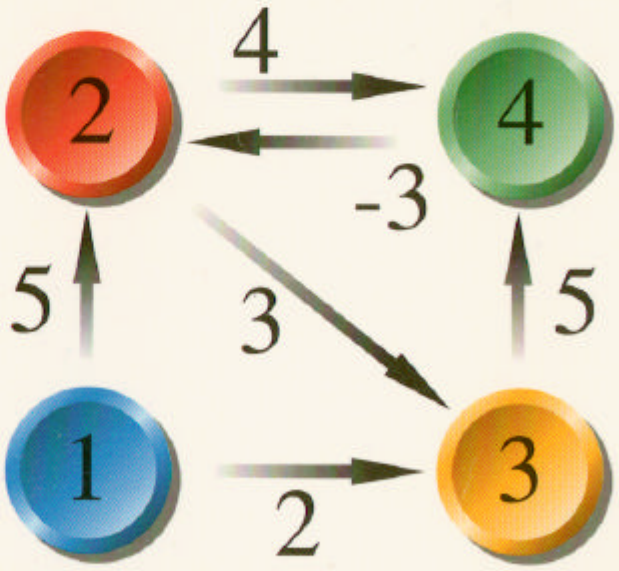
ALAN BERTOSSI

ALGORITMI e STRUTTURE DI DATI

Una presentazione organica e rigorosa.

Un corso completo racchiuso in un solo volume.

250 esercizi d'esame di cui 60 svolti.



UTET
LIBRERIA

ALAN BERTOSSI
Professore ordinario di Informatica
Università di Trento

ALGORITMI E STRUTTURE DI DATI

DIPARTIMENTO DI BIOLOGIA
BIBLIOTECA
SEZ. BOTANICA

UTET
LIBRERIA

UTET Libreria Srl
via P. Giuria, 20 - 10125 Torino

© 2000 UTET Libreria Srl

www.utetlibreria.it
e-mail: commerciale@utetlibreria.it

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

L'Editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume e fino a un massimo di settantacinque pagine. Le richieste di riproduzione vanno inoltrate all'Associazione Italiana per i Diritti di Riproduzione delle Opere dell'ingegno (AIDRO), via delle Erbe 2 - 20121 Milano. Telefono e fax 02/809506.

Stampa: Stampatre - Torino

Prima edizione: luglio 2000

Ristampe: 0 1 2 3 4 5 6 7 8 9
2000 2001 2002 2003 2004

Indice

Prefazione **XI**

Introduzione **3**
I.1 Algoritmi . . . 3
I.2 Livello di dettaglio . . . 4
I.3 Procedure . . . 4
I.4 Ricorsione . . . 6
I.5 Tipi di dato . . . 6
I.6 Il linguaggio Pascal . . . 8
I.7 Tempo di calcolo . . . 10
I.8 Ordine di grandezza e complessità . . . 13
I.9 Algoritmi efficienti e inefficienti . . . 15
I.10 Complessità di problemi e algoritmi ottimi . . . 18
I.11 Sommario . . . 19
I.12 Esercizi . . . 20

Parte prima Strutture di dati **23**

1 Tipi di dato e strutture di dati **25**
1.1 Strutture di dati: specifica e realizzazione . . . 26
1.2 Rappresentazione in memoria . . . 29
1.3 Puntatori . . . 31
1.4 Esercizi . . . 34

2 Liste **35**
2.1 Specifica . . . 35
2.2 Realizzazione con puntatori . . . 38
2.3 Realizzazione con cursori . . . 42
2.4 Esercizi . . . 45

3 Pile e code **48**
3.1 Specifiche . . . 49
3.2 Realizzazione di una pila con vettore . . . 50
3.3 Realizzazione di una coda con vettore circolare . . . 51
3.4 Pile e procedure ricorsive . . . 53
3.5 Esercizi . . . 59

4 Alberi **60**
4.1 Specifica . . . 63
4.2 Visite . . . 66
4.3 Realizzazione con vettore dei padri . . . 67
4.4 Realizzazione con liste dei figli . . . 68
4.5 Realizzazione con puntatori padre/primofiglio/fratello . . . 69
4.6 Alberi binari . . . 71
4.7 Esercizi . . . 75

5 Insiemi **80**
5.1 Specifica . . . 81
5.2 Realizzazione con vettore booleano . . . 82
5.3 Realizzazione con liste non ordinate . . . 83
5.4 Realizzazione con liste ordinate . . . 84
5.5 Mfset e unione di insiemi disgiunti . . . 85
5.6 Esercizi . . . 90

6 Dizionari **93**
6.1 Realizzazione con vettore ordinato . . . 93
6.2 Realizzazione con tabella hash . . . 96
6.2.1 Funzioni hash . . . 98
6.2.2 Metodi di scansione . . . 101
6.2.3 Complessità media . . . 104
6.2.4 Ristrutturazione . . . 106
6.3 Esercizi . . . 107

7 Code con priorità **110**
7.1 Realizzazione con heap . . . 111
7.2 Heapsort . . . 114
7.3 Esercizi . . . 117

8 Alberi bilanciati di ricerca **118**
8.1 Alberi binari di ricerca . . . 119
8.2 Alberi 2-3 . . . 121
8.3 B-alberi binari . . . 122
8.4 B-alberi . . . 129
8.5 Esercizi . . . 132

9 Grafi **135**
9.1 Specifica . . . 139
9.2 Realizzazioni con matrici . . . 140
9.3 Realizzazioni con insiemi di adiacenza . . . 142
9.4 Esplorazione di un grafo . . . 143
9.4.1 Componenti connesse . . . 145
9.4.2 Alberi di copertura DFS . . . 146
9.5 Esercizi . . . 148

Parte seconda Progetto ed analisi di algoritmi **153**

10 Strutture di dati e progetto di algoritmi **155**
10.1 Cammini minimi . . . 156
10.1.1 Algoritmo di Dijkstra . . . 159
10.1.2 Algoritmo di Johnson . . . 159
10.1.3 Algoritmo di Bellman-Ford-Moore . . . 162
10.1.4 Utilizzo di una pila . . . 163
10.1.5 Algoritmo di Pape-D'Esopo . . . 165

10.2 Esercizi . . . 165

11 Divide et impera **167**
11.1 Mergesort . . . 168
11.2 Quicksort . . . 171
11.3 Trasformata veloce di Fourier . . . 176
11.3.1 Radici complesse dell'unità . . . 177
11.3.2 Algoritmo FFT . . . 178
11.3.3 Moltiplicazione di polinomi . . . 180
11.4 Esercizi . . . 182

12 Backtrack **186**
12.1 Inviluppo convesso . . . 186
12.1.1 Algoritmo di Graham . . . 189
12.2 String matching . . . 193
12.2.1 Algoritmo di Knuth, Morris e Pratt . . . 194
12.3 Esercizi . . . 198

13 Greedy **200**
13.1 Minimo albero di copertura . . . 201
13.2 Scheduling di programmi . . . 204
13.3 Matroidi . . . 205
13.4 Esercizi . . . 207

14 Programmazione dinamica **210**
14.1 String matching approssimato . . . 212
14.2 Cammini minimi tra tutte le coppie . . . 214
14.2.1 Quadratura di matrici . . . 214
14.2.2 Algoritmo di Floyd . . . 215
14.3 Esercizi . . . 218

15 Ricerca locale **221**
15.1 Shellsort . . . 224
15.2 Flusso massimo . . . 228
15.2.1 Algoritmo dei 3 indiani . . . 230
15.3 Esercizi . . . 234

16 Analisi di algoritmi **238**
16.1 Relazioni di ricorrenza lineari a coefficienti costanti . . . 239
16.1.1 Moltiplicazione di matrici . . . 243
16.2 Analisi per tentativi . . . 244
16.2.1 Selezione . . . 245
16.3 Potenziale . . . 247
16.3.1 Algoritmi "on line" . . . 249
16.4 Esercizi . . . 250

Parte terza Complessità e decidibilità **253**

17 Non determinismo ed enumerazione **255**
17.1 Certificati polinomiali . . . 257
17.2 Non determinismo . . . 258
17.3 Enumerazione . . . 261
17.4 Esercizi . . . 264

18 Problemi NP-completi **267**
18.1 Le classi $\mathbb{P}$ ed $\mathbb{NP}$ . . . 267
18.2 Riducibilità polinomiale . . . 269
18.3 Teorema di Cook-Levin . . . 271
18.4 Prove di NP-completezza . . . 272
18.5 Esercizi . . . 279

