non possiamo ripetere il primo; analogamente, il terzo elemento potrà essere scelto in

$n-2$ modi, e via dicendo, fino a scegliere il k-esimo elemento, per cui avremo $n-k+1$ possibilità. Il numero totale di sottoinsiemi di k elementi ordinati è pertanto:

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$$

Se non siamo interessati all'ordinamento tra elementi nello stesso sottoinsieme, dobbiamo considerare che in questo conteggio lo stesso insieme appare ripetutamente, con gli elementi permutati in modo diverso. Poiché il numero di permutazioni di k elementi è $k!$, basterà quindi dividere il numero ottenuto precedentemente per $k!$, ottenendo esattamente il coefficiente binomiale. □

I coefficienti binomiali godono di innumerevoli utili proprietà, alcune delle quali sono state utilizzate nel corso del libro. In particolare, è facile convincersi dalla definizione di coefficiente binomiale che:

$$\binom{n}{k} = \binom{n}{n-k} \tag{17.16}$$

Inoltre, i coefficienti binomiali intervengono nell'espansione binomiale di polinomi, come mostrato dalla seguente utile relazione:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} \tag{17.17}$$

17.5 Elementi di teoria dei grafi

Nei Capitoli 11 - 14 abbiamo descritto numerosi algoritmi per risolvere problemi che possono essere formulati in termini di grafi. Un grafo è infatti una struttura combinatorica molto ricca e complessa, e non è quindi sorprendente che ben si presti come modello per numerosi problemi reali di varia natura. Rimandiamo al Capitolo 11 per la definizione di grafo, ed in particolare al Paragrafo 11.1 per le definizioni preliminari che riguardano le nozioni di adiacenza, grado, cammini, cicli, e connettività. In questo paragrafo, approfondiremo invece le proprietà di due specifiche classi di grafi: gli alberi e i grafi planari.

17.5.1 Alberi

Gli alberi ci hanno accompagnato dal primo all'ultimo capitolo di questo libro. Pur avendo trattato soprattutto con alberi radicati (vedi Capitolo 3), nel seguito ci soffermeremo sulle proprietà di *alberi liberi*, in cui non esiste un vertice designato come radice specifica e gli archi sono non orientati. Ad esempio, il minimo albero ricoprente di un grafo non orientato è un albero libero, come segue dalla Definizione 12.1. Formalmente, definiamo un albero libero come segue:

Definizione 17.4 *Un albero libero è un grafo connesso aciclico non orientato.*

Il seguente lemma riassume tre caratterizzazioni equivalenti degli alberi liberi.

Lemma 17.10 *Sia $T = (V, E)$ un grafo non orientato. Le seguenti affermazioni sono equivalenti:*

1. *T è un albero;*
2. *T è un grafo connesso minimale, ovvero la rimozione di un qualunque arco lo disconnetterebbe;*
3. *T è un grafo aciclico massimale, ovvero l'aggiunta di un qualunque arco introdurrebbe un ciclo.*

Dimostrazione. Dimostreremo, usando la tecnica della contraddizione (o assurdo), che ciascuna affermazione implica la successiva, ciclicamente.

(1)⇒(2) : occorre solo dimostrare la proprietà di minimalità, poiché la connessione segue dalla definizione stessa di albero. Supponiamo per assurdo che esista un arco $e = (u, v)$ la cui rimozione non disconnette T: deve quindi esistere un cammino alternativo tra u e v, non contenente e. Ciò è impossibile, poiché l'unione di tale cammino con e introdurrebbe un ciclo, contraddicendo l'ipotesi che T sia un albero.

(2)⇒(3) : se per assurdo esistesse un ciclo, la rimozione di un qualuque arco del ciclo non disconnetterebbe T, contraddicendo l'ipotesi di minimalità. Dimostriamo ora la proprietà di massimalità: se esistesse un arco $e = (u, v)$ la cui aggiunta non introduce un ciclo, u e v dovrebbero essere disconnessi in T, violando quindi l'ipotesi di connessione.

(3)⇒(1) : occorre solo dimostrare la proprietà di connessione. Se esistessero due nodi u e v disconnessi in T, l'aggiunta dell'arco (u, v) non introdurrebbe cicli, violando l'ipotesi di massimalità.

Le tre implicazioni appena dimostrate implicano la completa equivalenza delle caratterizzazioni nell'enunciato. □

Una proprietà degli alberi che abbiamo utilizzato spesso è il fatto un albero con n vertici ha esattamente $(n-1)$ archi. Tale proprietà, ad esempio, è fondamentale per comprendere perché la rappresentazione tramite vettore padri discussa nel Paragrafo 3.3 del Capitolo 3 funzioni correttamente.

Lemma 17.11 *Un albero con n vertici ha esattamente $(n-1)$ archi.*

Dimostrazione. Sia m il numero di archi di T. La dimostrazione procede per induzione sul namero di vertici n. Il passo base ($n = 1$) è banale, non esistendo alcun arco. Supponiamo ora che ogni albero con $k < n$ vertici abbia esattamente $k-1$ archi, e consideriamo un albero T con n vertici. Sia v un nodo di T, e sia d_v il namero dei suoi vicini; poiché T è connesso, deve essere $d_v > 0$. La rimozione di v disconnette T in d_v alberi, $T_1 \ldots T_{d_v}$, contenenti rispettivamente

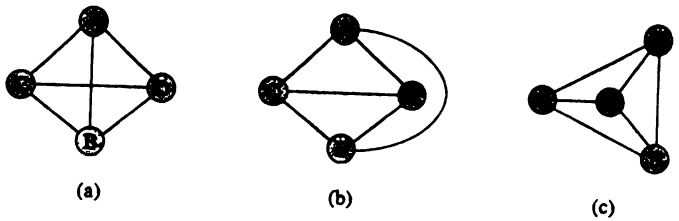


Figura 17.3 Un grafo planare e due sue diverse rappresentazioni piane.

$n_1 \ldots n_{d_v}$ vertici. Poiché $n_i < n$ per ogni i, per ipotesi induttiva il numero totale di archi $\tilde{m}$ in questi alberi soddisfa

$$\tilde{m} = (n_1 - 1) + (n_2 - 1) + \ldots + (n_{d_v} - 1) = \left(\sum_{i=1}^{d_v} n_i\right) - d_v = n - 1 - d_v$$

Riaggiungendo il nodo v e i d_v archi ad esso incidenti abbiamo quindi che

$$m = \tilde{m} + d_v = (n - 1 - d_v) + d_v = n - 1$$

che completa la dimostrazione. □

17.5.2 Grafi planari

Un grafo è *planare* se può essere disegnato nel piano senza intersezioni di archi: un disegno che soddisfi questa proprietà è detto *rappresentazione piana* del grafo. Ad esempio, il grafo mostrato in Figura 17.3 è planare: sebbene la rappresentazione in Figura 17.3(a) non sia una rappresentazione piana a causa dell'incrocio tra gli archi (a, b) e (c, d), il grafo può comunque essere rappresentato planarmente come mostrato dai due diversi disegni in Figura 17.3(b) e 17.3(c).

I grafi planari godono di innumerevoli proprietà combinatorie, e si dà spesso il caso che problemi difficili in generale (ad esempio, problemi NP-ardui introdotti nel Capitolo 16) ammettano algoritmi più efficienti (ad esempio, algoritmi polinomiali) su grafi planari. Una discussione delle eleganti caratterizzazioni e delle interessanti proprietà di questa classe di grafi è al di fuori dallo scopo di questo libro, e rimandiamo al classico testo [10] per una panoramica dettagliata.

Ci limitiamo in questa sede a discutere una utile formula, scoperta intorno al 1750 da Eulero [4] e dimostrata da Legendre [9] nel 1794, che lega il numero di vertici, di archi e di facce in un grafo planare. Data una rappresentazione piana, una *faccia* è una regione del piano racchiusa da archi. Ogni rappresentazione piana ha una faccia speciale, detta faccia esterna, di dimensione infinita. Ad esempio, nella rappresentazione piana in Figura 17.3(c), il triangolo delimitato dagli archi (a, c), (c, d) e (d, a) è una faccia; la faccia esterna in questo esempio è delimitata da (a, c), (c, b) e (b, a). La formula di Eulero può essere espressa come segue.