19 Gerarchia di complessità **281**
19.1 Problemi NP-intermedi . . . 282
19.2 La classe co-$\mathbb{NP}$ . . . 282
19.3 Gerarchia polinomiale . . . 285
19.4 La classe $\mathbb{P}$-$\mathbb{SPAZIO}$ . . . 286
19.5 Le classi $\mathbb{EXP}$ e $\mathbb{NEXP}$ . . . 287
19.6 Le classi $\mathbb{EXP}$-$\mathbb{SPAZIO}$ ed $\mathbb{E}$ . . . 289
19.7 Esercizi . . . 289

20 Algoritmi probabilistici **291**
20.1 Primalità . . . 292
20.2 Alberi randomizzati di ricerca . . . 293
20.3 Classi probabilistiche di complessità . . . 297
20.4 Esercizi . . . 299

21 Algoritmi pseudo-polinomiali **300**
21.1 NP-completezza "forte" . . . 301
21.2 Zaino . . . 303
21.3 Real-time scheduling . . . 303
21.4 Esercizi . . . 307

22 Algoritmi di approssimazione **308**
22.1 Approssimazione assoluta . . . 308
22.1.1 Commesso viaggiatore con diseguaglianze triangolari . . . 309
22.1.2 Scheduling multiprocessore . . . 311
22.1.3 Non approssimabilità . . . 314
22.2 ϵ-approssimazione . . . 315
22.3 Esercizi . . . 318

23 Algoritmi branch-&-bound **319**
23.1 Commesso viaggiatore . . . 321
23.2 Trasmissione via satellite . . . 324
23.3 Esercizi . . . 328

24 Algoritmi euristici **329**
24.1 Ricerca tabù . . . 331
24.1.1 Proibizione fissa . . . 333
24.1.2 Proibizione randomizzata . . . 334
24.1.3 Proibizione reattiva . . . 335
24.2 Esercizi . . . 336

25 Indecidibilità ed universalità **339**
25.1 Il problema della terminazione . . . 340
25.2 Gerarchia di indecidibilità . . . 344
25.3 Esercizi . . . 347

26 Macchine di Turing **349**
26.1 L'analisi di Turing . . . 349
26.2 La Tesi di Church-Turing . . . 352
26.3 Generalizzazioni e restrizioni . . . 354
26.4 Riduzioni mediante macchine di Turing . . . 356
26.5 Esercizi . . . 359

Parte quarta Algoritmi paralleli **361**

27 Algoritmi PRAM **363**
27.1 Somme prefisse . . . 364
27.2 Ordinamento . . . 366
27.3 Elaborazione di puntatori . . . 367
27.4 La Tesi della Computazione Parallela . . . 370
27.5 Esercizi . . . 371

28 Algoritmi per reti a grado limitato **375**
28.1 Reti di interconnessione . . . 376
28.2 Sommatoria . . . 379
28.3 Mergesort bitonico . . . 382
28.4 Moltiplicazione di matrici . . . 385
28.5 Trasformata di Fourier . . . 390
28.6 Esercizi . . . 392

29 Algoritmi VLSI **395**
29.1 Il modello a griglia . . . 396
29.2 Il solido spazio/tempo . . . 397
29.3 Layout . . . 400
29.4 Ordinamento . . . 404
29.5 Moltiplicazione di matrici . . . 406
29.6 Esercizi . . . 409

30 Algoritmi per mesh riconfigurabili **411**
30.1 Operazioni di base . . . 412
30.2 Ordinamento . . . 416
30.3 Massima sottosequenza comune . . . 417
30.4 Esercizi . . . 419

31 Algoritmi concorrenti **421**
31.1 Massimo . . . 424
31.2 Ordinamento . . . 424
31.3 Cammini minimi . . . 426
31.4 Branch-&-Bound . . . 428
31.5 Esercizi . . . 429

32 Algoritmi distribuiti **430**
32.1 Topologie di reti . . . 432
32.2 Mutua esclusione . . . 432
32.3 Elezione del leader . . . 435
32.4 Minimo albero di copertura . . . 438
32.5 Esercizi . . . 447

Conclusione **449**

Appendici **453**

A Numeri casuali **455**
A.1 Generatore di Lehmer 456
A.2 Realizzazione in Pascal 457
A.3 Prove sperimentali di algoritmi 459
A.4 Esercizi 460

B Il linguaggio C **461**
B.1 Confronto tra Pascal e C 461
B.2 Esercizi 465

Bibliografia **475**

Indice analitico **489**

Prefazione

L'algoritmica è quella branca dell'informatica che riguarda la definizione e il progetto degli algoritmi, l'analisi della loro correttezza ed efficienza, la dimostrazione delle loro inerenti limitazioni e complessità, lo studio delle strutture di dati da essi elaborati, e la definizione di nuovi modelli di calcolo.

Il termine algoritmo indica la descrizione precisa delle azioni che un esecutore deve compiere per giungere alla soluzione di un problema computazionale di qualsiasi genere. Gli algoritmi vengono comunemente descritti tramite programmi, che si avvalgono di costrutti e istruzioni tipici dei linguaggi di programmazione e che devono essere eseguiti da macchine (o calcolatori elettronici). Le proprietà degli algoritmi, però, sono talmente "fondamentali", "generali" e "robuste" da essere virtualmente indipendenti dalle caratteristiche di specifici linguaggi di programmazione o di particolari calcolatori elettronici.

Ovunque si impieghi un calcolatore elettronico occorrono algoritmi corretti ed efficienti per utilizzare il calcolatore al massimo delle sue possibilità. Per esempio, algoritmi sofisticati e veloci sono usati per controllare il volo di aerei, regolare reazioni nucleari, reperire informazioni in archivi di dati, risolvere problemi algebrici e matematici, pagare stipendi a impiegati di aziende, giocare con i "video-giochi", smistare comunicazioni telefoniche, controllare la produzione di catene di montaggio robotizzate, progettare nuovi calcolatori elettronici, riconoscere immagini di oggetti tridimensionali, elaborare testi letterari, comporre musica.

L'algoritmica è molto vasta e riguarda essenzialmente le seguenti aree:

1. Studio delle *caratteristiche dei dati* elaborati dagli algoritmi per risolvere certe classi di problemi e del modo più opportuno di organizzare e strutturare tali dati al fine di agevolarne l'uso da parte degli algoritmi che li elaborano.
2. Studio di *metodologie per il progetto* di algoritmi corretti (cioè che risolvono sempre e solo certe classi di problemi cui si è interessati) ed efficienti (cioè che li risolvono il più velocemente possibile).
3. Studio delle *inerenti limitazioni* dei problemi da risolvere. Molti problemi ben definiti e di indubbia utilità pratica non potranno mai essere risolti da alcun algoritmo in un lasso di tempo "ragionevole" (p.e. la vita di un essere umano), mentre altri problemi non potranno mai essere risolti neanche in un lasso di tempo "irragionevole". Occorre allora individuare metodologie per affrontare tali problemi inerentemente intrattabili.
4. Studio di nuovi *modelli di calcolo* che permettono la progettazione di algorit-

mi che risolvono molto velocemente, se non tutti i problemi, perlomeno alcuni problemi di rilevante interesse pratico.

Il presente testo è suddiviso in quattro parti: Strutture di Dati, Progetto e Analisi di Algoritmi, Complessità e Decidibilità ed Algoritmi Paralleli, che corrispondono alle suddette quattro aree principali dell'algoritmica, ed è rivolto agli studenti dei corsi di Algoritmi e Strutture Dati del Corso di Laurea in Informatica.

Nella Prima Parte sono considerate le principali strutture di dati (liste, pile, code, alberi, insiemi, dizionari, code con priorità, grafi), che sono presentate come tipi astratti di dato, permettendone una trattazione nel contempo sia chiara sia formale. Nella Seconda Parte sono descritte le tecniche di progetto basilari (divide et impera, backtrack, greedy, programmazione dinamica, ricerca locale) e presentati i metodi per la valutazione dell'efficienza di algoritmi (risoluzione di relazioni di ricorrenza, metodo del potenziale per l'analisi ammortizzata). Nella Terza Parte sono considerate tecniche per dimostrare l'intrattabilità di problemi (riduzioni) e per affrontare problemi intrattabili (randomizzazione, approssimazione, branch-&-bound), ed è inoltre accennato un semplice ma potente modello di calcolo (la macchina di Turing) che è molto utile per dimostrare risultati di intrattabilità. Infine, nella Quarta Parte sono considerati modelli di calcolo parallelo, dove è ammessa l'esecuzione contemporanea di più istruzioni dello stesso algoritmo da parte di più esecutori, e sono trattati sia i cosiddetti modelli sincroni (PRAM, reti a grado limitato, VLSI, mesh riconfigurabili) sia quelli asincroni (concorrenti e distribuiti).