Lemma 17.12 *Sia G un grafo planare connesso con n vertici e m archi, e sia f il numero di facce in una qualunque rappresentazione piana di G. Risulta*

$$n - m + f = 2$$

Dimostrazione. Dimostreremo la formula per induzione sul numero di archi. Se G non ha archi, essendo connesso conterrà necessariamente un solo vertice: quindi $n = 1$, $f = 1$ (la faccia esterna), e $m = 0$, da cui è facile convincersi che l'uguaglianza è verificata. Altrimenti, sia $e = (u, v)$ un qualunque arco di G. Se e connette due vertici distinti (ovvero $u \neq v$), contraendolo e trasformando i vertici u e v in un unico vertice, ridurremo di 1 sia n che m, mantenendo il numero di facce f inalterato: la formula varrà quindi per ipotesi induttiva. Se e è un cappio (ovvero $u = v$), deve necessariamente separare due facce e rimuovendolo ridurremo di 1 sia f che m, mantenendo il numero di vertici n inalterato: la formula varrà anche in questo caso per ipotesi induttiva. □

Diremo che un grafo è semplice se non contiene né cappi né archi multipli, dove un cappio è un arco del tipo (u, u). Usando la formula di Eulero si può facilmente vedere che i grafi semplici planari contengono un numero di archi e di facce lineare nel numero di vertici.

Corollario 17.1 *In un grafo semplice connesso planare con $n \geq 3$ vertici il numero m di archi ed il numero f di facce soddisfano $m \leq 3n - 6$ e $f \leq 2n - 4$, rispettivamente.*

Dimostrazione. In un grafo semplice connesso planare ogni faccia, tranne possibilmente la faccia esterna, è delimitata da almeno tre archi ed ogni arco tocca al più due facce. Quindi $3f \leq 2m$. Dalla formula di Eulero abbiamo:

$$2 = n - m + f \leq n - m + \frac{2m}{3} = n - \frac{m}{3}$$

da cui $m \leq 3n - 6$. Analogamente:

$$2 = n - m + f \leq n - \frac{3f}{2} + f = n - \frac{f}{2}$$

da cui $f \leq 2n - 4$. □

Un grafo semplice è detto *planare massimale* se è planare e l'aggiunta anche di un solo arco distruggerebbe la proprietà di planarità. Tutte le facce di un grafo planare massimale sono delimitate da tre archi, e per questo motivo un tale grafo è anche detto *triangolazione*. Dal Corollario 17.1, le triangolazioni hanno esattamente $3n - 6$ archi e $2n - 4$ facce.

Il limitato numero di archi in un grafo planare implica l'esistenza di numerosi vertici di grado basso, come dimostrato nel seguente corollario.

Corollario 17.2 *In un grafo planare con n vertici, almeno $n/2$ vertici hanno grado minore di 12.*

Dimostrazione. Supponiamo per assurdo che esistano almeno $n/2$ vertici con grado maggiore o uguale a 12. In tal caso il grafo avrebbe almeno

$$m \geq \frac{1}{2}\frac{12n}{2} = 3n$$

archi: potremmo infatti contare ogni arco due volte, una per ogni estremo. Ciò contraddirrebbe il Corollario 17.1, secondo cui deve essere $m \leq 3n - 6$. □

17.6 Note bibliografiche

Knuth [8] e Graham, Knuth, Patashnik [5] descrivono molte tecniche di matematica discreta utili nell'analisi degli algoritmi, tra cui i principali risultati di base di calcolo combinatorio e vari metodi per il calcolo di sommatorie. Classici testi di matematica non strettamente specifici per le discipline informatiche, quali [1] e [15], presentano in dettaglio le proprietà di funzioni, serie e successioni. Una famosa corrispondenza tra Pascal e Fermat, risalente al 1654, sembra rappresenti l'inizio degli studi di problemi di probabilità [11]. Un classico libro di testo sulla teoria della probabilità è dovuto a Grimmet e Stirzaker [6]. Steele [14] discute le tecniche del calcolo della probabilità più direttamente applicabili a problemi di ottimizzazione combinatoria, e Spencer [13] presenta molte tecniche probabilistiche sofisticate. Esistono infine innumerevoli testi di teoria dei grafi, tra cui ricordiamo [2, 3, 7].

Riferimenti bibliografici

[1] T. M. Apostol, *Calculus*, volume 1, seconda edizione, Blaisdell Publishing Company, 1967.

[2] C. Berge, *Graphs*, Elsevier Science Publishers, 1991.

[3] B. Bollobas, *Graph theory: an introductory course*, Springer Verlag, 1979.

[4] L. Euler, *Elementa doctrinae solidorum. Demonstratio nonnullarum insignium proprietatum, quibus solida hedris planis inclusa sunt praedita*, Novi comment acad. sc. imp. Petropol., 4, 1752-3, 109-140-160.

[5] R. L. Graham, D. E. Knuth e O. Patashnik, *Concrete Mathematics*, seconda edizione, Addison-Wesley, 1994.

[6] G. R. Grimmet, D. Stirzaker, *Probability and Random Processes*, Oxford University Press, 1992.

[7] F. Harary, *Graph theory*, Addison Wesley, 1969.

[8] D. E. Knuth, *The Art of Computer Programming*, volume 1 *Fundamental algorithms*, volume 2 *Seminumerical algorithms*, volume 3 *Sorting and searching*, Addison-Wesley, Reading, 1973.

[9] A. M. Legendre, *Élements de géométrie*, Paris, 1794.

[10] T. Nishizeki e N. Chiba, *Planar Graphs; Theory and Algorithms*, North-Holland, 1988.

[11] A. Rényi, *Pascal: lettere sulla probabilità*, Stampa Alternativa Millelire, edizione italiana a cura di Enzo Lombardo, Roma, 1991.

[12] W. W. Rouse Ball, *Mathematical recreations and problems of past and present times*, Macmillan, London, 1892.

[13] J. Spencer, *Ten Lectures on the Probabilistic Method*, seconda edizione, SIAM, 1994.

[14] J. M. Steele, *Probability theory and combinatorial optimization*, SIAM, 1997.

[15] B. B. Thomas e R. L. Finney, *Calculus and analytic geometry*, settima edizione, Addison Wesley, 1988.

Indice analitico

Ackermann
  funzione di, 233, 235, 317
  funzione inversa α, 233, 235, 317
Adel'son-Vel'skiĭ, G. M., 139
adiacenza di vertici
  in un grafo, 265
aggiornamento
  operazione di, 47
agglomerazione, 173
  primaria, 180
  secondaria, 180
Ahuja, R. K., 341, 363
albero, 69–77, 268, 274, 278, 282, 289
  2-3, 152
  2-3-4, 162
  altezza di un, 70
  antenato in un, 70
  arco di un, 70
  autoaggiustante, 146
  AVL, 139
  B*-albero, 166
  B-albero, 156
  BFS di un grafo, 277, 278, 281, 282, 285, 297
  bilanciato
    in altezza, 140
    nel peso, 167
  binario di ricerca, 134
  completo, 71
  degli intervalli, 387
  dei cammini minimi, 324
  del torneo, 118
  della ricorsione, 7, 40
  DFS di un grafo, 282, 283, 285, 290–294, 297
  di decisione, 83
  di Fibonacci, 140
  discendente in un, 70
  figlio di un nodo, 70
  fratello di un nodo, 70
  genitore di un nodo, 70
  grado di un nodo, 70
  livello di un nodo, 70
  minimo antenato comune, 234, 291
  minimo ricoprente, 301–305, 307, 308, 311, 313, 315–317
  nodo di un, 70
  operazione
    `aggiungiNodo`, 72
    `aggiungiSottoalbero`, 72
    `figli`, 72
    `grado`, 72
    `numNodi`, 72
    `padre`, 72
    `rimuoviSottoalbero`, 72
  padre di un nodo, 70
  priority search tree, 382
  profondità di un nodo, 70
  QuickFind, 213
  QuickFind bilanciato, 218
  QuickUnion, 215
  QuickUnion bilanciato, 221
  range tree, 380
  rappresentazione
    collegata, 73
    con lista di figli, 74
    con primo figlio-fratello successivo, 74
    con puntatori ai figli, 74
    con vettore padri, 71
    con vettore posizionale, 72
    indicizzata, 71