Ciascuna parte del presente testo può essere insegnata in una unità didattica (un corso semestrale di circa 30 ore di lezione), così che l'intero testo può coprire quattro unità didattiche fondamentali dei corsi di Algoritmi e Strutture Dati. Per esempio, le prime due parti (Strutture di Dati e Progetto e Analisi di Algoritmi) possono essere insegnate, in quest'ordine, in due unità dei primi due anni del Corso di Laurea in Informatica, mentre la terza (Complessità e Decidibilità) e la quarta (Algoritmi Paralleli) possono essere insegnate, non necessariamente in quest'ordine, in due unità degli anni successivi al secondo.

Una novità di questo testo consiste nel presentare, in modo organico, compatto e rigoroso, nozioni che, essendo usualmente insegnate in corsi universitari distinti, non si trovano riunite in un unico testo ma distribuite tra vari testi. Per esempio, la maggior parte del contenuto della terza e quarta parte non è trattato in altri testi di algoritmi, perché "storicamente" legato a corsi di Calcolabilità (§ 19, 25-26), Ottimizzazione Combinatoria (§ 21-24), oppure Sistemi di Elaborazione (§ 28-32). Nel presente testo ne è effettuata una trattazione algoritmica che rappresenta in modo omogeneo il naturale proseguio delle "classiche" due parti precedenti, evidenziando inoltre le correlazioni esistenti tra le varie discipline (Algoritmica, Logica Matematica, Complessità, Calcolabilità, Ottimizzazione Combinatoria, Sistemi di Elaborazione e Reti di Calcolatori). Inoltre, sono riuniti insieme in un unico testo tutti i principali modelli di calcolo parallelo, concorrente e distribuito, mentre sono riportati per la prima volta in un libro di testo risultati molto recenti e rilevanti, come i "B-alberi binari di ricerca" di Andersson (§ 8), gli "alberi randomizzati di ricerca" di Seidel e Aragon (§ 20), gli algoritmi "on line" (§ 16), lo "scheduling real-time" (§ 21), le euristiche di "ricerca tabù reattiva" (§ 24) ed il modello parallelo delle "mesh riconfigurabili" (§ 30).

In molti libri di testo, vengono presentati algoritmi descritti in linguaggi didattici "immaginari", dove molti dettagli sono omessi o lasciati all'immaginazione del lettore. Gli algoritmi illustrati nel presente testo, invece, oltre ad essere atti a risolvere problemi di indubbia "utilità pratica", sono, nella maggior parte dei casi, descritti

tramite procedure scritte in uno specifico linguaggio di programmazione, il Pascal. Poiché le prime due parti possono essere insegnate nel primo anno del Corso di Laurea, si è preferito usare un linguaggio "vero" anziché uno "immaginario". In questo modo la matricola, ancora poco pratica di programmazione, può eseguire direttamente al calcolatore, sperimentare e modificare gli algoritmi proposti (a tal fine, è stata inserita in appendice una breve descrizione della generazione di numeri casuali, che torna utile per produrre dati di ingresso arbitrari sui quali sperimentare il comportamento degli algoritmi). In ogni caso, le procedure descritte utilizzano solo pochi costrutti fondamentali del Pascal, che si ritrovano praticamente identici in tutti gli altri linguaggi. Comunque, data l'enorme diffusione del linguaggio C, si è fornita in appendice anche una breve descrizione di tale linguaggio, mettendolo a confronto col Pascal e riscrivendo in C alcune delle procedure principali già scritte in Pascal nel testo. Nella quarta parte, invece, poiché sono considerati modelli "astratti" di calcolo parallelo, gli algoritmi sono descritti in un linguaggio "immaginario", comunque vicinissimo al Pascal.

Nel testo sono proposti più di 250 esercizi (per la maggior parte d'esame), dei quali oltre 60 sono completamente svolti, mentre per molti altri sono forniti opportuni suggerimenti. Prima di seguire i suggerimenti o di guardare gli svolgimenti, però, è bene che il lettore provi a risolvere gli esercizi per conto proprio.

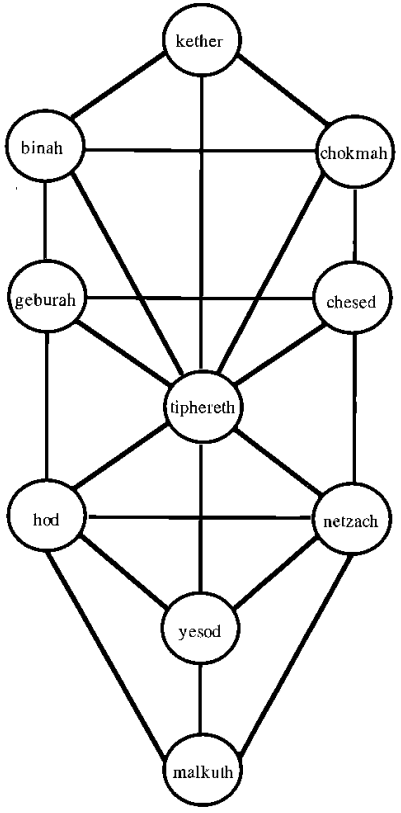
L'autore desidera ringraziare quanti hanno letto e commentato porzioni preliminari del presente testo, suggerendo modifiche e miglioramenti, proponendo nuovi esercizi ed indicando riferimenti bibliografici aggiornati. In particolare, l'autore è riconoscente a Roberto Battiti, Maurizio Bonuccelli, Lorenzo Donatiello, Moreno Falaschi, Paolo Ferragina, Roberto Grossi, Luigi Mancini, Alessandro Mei, Carlo Montangero, Linda Pagli, Cristina Pinotti e Geppino Pucci. L'autore desidera altresì ricordare i numerosi studenti delle Università di Pisa e Trento che hanno fornito molte soluzioni originali agli esercizi proposti, alcune delle quali sono riportate nel testo. In ogni caso, la responsabilità degli errori contenuti nel testo resta dell'autore, che sarà riconoscente a chiunque glieli segnali (anche spedendo un messaggio di posta elettronica all'indirizzo: bertossi@science.unitn.it).

Il presente testo è stato quasi interamente scritto mentre l'autore era ospite dell'Istituto di Elaborazione dell'Informazione del Consiglio Nazionale delle Ricerche (IEI-CNR) di Pisa. L'autore desidera ringraziare Piero Maestrini, direttore di tale istituto, per l'ospitalità concessagli.

Algoritmi e Strutture di Dati

Scacciò l'uomo e pose ad oriente del giardino di Eden i cherubini e la fiamma della spada folgorante, per custodire la via all'albero della vita.

Genesi 3, 24



Il biblico "Legno (o Albero) della Vita" della tradizione cabalistica

Introduzione

I.1 Algoritmi

Il termine algoritmo, che significa procedimento di calcolo, è una versione moderna del termine latino medievale algorismus, che deriva dal nome del matematico usbeco Mohammed ibn-Musa al-Khowarismi, vissuto nel IX secolo d.C. Tale termine indica la descrizione precisa di una sequenza di azioni che un esecutore deve compiere per giungere alla soluzione di un qualsiasi problema computazionale. Mentre l'enunciato del problema specifica quale relazione si desideri tra l'ingresso e l'uscita (cioè il risultato), l'algoritmo descrive la sequenza di azioni da eseguire per ottenere l'uscita desiderata a partire dall'ingresso. Per poter risolvere il problema, l'algoritmo ha bisogno di rappresentare, organizzare ed elaborare l'ingresso, l'uscita, ed altre informazioni intermedie utili al calcolo. Questo è possibile manipolando oggetti elementari, detti dati. In pratica, algoritmi e dati sono inscindibilmente legati. L'algoritmo può essere considerato come un elaboratore di dati che, a fronte di certi dati d'ingresso consistenti con la natura del problema da risolvere (dati di input), li rielabora fino a produrre altri dati come risultato del problema (dati di output).