red-black, 162
ricoprente di un grafo, 288, 301
splay, t46
visita di un, 74–77
in ampiezza, 77
in postordine, 76
in preordine, 76, 282
in profondità iterativa, 75
in profondità ricorsiva, 76
simmetrica, 76
algoritmo
deterministico, 45
di Beltman e Ford, 326–327
di Borůvka, 313
di Dijkstra, 330–336
di Edmonds e Karp, 356
di Floyd e Warshall, 336–338
di Ford e Fulkerson, 355
di Graham, 370
di Jarvis, 369
di Kruskal, 305, 307
di Prim, 308, 311
di visita di un albero, 74–77
di visita in ampiezza
di un albero, 77
di un grafo, 278
di visita in postordine di un albero, 76
di visita in preordine di un albero, 76, 282
di visita in profondità
di un albero, 75–77
di un grafo, 282
di visita simmetrica di un albero, 76
non deterministico, 402
ottimo, 29
randomizzato, 45
algoritmo
`BellmanFord`, 326
`bubbleSort`, 88
`bucketSort`, 105
`cammino`, 323
`connessoGrafo`, 289
`Dijkstra`, 334
`DijkstraGenerico`, 332
`distanzaStringhe`, 248
`distanzeAciclico`, 330
`distanzeGenerico`, 325
`distribuisciResto`, 255
`EdmondsKarp`, 356
`fibonacci1`, 5
`fibonacci2`, 6
`fibonacci3`, 9, 242
`fibonacci4`, 11
`fibonacci5`, 14
`fibonacci6`, 15
`fixheap`, 92
`FloydWarshall`, 337
`flussoMassimo`, 351
`flussoMassimoRic`, 352
`FordFulkerson`, 355
`formulaQuantificata`, 401
`fortementeConnesse`, 296
`heapify`, 93
`heapSelect`, 119
`heapSort`, 90
`insertionSort`, 86
`integerSort`, 103
`inviluppoInduttivo`, 367
`Kruskal`, 307
`KruskalGenerico`, 305
`localizzazioneDiPunti`, 377
`marciaDiJarvis`, 370
`merge`, 96
`mergeSort`, 95
`minimo`, 116
`ordinamentoTopologico`, 329
`ordineMatrici`, 252
`paradigmaGreedy`, 254
`partition`, 98
`preordine`, 282
`Prim`, 311
`PrimGenerico`, 308
`profondita`, 78
`queryPriority`, 383
`quickSelect`, 121
`quickSort`, 98
`radixSort`, 106
`ricercaBinariaIter`, 34
`ricercaBinariaRic`, 36
`ricercaRandomizzata`, 46

`ricercaSequenziale`, 30
`sampleSelect`, t23
`scansioneDiGraham`, 370
`secondoMinimo`, t17
`select`, t25
`select1`, t19
`select2`, t20
`selectionSort`, 86
`visitaBFS`, 77, 278
`visitaDFS`, 75
`visitaDFSRicorsiva`, 76, 282
`visitaFortementeConnesse`, 294
`visitaGenerica`, 74, 274
algoritmo randomizzato
`ricercaRandomizzata`, 46
`quickSelect`, t21
`quickSort`, 98
Alon, N., 34t
altezza
dell'albero della ricorsione, 40
di un albero, 70
di un albero 2-3, 152
di un albero AVL, t40
di un albero *red-black*, 163
di un B-albero, 157
Amdahl, G. M., 185
analisi
ammortizzata, 47–53, 147
metodo dei crediti, 50
metodo del potenziale, 51, 147
di algoritmi, 2
nel caso medio, 31
nel caso migliore, 31
nel caso peggiore, 31
probabilistica, 45–47
di `quickSelect`, 121
di `quickSort`, 99
di `ricercaRandomizzata`, 46
antenato
in un albero, 70
in un grafo, 266
arco di un grafo
incidente su un vertice, 265
arco di un grafo orientato
entrante in un vertice, 265
uscente da un vertice, 265
aumentante
cammino in una rete, 353
passo, 350
average case, 31

B*-albero, t66
B-albero, 156
Bayer, R., 157
Beltman
condizione di, 322
Beltman e Ford
algoritmo di, 326–327
Beltman, R., 326–327, 34t
best case, 31
bilanciamento
fattore di, t40
in altezza, t40
net peso, t67
spazio-tempo, t76
binomiale
heap, t93
heap rilassato, 197
Blum, M., t24
Borůvka
algoritmo di, 313
Borůvka, O., 317
Brown, M. R., 209

cambiamento di base dei logaritmi, 420
cammini minimi, 347
albero dei, 324
algoritmo di
Beltman e Ford, 326
Dijkstra, 332, 334
Floyd e Warshall, 337
tecnica del rilassamento, 325
cammino
aumentante in una rete, 353
in un grafo, 266
minimo k-vincolato in un grafo, 336
minimo in un grafo, 320, 347
semplice in un grafo, 266
campionamento, 123

capacità
di un arco, 347
residua, 349
caso peggiore, migliore e medio
analisi nel, 31
Chazelle, B., 317
Cherkassky, B. V., 341
chiave, 61–68, 133–167, 169–185
ciclo
in un grafo, 266
in un grafo orientato, 266
negativo, 321
semplice in un grafo, 266
classe
Albero23, 156
AlberoAVL, 145
AlberoBinarioDiRicerca, 134
AlberoSplay, 147
ArrayDoubling, 65
ArrayOrdinato, 63
BAlbero, 161
DHeap, 189
HeapBinomiale, 194
HeapBinomialeRilassato, 198
HeapFibonacci, 203
QuickFind, 213
QuickFindBilanciato, 218
QuickUnion, 215
QuickUnionBilanciato, 221
QuickUnionBilanciatoSize, 224
StrutturaCollegata, 67
TavolaAccessoDiretto, 170
TavolaHashAperta, 177
TavolaHashApertaBis, 182
TavolaHashListeColl, 175
TavolaHashPerfetta, 171
classe di complessità, 399
EXPTIME, 399
NP, 402
PSPACE, 399
P, 399
coda, 68
operazione
dequeue, 69
enqueue, 69
first, 69
isEmpty, 69
coda con priorità, 187–209
classe
DHeap, 189
HeapBinomiale, 194
HeapBinomialeRilassato, 198
HeapFibonacci, 203
implementazione con
d-heap, 189
heap binario, 90
heap binomiale, 194
heap binomiale rilassato, 198
heap di Fibonacci, 203
operazione
decreaseKey, 188, 189, 194, 198, 203
delete, 188, 189, 194, 198, 203
deleteMin, 188, 189, 194, 198, 203
findMin, 188, 189, 194, 198, 203
increaseKey, 188, 189, 194, 198, 203
insert, 188, 189, 194, 198, 203
merge, 188, 189, 194, 198, 203
componente fortemente connessa
di un grafo orientato, 290
testa di una, 292
Cooper, G. M., 80
Coppersmith, D., 341
Cormen, T. H., 80
costo
ammortizzato, 48
logaritmico, 25
uniforme, 24

Dantzig, G. B., 341, 362
decomposizione
del flusso, 356
delimitazione
inferiore (o *lower bound*), 28

superiore (o *upper bound*), 27
Dial, R. B., 341
Dijkstra
algoritmo di, 330–336
Dijkstra, E. W., 317, 330–336, 341
Dinitz, E. A., 362
discendente
in un albero, 70
in un grafo, 266
disciplina di accesso
FIFO, 68
LIFO, 68
distanza
k-vincolata in un grafo, 336
disuguaglianza triangolare in un grafo, 322
in un grafo, 322
tra stringhe, 244–249
divide et impera, 37, 40, 93, 95, 97, 111, 128, 237–242, 259, 260
algoritmo heapify, 93
algoritmo mergeSort, 95
algoritmo quickSort, 97
analisi di algoritmi basati su, 37
relazione di ricorrenza del, 40, 128
ricerca binaria, 34
dizionario, 61–68, 133–167, 169–185
classe
Albero23, 156
AlberoAVL, 145
AlberoBinarioDiRicerca, 134
AlberoSplay, 147
ArrayDoubling, 65
ArrayOrdinato, 63
BAlbero, 161
StrutturaCollegata, 67
TavolaAccessoDiretto, 170
TavolaHashAperta, 177
TavolaHashApertaBis, 182
TavolaHashListeColl, 175
TavolaHashPerfetta, 171
implementazione con
albero 2-3, 152
albero 2-3-4, 162
albero autoaggiustante, 146
albero AVL, 139
albero binario di ricerca, 134
albero splay, 146
albero *red-black*, 162
array non ordinato, 65
array ordinato, 63
B-albero, 156
struttura circolare doppiamente collegata, 67
tavola ad accesso diretto, 170
tavola hash con indirizzamento aperto, 177, 182
tavola hash con liste di collisione, 175
tavola hash perfetta, 171
operazione
delete, 62, 63, 65, 67, 137, 144, 155, 161, 170, 171, 175, 177, 182
insert, 62, 63, 65, 67, 136, 143, 154, 160, 170, 171, 175, 177, 182
max, 136
pred, 137
search, 62, 63, 65, 67, 135, 153, 159, 170, 171, 175, 177, 182
Driscoll, J. R., 209

Edelsbrunner, H., 387
Edmonds e Karp
algoritmo di, 356
Edmonds, J., 356–358, 362
Eulero
formula di, 373, 432
Even, S., 362

Fibonacci, 2, 242, 243, 247, 249
albero di, 140
heap di, 201
numeri di, 2, 20, 242, 243, 247, 249
relazione di, 4, 6
successione di, 425
FIFO

disciplina di accesso, 68
Floyd e Warshall
algoritmo di, 336–338
Floyd, R. W., 123, 124, 336–338, 341
flusso, 348–363
algoritmo di
Edmonds e Karp, 356
Ford e Fulkerson, 355
decomposizione del, 356
massimo, 349–363
nullo, 349
quantità di, 348–363
rete di, 347–363
su un cammino, 353
Ford e Fulkerson
algoritmo di, 355
Ford, L. R., 326–327, 341, 355, 361, 362
foresta, 193, 197, 201
minima ricoprente, 302
forma normale congiuntiva, 400
formula Booleana quantificata, 400
formula di De Moivre-Stirling, 429
Fredman, M. L., 209, 235, 317, 341
Fulkerson, D. R., 341, 355, 361, 362
funzione
hash, 171
hash perfetta, 171