Esempio I.1 (Ricetta gastronomica). Una cucina di un ristorante è provvista di ingredienti, pentole, fornelli e altri utensili vari. Per cucinare un buon minestrone alla toscana, il cuoco esegue delle azioni specificate da una ricetta. Eccone una per un numero arbitrario p di persone:

> Ingredienti: $75p$ grammi di fagioli, $50p$ grammi di pasta, p patate, $50p$ grammi di cavolo, $50p$ grammi di cardi, $40p$ grammi di pomodori maturi, $p/4$ mazzetti di basilico, $p/4$ cipolle, p pizzichi di sale e pepe, p cucchiai d'olio e p cucchiai di formaggio grana grattugiato.
>
> Mettere in una pentola l'olio, le cipolle tritate e il basilico, lasciare dorare e unire tutte le verdure tagliate a dadini, mescolare, unire p cucchiai d'acqua dei fagioli, precedentemente cotti e passati al setaccio. Aggiungere la purea di fagioli, sale, pepe e continuare la cottura. Dopo un'oretta aggiungere la pasta e lasciarla cuocere per circa sei minuti. Servire con il formaggio grattugiato e, a piacere, con un cucchiaio di olio di oliva di frantoio.

Cucinare il minestrone è un procedimento che viene eseguito dal cuoco, a partire dagli ingredienti, con l'aiuto di pentole, fornelli e altri utensili, e in accordo alla ricetta. Gli ingredienti sono i dati di input, il minestrone è l'output, la ricetta è l'algoritmo e il cuoco è l'esecutore (o elaboratore, o calcolatore). Volendo spingere oltre l'analogia tra la cucina di un ristorante ed un moderno sistema di calcolo, si può dire

che le ricette (o meglio, i testi scritti che le descrivono) rappresentano il cosiddetto software, mentre il cuoco, i fornelli, le pentole e gli altri utensili rappresentano il cosiddetto hardware. □

I.2 Livello di dettaglio

Nel descrivere un algoritmo, si utilizzano azioni elementari, che debbono dare in tempo limitato un risultato certo, unico e ripetibile. In altri termini, l'effetto di un'azione elementare su un dato è determinato univocamente dall'azione e dal dato, cioè non dipende da variabili casuali. Se viene ripetuta la stessa azione sullo stesso dato si ottiene sempre lo stesso risultato. Il ritardo per ottenere il risultato è finito e noto a priori. Inoltre, le azioni debbono essere comprensibili ed eseguibili senza possibilità di ambiguità alcuna da parte dell'esecutore.

Esempio I.1 (continua). Si consideri l'istruzione mescolare. Perché invece non è scritto: "impugnare un mestolo, immergerlo nella pentola, con un angolo di 25° rispetto alla perpendicolare, fino a toccare il fondo, e muovere il braccio in modo rotatorio ad una velocità di circa mezzo metro al secondo"?

La risposta è ovvia: perché il cuoco sa benissimo come si fa a mescolare! Oppure perché non è scritto più semplicemente: "prepara un minestrone alla toscana"? La risposta è altrettanto ovvia: perché questa azione sarebbe eseguibile da chi conoscesse già la ricetta, ma ben difficilmente da una persona che non sapesse come si prepara un minestrone! □

Il livello di dettaglio con cui si specificano le azioni elementari è molto importante. I diversi livelli di dettaglio dipendono dalle capacità dell'esecutore e quando l'esecutore è una macchina elettronica è necessario introdurre moltissimi dettagli. Infatti, anche i più moderni ed evoluti calcolatori elettronici altro non sono che esecutori di algoritmi espressi da lunghe sequenze di bit. Un bit è un oggetto che può assumere uno e un solo valore tra due possibili, indicati di solito con i valori booleani 0 e 1 (o falso e vero). Può sembrare incredibile, eppure le azioni che la macchina può effettuare sono semplicemente verificare il valore di un singolo bit e modificarne il valore! Fortunatamente, per descrivere algoritmi eseguibili da macchine elettroniche non è necessario usare tutti i dettagli e i tecnicismi richiesti dalla manipolazione di sequenze di bit, ma si possono utilizzare costrutti tipici di linguaggi di programmazione ad alto livello. Tali costrutti sono espressi in una forma intermedia tra il linguaggio discorsivo tipico degli esseri umani, che purtroppo spesso ha un significato ambiguo e i comandi specifici di una macchina che "lavora" sui bit, che non sono ambigui ma troppo "tecnici" e dettagliati e quindi di difficile interpretazione da parte di un essere umano. I costrutti dei linguaggi di programmazione ad alto livello permettono concisione e chiarezza e hanno carattere di universalità (non dipendono dalla struttura circuitale di alcuna specifica macchina elettronica), ma nel contempo possono essere tradotti automaticamente (cioè tramite l'esecuzione di altri algoritmi, detti compilatori) in sequenze di bit eseguibili direttamente dai circuiti di un vero calcolatore elettronico.

I.3 Procedure

Un algoritmo descritto per mezzo di costrutti tipici di un linguaggio di programmazione è comunemente detto procedura.

Esempio I.2 (Mangia la minestra). Si consideri un cliente del ristorante a cui è stata servita una razione di minestrone alla toscana. Per gustare la pietanza, il cliente impugnerà un cucchiaio e mangerà una cucchiaiata di minestrone per volta, fino a svuotare il piatto. Utilizzando costrutti tipici di molti linguaggi di programmazione, tale procedimento può essere descritto informalmente con la procedura seguente:

procedure MANGIALAMINESTRA(*minestra*: minestrainbrodo);
  begin
    while il piatto non è vuoto **do** mangia una cucchiaiata di *minestra*;
  end; □

I termini in grassetto indicano parole chiave (cioè riservate) del linguaggio. La procedura MANGIALAMINESTRA indica un algoritmo che può essere utilizzato con dato di ingresso arbitrario (il parametro formale), chiamato all'interno della procedura col nome *minestra*. Questo nome indica una variabile di tipo minestrainbrodo e la procedura può essere richiamata con valori del parametro attuale del medesimo tipo, ad esempio:

MANGIALAMINESTRA(*zuppa di cipolle*),
MANGIALAMINESTRA(*crema di funghi*).

Le istruzioni che descrivono le azioni da compiere sono racchiuse tra i delimitatori **begin** e **end**, separate tra loro da ";" e devono essere interpretate ed eseguite sequenzialmente, cioè una alla volta nell'ordine in cui vengono lette. Sono usate azioni "elementari" come controllare che il piatto sia vuoto e mangiare una cucchiaiata di minesta (facilmente eseguibili da un essere umano, un po' più difficilmente da una macchina elettronica priva di apparato digerente...). Inoltre, è usato un costrutto fondamentale dei linguaggi di programmazione: l'iterazione illimitata. Tale costrutto ha la forma generale

"**while** condizione **do** istruzione"

e permette di ripetere iterativamente l'istruzione. La ripetizione prosegue se la condizione è verificata e si arresta non appena la condizione non è più verificata.

Il testo della procedura che descrive un algoritmo è strutturato come una sequenza di azioni (o passi) elementari tra loro distinti ed ha lunghezza finita e costante per tutti i possibili dati di ingresso. D'altro canto, l'algoritmo è generale ed applicabile ad una infinità di dati d'ingresso, di grandezza arbitraria, ed il tempo necessario per eseguire la procedura può variare a seconda della grandezza dei dati di ingresso.

Esempio I.3 (Descrizioni brevi di un'infinità di processi lunghi). Il numero p di persone per la ricetta del minestrone alla toscana e la quantità di minestra per la procedura MANGIALAMINESTRA possono essere arbitrariamente grandi. Il testo degli algoritmi rimane sempre lo stesso, ma ne può variare il tempo di esecuzione (si mangia più in fretta una ciotolina di minestrone piuttosto che un barile!). □

Specificare una procedura permette di definire un algoritmo una volta per tutte, di poterlo individuare con un nome e di poterlo utilizzare più volte, con valori distinti e arbitrari, purché consistenti, dei dati d'ingresso. Ciò ha un duplice vantaggio:

1. si evita di riscrivere più volte un algoritmo per effettuare la stessa computazione con valori diversi dei dati;
2. è possibile sviluppare algoritmi complessi gradualmente, scrivendo dapprima procedure che, utilizzando costrutti elementari del linguaggio, risolvono sottoproblemi più semplici, e successivamente procedure più complesse che utilizzano le procedure scritte precedentemente come se quest'ultime fossero nuove istruzioni elementari.

In questo modo, una procedura A per la risoluzione di un certo problema P può richiamare al suo interno una procedura B che risolve un sottoproblema Q, che a sua volta può richiamare C per risolvere R, e così via per un numero arbitrario di livelli di chiamate. Al momento della chiamata, l'esecuzione di A viene temporaneamente sospesa per permettere l'esecuzione di B, sui dati fornitigli da A. L'esecuzione di A viene poi ripresa, al termine dell'esecuzione di B, a partire dall'istruzione seguente alla chiamata.