Gabow, H. N., 209, 341
Galil, Z., 341, 362
Gallo, G., 341
gerarchia di complessità, 404
Goldberg, A. V., 341, 362
Gonnet, G. H., 80, 185
grafo, 70, 263, 267
adiacenza di vertici, 265
albero ricoprente di un, 288, 301
antenato in un, 266
cammino in un, 266
cammino minimo k-vincolato in un, 336
cammino minimo in un, 320
cammino semplice in un, 266
ciclo in un, 266
ciclo semplice in un, 266
componente connessa di un, 288
componente fortemente connessa di un, 290
connesso, 266
costo di un arco, 319
diretto, 264
diretto aciclico, 266
discendente in un, 266
distanza k-vincolata tra vertici, 336
distanza tra vertici, 322
disuguaglianza triangolare, 322
grado di un vertice, 271
lunghezza di un cammino, 266
minima foresta ricoprente di un, 302
minimo albero ricoprente di un, 301
non orientato (o non diretto), 263
componente connessa di un, 288
ordinamento topologico di un grafo aciclico, 327
orientato (o diretto), 264
ciclo in un, 266
componente fortemente connessa di un, 290
fortemente connesso, 266
planare, 372, 432
formula di Eulero, 432
rappresentazione, 268
con lista di archi, 268
con liste di adiacenza, 270
con liste di incidenza, 271
con matrice di adiacenza, 271
con matrice di incidenza, 273
sottografo di un, 266, 301
sottografo indotto di un, 267
taglio in un, 303, 358
vertici connessi in un, 319
visita generica di un, 274
visita in ampiezza di un, 277, 278
visita in profondità di un, 282
Graham
algoritmo di, 370
Graham, R. L., 317, 370

hash, 169–185
agglomerazione, 173
chiavi non intere, 173
chiavi non numeriche, 173
funzione hash, 171
funzione hash perfetta, 171
hashing doppio, 180
indirizzamento aperto, 177
lista di collisione, 175
scansione lineare, 179
scansione quadratica, 180
tavola, 171
uniformità semplice, 173
hashing, 170
heap, 90, 188
binario, 90
binomiale, 193
binomiale rilassato, 197
di Fibonacci, 201
ordinamento mediante, 90
Hell, P., 317
Hicks, E., 79
Hopcroft, J. E., 235

indirizzamento
aperto in una tavola hash, 177
interrogazione
operazione di, 47
inviluppo convesso, 366–372
algoritmo di Graham, 370
algoritmo di Jarvis, 369
algoritmo induttivo, 366
iterazione
metodo della, 37

Jarník, V., 317
Jarvis, R. A., 369
Johnson, D. B., 341

Kaas, R., 341
Karger, D. R., 341, 363
Karp, R., 356–358, 362
Knuth, D. E., 58, 79, 95, 185
Koller, D., 341
Kruskal
algoritmo di, 305, 307
Kruskal, J. N., 317

Landis, Y. M., 139
Leiserson, C. E., 80
Levine, M. S., 363
LIFO
disciplina di accesso, 68
lista
di archi in un grafo, 268
di collisione, 175
liste di adiacenza
in un grafo, 270
localizzazione di punti, 372–377
lower bound, 28
Luhn, H. P., 185
lunghezza
di un cammino in un grafo, 266

macchina
a puntatori, 24
a registri, 23
di Turing, 23
Magnanti, T. L., 341, 363
Margalit, O., 341
matrice
di adiacenza in un grafo, 271
di incidenza in un grafo, 273
matrici
moltiplicazione tra, 241, 242
McCreight, E., 157, 382
mediano, 115
calcolo deterministico del, 124
calcolo randomizzato del, 119
Mehlhorn, K., 341
metodo
dell'iterazione, 37
della sostituzione, 38
minima foresta ricoprente, 302
minimo albero ricoprente, 301–317
algoritmo di
Borůvka, 313
Kruskal, 305, 307
Prim, 308, 311
regola del
ciclo, 303, 316
taglio, 303, 316

minimo antenato comune, 234, 291
off-line, 234
misura di costo
logaritmico, 25
uniforme, 24
modello di calcolo, 23–24, 53–55
con memoria esterna, 53
macchina a puntatori, 24
macchina a registri, 23
macchina di Turing, 23
moltiplicazione
tra interi di grandezza arbitraria, 239
tra matrici, 241
Moore, E. F., 341

Naamad, A., 362
Newell, A., 79
notazione asintotica, 11, 12, 25–27
O, Ω, Θ, 26

operazione
di aggiornamento, 47
di interrogazione, 47, 376, 377
operazione
aggiungiNodo, 72
aggiungiSottoalbero, 72
decreaseKey, 188, 189, 194, 198, 203
delete, 62, 63, 65, 67, 137, 144, 155, 161, 170, 171, 175, 177, 182, 188, 189, 194, 198, 203
deleteMin, 188, 189, 194, 198, 203
dequeue, 69
enqueue, 69
figli, 72
find, 211–213, 215, 218, 221, 224, 307
findMin, 188, 189, 194, 198, 203
first, 69
fuse, 154, 161
grado, 72
increaseKey, 188, 189, 194, 198, 203
insert, 62, 63, 65, 67, 136, 143, 154, 160, 170, 171, 175, 177, 182, 188, 189, 194, 198, 203
isEmpty, 68, 69
makeSet, 212, 213, 215, 221
max, 136
merge, 188, 189, 194, 198, 203
muoviAlto, 191
muoviBasso, 191
numNodi, 72
padre, 72
pop, 68
pred, 137
push, 68
rimuoviSottoalbero, 72
ristruttura, 195
search, 62, 63, 65, 67, 135, 153, 159, 170, 171, 175, 177, 182
split, 154, 160
staccaInCascata, 204
top, 68
union, 211–213, 215, 218, 221, 224, 307
ordinamento
in loco, 85
a bolle, 88
basato su confronti, 81
in tempo lineare, 102
mediante heap, 90
per fusione, 95
per inserimento, 86
per selezione, 86
stabilità di un algoritmo di, 105
topologico di un grafo aciclico, 327
Orlin, J. B., 341, 363

Pallottino, S., 341
paradosso del compleanno, 172
permutazione, 428
Pettie, S., 317, 341
Phillips, S. J., 341
pila, 68
operazione
isEmpty, 68
pop, 68

push, 68
top, 68
Pratt, V., 124
Prim
algoritmo di, 308, 311
Prim, R. C., 317
priority search tree, 382
problema
NP-COMPLETO, 405
di decisione, 398
di ottimizzazione, 398
di ricerca, 398
programmazione dinamica, 9, 237, 242–253, 259, 260

Radzik, T., 341
Ramachandran, V., 317
Raman, R., 209
range tree, 380
Rao, S., 362
record e puntatori, 66–68
relazione
di equivalenza, 288, 289
di ricorrenza, 4, 7, 36, 40, 128
rete, 347–363
di flusso, 347–363
residua, 351
Ricci, P., 264
ricerca
albero binario di, 134
albero di, 133
binaria (o dicotomica), 34
in spazi multidimensionali, 377
sequenziale, 30
ricorrenza
relazioni di, 36
ricorrenze
teorema fondamentale delle, 40, 41
ricorsione
albero della, 40
riducibilità polinomiale, 406
rilassamento
passo di, 325
tecnica del, 325
Rivest, R. L., 80, 123, 124
rotazione
DD, SS, DS, SD, 142
di base, 142
doppia, 142
nell'operazione splay, 147
semplice, 142

scansione del piano, 387
Seidel, R., 341
selezione
del mediano, 119–129
per piccoli valori di k, 116
problemi di, 115
serie
aritmetica, 421
armonica, 424
geometrica, 421
sezione aurea ϕ, 4
Shaw, J. C., 79
Shoshan, A., 341
Shrairman, R., 209
Simon, H. A., 79
Sleator, D. D., 146
soddisfacibilità, 400
sostituzione
metodo della, 38
sottografo
di un grafo, 266, 301
indotto di un grafo, 267
sottostruttura ottima, 252
principio di, 252
stabilità di algoritmi di ordinamento, 105
Stein, C., 80
Strassen, V., 341
stringa di lunghezza n, 428
struttura dati, 61–80
collegata, 66–68
indicizzata, 62–66
suddivisione planare, 372

taglio
in un grafo, 303, 358
Takaoka, T., 341
Tarjan, R. E., 58, 124, 146, 209, 233, 235, 317, 341, 362