I.4 Ricorsione

Il meccanismo di chiamate tra procedure può essere anche ricorsivo, nel senso che una procedura A può richiamare se stessa al suo interno!

Esempio I.4 (Mangia la minestra ricorsivamente). Per gustare il minestrone, si può definire la seguente procedura ricorsiva:

```
procedure MANGIALAMINESTRA(minestra: minestrainbrodo);
  begin
    if il piatto non è vuoto then
      begin
        mangia una cucchiaiata di minestra;
        MANGIALAMINESTRA(minestra)
      end
  end;                                                    □
```

Nella procedura MANGIALAMINESTRA si è usato il costrutto condizionale:

"**if** condizione **then** istruzione"

che comanda l'esecuzione dell'istruzione purché la condizione sia vera. La sua forma più generale comprende anche un'istruzione alternativa, da eseguire qualora la condizione sia falsa, ed ha la forma

"**if** condizione **then** istruzione1 **else** istruzione2"

come, ad esempio, nell'istruzione seguente:

```
if il piatto non è vuoto
  then mangia una cucchiaiata di minestra
  else salta dalla finestra;
```

L'apparente circolarità di una procedura ricorsiva (per eseguire A si richiama ancora A!) è spezzata da una condizione di chiusura, che interrompe l'esecuzione della procedura quando i dati raggiungono opportuni valori (nell'esempio, quando il piatto è vuoto). Altrimenti, la procedura richiama se stessa, sospendendo temporaneamente la vecchia computazione e iniziando una nuova computazione su dati che hanno sì lo stesso nome (nell'esempio: "minestra"), ma valori diversi (nell'esempio: la quantità di minestra è diminuita di una cucchiaiata).

I.5 Tipi di dato

Nello scrivere una procedura in un linguaggio di programmazione, si assume che certi tipi di dato "primitivi" (di solito: interi, booleani, reali, caratteri) siano forniti direttamente dal linguaggio. Può tuttavia presentarsi la necessità di voler definire

ed utilizzare, oltre ai tipi di dato primitivi, nuovi tipi di dato, che siano in qualche modo più vicini alla natura del problema che si vuole risolvere. Per definire nuovi tipi di dato, è sufficiente specificarne un nome, l'insieme dei valori, e stabilire quali operazioni peculiari possono manipolare tali valori. In questo modo, l'elaborazione dei nuovi tipi di dato viene a dipendere soltanto dalle caratteristiche dei dati stessi e non dalla loro rappresentazione che, al limite, può essere addirittura "ignorata".

Esempio I.5 (Tipo di dato insieme). Il tipo di dato insieme consiste di una collezione di elementi dello stesso tipo sui quali sono possibili le ben note operazioni di inserzione, unione, e intersezione. Se sull'insieme è definibile una relazione d'ordinamento totale (per esempio su un insieme di interi), allora è possibile anche l'operazione di ricerca del minimo. Ad un livello "astratto" di dettaglio non è specificato, perché non interessa affatto, come sia rappresentato il tipo di dato "insieme" ed il problema consiste semplicemente nel realizzare le operazioni astratte ivi definite. Per esempio, consideriamo l'operazione astratta MIN(A), che ritorna come valore un intero uguale al minimo elemento dell'insieme A, assumendo che A ne contenga almeno uno. Per realizzare MIN si possono esaminare uno per uno gli elementi di A, registrando via via il minimo tra tutti gli elementi esaminati, fino ad esaurire tutti gli elementi. Ecco una "funzione" che effettua tale calcolo (una funzione è una particolare procedura, che restituisce un valore):

```
function MIN(A: insieme): intero;
  utilizza una variabile di nome m e di tipo intero;
  begin
    considera un elemento qualsiasi di A;
    assegna ad m il valore di tale elemento;
    while A contiene ancora elementi non considerati do
      begin
        considera un altro elemento di A;
        if il valore di tale elemento è minore di m
          then assegna ad m il valore di tale elemento
      end;
    assegna a MIN il valore di m
  end;                                                    □
```

Come le procedure (p.e. la MIN) generalizzano le operazioni primitive fornite dal linguaggio (di solito sono primitive le operazioni aritmetiche e logiche: $+$, $-$, $/$, $*$, **div**, **mod**, **and**, **or**, **not**, ecc.), così i tipi di dato generalizzano i tipi primitivi (interi, booleani, reali, caratteri, ecc.). Per essere utilizzabili, però, i dati non primitivi devono essere "simulati" per mezzo di dati primitivi o di altri tipi di dato non primitivi ma già disponibili.

Esempio I.5 (continua). Si supponga di avere a disposizione un tipo di dato "lista", che consiste in una sequenza di elementi dello stesso tipo, ciascuno caratterizzato da una posizione nella sequenza, su cui sono definite le operazioni seguenti:

PRIMOLISTA(L), che restituisce il primo elemento della sequenza L,

SUCCLISTA(L), che restituisce l'elemento successivo a quello su cui si è attualmente posizionati, se ce n'è uno, e vuoto altrimenti.

Se il tipo di dato insieme è rappresentato per mezzo del tipo di dato lista, la funzione MIN può essere realizzata come segue:

```
function MIN(A: insieme): intero;
    utilizza due variabili di nome m e k, di tipo intero;
    begin
        assegna ad m e a k il valore di PRIMOLISTA(A);
        while il valore di k è diverso da vuoto do
            begin
                assegna a k il valore di SUCCLISTA(A)
                if il valore di k è diverso da vuoto ed è minore di m
                    then assegna ad m il valore di k;
            end;
        assegna a MIN il valore di m
    end;                                                    □
```

Questo esempio illustra un aspetto importante dei tipi di dato. Si utilizzano operazioni di un tipo di dato (nell'esempio: la lista) senza conoscerne la realizzazione. Se si modifica la realizzazione del tipo di dato, le procedure che usano gli oggetti di quel tipo non cambiano (nell'esempio: MIN). Solo le procedure e funzioni che realizzano le operazioni specifiche del dato in questione (nell'esempio: PRIMOLISTA e SUCCLISTA) devono essere modificate qualora si cambi la realizzazione del tipo di dato.

I.6 Il linguaggio Pascal

In seguito, gli algoritmi proposti saranno descritti nel linguaggio Pascal sotto forma di procedure e funzioni. Poiché non è scopo di questo testo insegnare il Pascal, i costrutti principali di tale linguaggio, i dati primitivi da esso forniti e i meccanismi per il passaggio di parametri saranno introdotti solo con alcuni semplici esempi. La descrizione di un linguaggio richiede una precisa sintassi che specifichi la forma delle istruzioni ammesse ed una precisa semantica che ne dia il significato qualora esse siano eseguite: il lettore è incoraggiato ad apprendere tutti i segreti sintattici e semantici del Pascal su un buon manuale!

Esempio I.5 (continua). Mentre in Pascal gli interi (al pari dei reali, caratteri, booleani, puntatori) sono dati primitivi, gli insiemi (a parte una lieve eccezione che poi vedremo) non lo sono e per lavorare con essi occorre "simularli" con altri dati primitivi già disponibili. Supponiamo per semplicità di conoscere il numero n di elementi di A, che indichiamo con $A[1], A[2], \ldots, A[n]$, sotto un ordinamento qualsiasi. L'insieme A può essere rappresentato mediante il dato primitivo Pascal **array**, che permette di riunire con un unico nome un numero prefissato di elementi dello stesso tipo e di individuare i singoli elementi mediante un indice. Per esempio, assumendo per semplicità che $1 \leq n \leq 50$, si hanno le seguenti istruzioni Pascal:

```
const maxlung = 50;
 type insieme = array [1..maxlung] of integer;
var A: insieme; n: integer;
```

Le istruzioni precedenti sono dichiarazioni, in cui appunto si dichiara che "*maxlung*" è un nome utilizzato per indicare la costante 50, che "insieme" è un tipo di dato rappresentato con un array di *maxlung* interi, i cui elementi possono essere acceduti direttamente tramite un indice che può assumere un qualsiasi valore compreso nell'intervallo tra 1 e *maxlung*. Infine, A ed n sono i nomi di due variabili di tipo insieme e intero, rispettivamente, le quali potranno assumere valori dei rispettivi tipi.

Una funzione iterativa MIN per trovare l'elemento minimo dell'insieme A può essere allora scritta in Pascal nel modo seguente:

```
function MIN(var B: insieme; j, k: integer): integer;
    var m, i: integer;
    begin
        m := B[j];
        for i := j + 1 to k do
            if B[i] < m then m := B[i];
        MIN := m
    end;
```

La funzione sarà richiamata specificandone il nome seguito dalla lista dei parametri attuali racchiusi tra parentesi tonde e separati da virgole: MIN(A, 1, n). □

Nell'intestazione di una procedura Pascal, i parametri formali (nell'esempio: B, j, k) possono essere preceduti dalla parola chiave **var** o no. Nel primo caso (nell'esempio: B, di tipo insieme), si indicano con tali nomi oggetti del mondo esterno alla procedura, dei quali possono essere variati i valori all'interno della procedura stessa: tale meccanismo di passaggio dei parametri è detto "per variabile" (poiché di fatto il nome B diventa un sinonimo di A, ed entrambi individuano la stessa variabile). Nel secondo caso (nell'esempio: j e k, di tipo intero) eventuali modifiche del valore del parametro formale non influiscono sul mondo esterno: questo passaggio è detto "per valore" (poiché k assume inizialmente lo stesso valore di n, ma individua una variabile distinta da n). Subito dopo il **begin** iniziale vengono dichiarate le variabili (**var**) e le eventuali costanti (**const**), procedure e funzioni che saranno utilizzate nel corpo della procedura. Nel corpo della procedura possono essere usati costrutti condizionali, di iterazione illimitata e limitata. L'iterazione limitata ha la forma generale:

"**for** indice := estremoinferiore **to** estremosuperiore **do** istruzione"

che è un modo più compatto di scrivere:

```
"indice := estremoinferiore;
 while indice ≤ estremosuperiore do
     begin
         istruzione;
         indice := indice + 1
     end"
```

L'istruzione

"variabile := espressione"

indica un assegnamento alla variabile del valore dell'espressione, che ovviamente deve essere dello stesso tipo di quello della variabile. I delimitatori **begin** ed **end** sono usati per racchiudere una sequenza di istruzioni in un unico blocco, da considerarsi come una sola istruzione composita.

Oltre a procedure iterative, in Pascal è possibile scrivere anche procedure ricorsive, come mostrato nel seguente esempio.

Esempio I.5 (continua). Si supponga sempre di rappresentare l'insieme A con un **array**, i cui elementi hanno indici compresi tra 1 ed n, con $n \geq 1$. Una versione ricorsiva di MIN per trovare l'elemento minimo dell'insieme A può essere scritta in Pascal come segue:

```
function MIN(var B: insieme; j, k: integer): integer;
    var m: integer;
    begin
        if j = k then
            MIN := B[k]
        else begin
            m := MIN(B, j + 1, k);
            if B[j] < m then MIN := B[j] else MIN := m
        end
    end;
```

La funzione, che è richiamata come prima con l'istruzione MIN(A, 1, n), suddivide l'array in due porzioni, quella contenente soltanto il primo elemento $B[j]$, e quella contenente i rimanenti elementi $B[j+1], \ldots, B[k]$, se ci sono. Se j è uguale a k, allora vi è un solo elemento, che è anche il minimo; altrimenti, la procedura richiama se stessa sulla seconda porzione, e decide successivamente se il minimo complessivo era proprio $B[j]$ o il minimo elemento della seconda porzione. □

I.7 Tempo di calcolo

Per risolvere un problema, spesso sono disponibili molti algoritmi diversi. Come scegliere il "migliore"? Un criterio generalmente adottato consiste nel valutare la "bontà" di un algoritmo in base alla quantità di risorsa utilizzata per il calcolo. La risorsa di interesse dominante è il tempo richiesto per eseguire le azioni elementari, ma talvolta è considerato anche lo spazio necessario per immagazzinare (o memorizzare) e manipolare i dati.

In pratica, non è possibile misurare il tempo di calcolo di un algoritmo col numero di secondi richiesto per eseguire su un calcolatore elettronico una procedura che lo descriva. Infatti tale tempo dipende pesantemente dai dati di ingresso, dal linguaggio in cui la procedura è descritta, dalla qualità con cui viene automaticamente tradotta dal compilatore in una sequenza di bit e dalla natura e velocità del calcolatore elettronico. Occorre una misura "astratta" che tenga maggiormente conto del "metodo di risoluzione" con cui l'algoritmo effettua la computazione.

Poiché i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati di ingresso, viene spontaneo esprimere il tempo di calcolo come il costo complessivo delle operazioni elementari in funzione della dimensione n dei dati di ingresso. Sono considerate elementari le operazioni aritmetiche, logiche, di confronto e di assegnamento. Talvolta, è utile considerare non tutte le operazioni, ma solo quelle dominanti, cioè quelle che incidono maggiormente sul tempo di esecuzione. Il costo delle operazioni è principalmente valutato nel caso pessimo, cioè sul dato d'ingresso più sfavorevole, tra tutti quelli di dimensione n, ma talvolta è valutato anche nel caso medio, cioè mediando su tutti i possibili dati di dimensione n, tenendo conto della probabilità con cui ciascun dato può occorrere.

Esempio I.6 (Tempo di calcolo di MIN iterativa). Per valutare il tempo di calcolo della funzione iterativa MIN, si osservi che la funzione MIN è formata da un numero finito di istruzioni elementari. Ogni istruzione richiede un tempo costante di esecuzione, ma la costante può essere diversa da istruzione a istruzione. Indichiamo con c_h la costante richiesta per l'esecuzione dell'istruzione h-esima. Rivediamo la funzione MIN con indicato per ogni istruzione il costo per eseguirla una volta ed il numero di volte che l'istruzione viene eseguita complessivamente. Tenendo conto

che la dimensione del problema è n, numero di elementi dell'array A, poiché MIN è richiamata con $j = 1$ e $k = n$, si ottiene:

	costo	*volte*
function MIN(**var** B: insieme; j, k: integer): integer;	c_1	1
var m, i: integer;	0	1
begin		
$m := B[j]$;	c_2	1
for $i := j + 1$ **to** k **do**	c_3	n
if $B[i] < m$ **then**	c_4	$n-1$
$m := B[i]$;	c_5	$n-1$
MIN $:= m$	c_6	1
end;		

Si noti che l'incremento dell'indice i nel ciclo **for** è ripetuto n volte e non $n-1$. Infatti il **for**, come spiegato precedentemente, è un modo più compatto di scrivere un **while**, nel quale la condizione $i \leq k$ deve essere verificata una volta in più per poter uscire dal ciclo quando i supera k. Il tempo di calcolo $T(n)$ di MIN si ottiene sommando su tutte le istruzioni il prodotto del costo di ciascuna istruzione per il numero di volte che tale istruzione è eseguita:

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 n + c_4(n-1) + c_5(n-1) + c_6 = \\ &= (c_3 + c_4 + c_5)n + (c_1 + c_2 + c_6 - c_4 - c_5). \end{aligned}$$

Pertanto, il tempo di calcolo di MIN può essere espresso come

$$T(n) = an + b,$$

con a e b costanti intere positive. Si noti che questo andamento di $T(n)$ lineare in n vale sia nel caso pessimo che in quello medio, poiché il corpo del **for** è sempre ripetuto esattamente $n-1$ volte, qualsiasi sia il dato A d'ingresso. □

Esempio I.7 (Tempo di calcolo di MIN ricorsiva). Valutare il tempo della versione ricorsiva di MIN è un po' più complicato. Si indichi ancora con $T(n)$ il tempo richiesto per un dato d'ingresso di dimensione n. Si assuma dapprima che $n = 1$ (cioè che $j = k$). In tal caso la funzione esegue direttamente la condizione di chiusura:

	costo	*volte*
function MIN(**var** B: insieme; j, k: integer): integer;	c_1	1
var m: integer;	0	1
begin		
if $j = k$ **then**	c_2	1
MIN $:= B[k]$	c_3	1
else		
...		0
end;		

Se vi è un solo elemento, il tempo di calcolo è $c_1 + c_2 + c_3$ e quindi:

$$T(1) = c$$

dove c è una costante. Altrimenti, la funzione suddivide l'array in due porzioni, la prima contenente un solo elemento, e la seconda contenente i rimanenti $n-1$ elementi, e richiama se stessa sulla seconda porzione, la cui dimensione è $n-1$:

BIBLIOTECA SALA DOCENTI

```
                                                   costo      volte
function MIN(var B: insieme; j, k: integer): integer;   c1         1
    var m: integer;                                     0          1
    begin
        if j = k then                                   c2         1
            MIN := B[k]                                 c3         0
        else begin
            m := MIN(B, j + 1, k);                      c4 + T(n - 1)  1
            if B[j] < m                                 c5         1
                then MIN := B[j]                        c6         1
                else MIN := m                           c7         1
        end
    end;
```

dove la costante c_4 include sia il costo dell'incremento del parametro j che quello dell'assegnamento ad m, mentre $T(n-1)$ è il tempo di calcolo dovuto alla chiamata ricorsiva. Tenendo inoltre conto che solo uno dei due assegnamenti finali su MIN è eseguito, si ha:

$$\begin{aligned} T(n) &= T(n-1) + c_1 + c_2 + c_4 + c_5 + \max\{c_6, c_7\} = \\ &= T(n-1) + d \end{aligned}$$

dove d è una costante. Il tempo di calcolo $T(n)$ può essere dunque ricavato dalla soluzione delle seguenti relazioni, dette relazioni di ricorrenza:

$$\begin{aligned} T(n) &= c && \text{se } n = 1, \\ T(n) &= T(n-1) + d && \text{se } n > 1. \end{aligned}$$

Una tecnica utile per risolvere relazioni di ricorrenza consiste nel produrre una catena di uguaglianze ottenute per sostituzioni successive, applicando le relazioni per $n-1, n-2, \ldots, 2, 1$. Si ottiene:

$$\begin{aligned} T(n) &= T(n-1) + d \\ &= T(n-2) + 2d \\ &= T(n-3) + 3d \\ &\ldots \\ &= T(1) + (n-1)d \\ &= dn + (c-d). \end{aligned}$$

Quindi la versione ricorsiva di MIN richiede un tempo di calcolo che, a meno dei valori delle costanti d e $c-d$ che possono differire da a e b, cresce linearmente in n come il tempo di calcolo della versione iterativa di MIN. □

In pratica, non è necessario tener conto esplicitamente dei costi delle singole operazioni. Infatti, ha poca importanza che una moltiplicazione sia, mettiamo, dieci volte più lenta di una addizione, quando alla fin fine entrambe le operazioni richiedono tempi costanti, seppur diversi. Di fatto, tutte le costanti delle singole operazioni possono venire "inglobate" in poche costanti, per esempio a e b, quando si conclude che il tempo di calcolo $T(n)$ cresce linearmente in n, cioè come $an + b$. Pertanto,

il tempo di calcolo può essere valutato semplicemente contando il numero di operazioni elementari eseguite nel caso pessimo (o medio) su tutti i dati di ingresso di dimensione n.

Una obiezione che può essere fatta alla valutazione del tempo di calcolo nel caso pessimo è che sia appunto troppo "pessimistica", perché il caso pessimo potrebbe verificarsi molto raramente. Tale valutazione presenta però l'indubbio vantaggio di garantire che l'algoritmo non richiederà mai, per nessun dato di ingresso di dimensione n, un tempo maggiore. Inoltre il caso pessimo, per certi problemi, occorre molto di frequente. Per esempio, nel verificare se un dato elemento appartiene oppure no ad un insieme, il caso pessimo occorre ogni volta che l'elemento non è presente nell'insieme, perché è necessario esaminarne tutti gli elementi prima di accorgersene! Inoltre, benché la valutazione nel caso medio possa apparire più realistica, spesso non è chiaro sotto quali ipotesi di distribuzione di probabilità debba essere svolta. In genere viene assunta una distribuzione uniforme (in cui tutti i casi sono equiprobabili), che per molti problemi è irrealistica o porta allo stesso risultato, a meno di costanti, del caso pessimo. Per esempio, assumendo che un dato elemento appaia come i-esimo in un insieme di n elementi con la stessa probabilità $1/n$, per reperirlo occorre un tempo lineare in n sia nel caso medio che in quello pessimo, perché bisogna esaminare $n/2$ elementi in media ed n nel caso pessimo. Inoltre, la valutazione nel caso medio richiede l'impiego di formule matematiche molto complicate e di difficilissima risoluzione. A parte poche eccezioni, la valutazione del tempo di calcolo principalmente considerata è quella nel caso pessimo. Talvolta è anche considerato il cosiddetto tempo ammortizzato, dove il tempo $T(m)$ richiesto nel caso pessimo per eseguire una sequenza di m operazioni (per esempio, ricerche, inserimenti e cancellazioni di elementi in un insieme) viene diviso per il numero di operazioni, ottenendo un tempo medio $T(m)/m$ per operazione, detto appunto ammortizzato. È bene notare che in questo caso, a differenza della valutazione nel caso medio, non sono affatto coinvolte probabilità, poiché il tempo ammortizzato è una sorta di tempo medio per ciascuna operazione valutato nel caso pessimo su tutte le sequenze di m operazioni.

I.8 Ordine di grandezza e complessità

Nel valutare il tempo di calcolo $T(n)$ di una procedura, è in genere assai difficile quantificare con esattezza il numero di operazioni elementari eseguite. Questa difficoltà viene aggirata valutando il numero di operazioni in ordine di grandezza, cioè esprimendolo come limitazione della funzione $T(n)$ al tendere all'infinito della dimensione n, trascurando le costanti moltiplicative ed additive. Si parla così di complessità computazionale asintotica in ordine di grandezza (o, brevemente, complessità) di una procedura.

A tal fine, si usano le notazioni "O", "Ω" e "Θ" definite come segue:

$O(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano due costanti positive c ed m per cui $g(n) \leq cf(n)$ per ogni $n \geq m$.

$\Omega(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano due costanti positive c ed m per cui $cf(n) \leq g(n)$ per ogni $n \geq m$.

$\Theta(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano tre costanti positive c, d ed m per cui $cf(n) \leq g(n) \leq df(n)$ per ogni $n \geq m$.

Si dice che "una funzione $g(n)$ è di ordine omicron di $f(n)$" (in breve: "$g(n)$ è $O(f(n))$") per indicare una limitazione superiore al comportamento asintotico di

$g(n)$. Analogamente, si dice che "$g(n)$ è di ordine omega di $f(n)$" (in breve: "$g(n)$ è $\Omega(f(n))$") per indicarne una limitazione inferiore, e che "è di ordine theta di $f(n)$" (in breve: "$g(n)$ è $\Theta(f(n))$") per indicarne sia una limitazione inferiore che superiore[1].

Esempio I.8 (Ordini di grandezza). La funzione $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$, poiché esistono le costanti $c = 5$ ed $m = 4$ per cui $g(n) \leq 5n^2$ per $n \geq 4$. $g(n)$ è anche $\Omega(n^2)$, perché esistono le costanti $c = 1$ ed $m = 1$ tali che $g(n) \geq n^2$ per $n \geq 1$. Si noti che, in accordo alla definizione, $g(n)$ è $O(n^k)$ per ogni $k \geq 2$, ed anche $\Omega(n^k)$ per ogni $k \leq 2$. In genere però si richiede l'ordine più stretto che è $\Theta(n^2)$, poiché $g(n)$ è sia $O(n^2)$ che $\Omega(n^2)$. Similmente, è facile verificare che le funzioni $T(n)$ degli Esempi I.6 e I.7 sono $\Theta(n)$: le due versioni della procedura Pascal MIN hanno quindi complessità $\Theta(n)$. □

Per stimare gli ordini di grandezza, non è necessario ogni volta applicarne le definizioni e ricavare esplicitamente le costanti richieste. Si possono invece usare le seguenti regole, la cui dimostrazione è lasciata per esercizio, che permettono di semplificare la stima dell'ordine di grandezza di una funzione non negativa combinando insieme le stime delle sue parti.

1. *Riflessività*: Per ogni costante c (inclusa quindi $c = 1$) ed ogni funzione f, $cf(n)$ è $O(f(n))$ (lo stesso vale per Ω e per Θ);
2. *Transitività*: Se $g(n)$ è $O(f(n))$ ed $f(n)$ è $O(h(n))$, allora $g(n)$ è $O(h(n))$ (lo stesso vale per Ω e per Θ);
3. *Simmetria*: $g(n)$ è $\Theta(f(n))$ se e solo se $f(n)$ è $\Theta(g(n))$;
4. *Simmetria trasposta*: $g(n)$ è $O(f(n))$ se e solo se $f(n)$ è $\Omega(g(n))$;
5. *Somma*: La funzione $f(n) + g(n)$ è $O(\max\{f(n), g(n)\})$ (lo stesso vale per Ω e per Θ);
6. *Prodotto*: Se $g(n)$ è $O(f(n))$ ed $h(n)$ è $O(q(n))$, allora la funzione $g(n)h(n)$ è $O(f(n)q(n))$ (lo stesso vale per Ω e per Θ).

Esempio I.8 (continua). La funzione $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$, poiché $4n$ è $O(n)$ per la regola (1), $4n^2 = 4nn$ è $O(nn)$ per la regola (6), ed infine $4n^2 + 4n - 1$ è $O(\max\{n^2, n, 1\})$ applicando due volte la (5). □

Si noti che le proprietà (1), (2), e (3) implicano che Θ definisce una relazione di equivalenza sulle funzioni, tale che ogni insieme $\Theta(f(n))$ è una classe di equivalenza (detta classe di complessità).

Esempio I.9 (Classificazione ordini di grandezza). I seguenti ordini di grandezza sono via via crescenti: $\Theta(1)$ (ordine costante), $\Theta(\log n)$ (logaritmico), $\Theta(n)$ (lineare), $\Theta(n \log n)$ (pseudolineare), $\Theta(n^2)$ (quadratico), $\Theta(n^3)$ (cubico), $\Theta(2^n)$ (esponenziale in base 2), $\Theta(n!)$ (fattoriale), $\Theta(n^n)$ (esponenziale in base n). In generale, un ordine $\Theta(n^k)$, con k costante positiva, viene detto polinomiale, mentre $\Theta(a^n)$, con a costante maggiore di uno, è detto esponenziale. Un ordine esponenziale o maggiore è detto superpolinomiale. □

Le definizioni degli ordini di grandezza valgono per funzioni qualsiasi, ma nella valutazione della complessità di algoritmi sono applicate a funzioni come la $T(n)$

[1] Dato che $\Theta(f(n))$ è un insieme, alcuni autori, per indicare che $g(n)$ è $\Theta(f(n))$, utilizzano correttamente la notazione $g(n) \in \Theta(f(n))$, mentre altri usano impropriamente la notazione $g(n) = \Theta(f(n))$, intendendo esprimere con ciò che $\Theta(g(n)) = \Theta(f(n))$.

che contano il numero di operazioni nel caso pessimo (o medio) eseguite su tutti i possibili dati di ingresso di dimensione n. Poiché contano operazioni, tali funzioni $T(n)$ di complessità sono a valori non negativi (e quindi i loro ordini di grandezza verificano le proprietà (1)-(6) su menzionate).

Utilizzando gli ordini di grandezza, ogni operazione elementare costa $O(1)$ tempo. Le uniche istruzioni che non danno un contributo unitario sono ovviamente quelle condizionali, di iterazione (limitata o illimitata) e le chiamate di procedure e funzioni. Per esempio, la complessità della seguente istruzione condizionale è $O(max\{f(n), g(n), h(n)\})$ per la regola (5):

if condizione che richiede $O(f(n))$ tempo
  then qualcosa che richiede $O(g(n))$ tempo
  else qualcosa che richiede $O(h(n))$ tempo;

Invece, la complessità del seguente brano contenente tre costrutti iterativi, i primi due in sequenza, ma il terzo annidato nel secondo, è $O(max\{nf(n), nmg(m)\})$ per le regole (5) e (6):

for i := 1 **to** n **do**
  qualcosa che richiede $O(f(n))$ tempo;
for j := 1 **to** n **do**
  for k := 1 **to** m **do**
    qualcosa che richiede $O(g(m))$ tempo;

A questo punto, può sorgere una domanda legittima: poiché si trascurano le costanti, è sempre lecito preferire, tra due algoritmi, quello avente complessità di ordine più basso? Ad esempio, un algoritmo di complessità $\Theta(n^2)$ è da preferire ad un altro di complessità $\Theta(n^3)$, anche se la costante moltiplicativa nel primo è 200 e quella del secondo è 4? La risposta dipende dalla dimensione attesa dell'input. Per $n < 50$ il secondo algoritmo è più veloce del primo; pertanto se l'algoritmo deve essere eseguito sempre su dati di piccola dimensione si sceglierà quello con tempo $\Theta(n^3)$. Se invece sono previsti dati di dimensione grande, il rapporto tra i tempi di esecuzione, che è $4n^3/200n^2 = n/50$, cresce arbitrariamente con n e l'algoritmo di complessità $\Theta(n^2)$ risulta nettamente il migliore. Poiché in generale l'algoritmo deve poter essere eseguibile su dati di dimensione arbitraria, risulta sensato e conveniente scegliere sempre l'algoritmo la cui complessità è di ordine più basso possibile.

I.9 Algoritmi efficienti e inefficienti

Ricapitolando, per stabilire la complessità di un algoritmo, se ne studia l'ordine di grandezza O oppure Θ del tempo di calcolo $T(n)$, inteso come numero di operazioni elementari eseguite nel caso pessimo (o medio) in funzione della dimensione n dei dati di ingresso. In particolare, si dice che un algoritmo è efficiente se la sua complessità è di ordine polinomiale, mentre è inefficiente se la sua complessità è superpolinomiale. Gli algoritmi superpolinomiali sono in genere molto facili da trovare, poiché si basano sul semplice principio di "provare tutte le possibilità", ma hanno scarsa utilità pratica a causa dell'irragionevole quantità di tempo di calcolo richiesta.

Esempio I.10 (Ordinamento di un array). Si consideri il problema di ordinare un insieme di n interi, memorizzati in un array. Un banale algoritmo superpolinomiale è il seguente:

"Genera tutte le permutazioni degli n elementi dell'array e verifica, per ciascuna permutazione, se per ogni coppia di elementi adiacenti $A[i]$ e $A[i+1]$ risulti $A[i] < A[i+1]$".

Poiché verificare che un array sia ordinato richiede $\Theta(n)$ tempo (basta un ciclo **for** in Pascal) e il numero di permutazioni possibili è $n!$, la complessità dell'algoritmo è $\Theta(nn!)$. Un semplice algoritmo polinomiale invece è:

"Trova il minimo elemento tra gli n iniziali e scambialo con quello che occupa la prima posizione; ripeti il procedimento sui restanti $n-1$ elementi, poi sui rimanenti $n-2$, e così via fino ad aver esaurito gli elementi".

Tale algoritmo può essere descritto con la seguente procedura Pascal, la quale richiama al suo interno una nuova versione iterativa di MIN che restituisce nel parametro *pos* la posizione dell'elemento minimo nella porzione di array $A[j..n]$, per $j = 1, 2, \ldots, n$:

```
procedure ORDINA(var A: insieme; n: integer);
    var j, pos, temp: integer;
    begin
        for j := 1 to n do begin
            temp := A[j];
            A[j] := MIN(A, j, n, pos);
            A[pos] := temp
        end
    end;

function MIN(var B: insieme; j, k: integer; var pos: integer): integer;
    var m, i: integer;
    begin
        m := B[j]; pos := j
        for i := j + 1 to k do
            if B[i] < m then begin
                m := B[i];
                pos := i
            end;
        MIN := m
    end;
```

La procedura ORDINA esegue il ciclo **for** n volte. Alla generica iterazione j, viene chiamata la funzione MIN che a sua volta esegue un altro **for** $n-j$ volte. Il tempo complessivo di calcolo della ORDINA cresce, a meno di costanti, come

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

La complessità di ORDINA è quindi $\Theta(n^2)$. □

Come illustrato nella Figura I.1, la crescita vertiginosa delle funzioni superpolinomiali rende di fatto inutilizzabili algoritmi aventi tale ordine di complessità. Ad esempio, si assuma che un'istruzione elementare sia eseguita ogni microsecondo, come accade in un calcolatore elettronico attuale. Allora la procedura ORDINA richiede

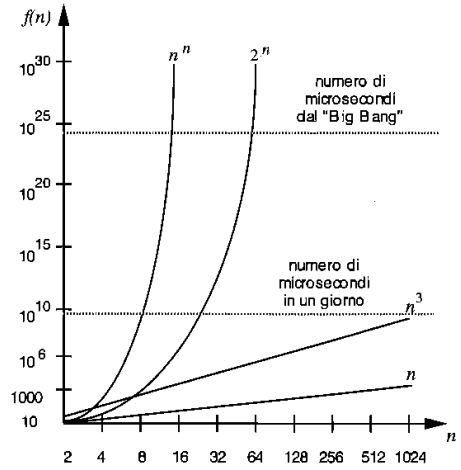


Figura I.1: Crescita di alcune funzioni (in scala logaritmica).

pressappoco un minuto per ordinare un insieme di 10.000 elementi, mentre l'algoritmo di ordinamento che generi tutte le permutazioni richiederebbe, per ordinare un insieme di soli 50 elementi, un numero di operazioni elementari avente oltre 66 cifre. Come termine di paragone si consideri che il numero di microsecondi dal "Big-Bang" ad ora ha "solo" 24 cifre (il Big-Bang sarebbe avvenuto circa 15 miliardi di anni fa). La conclusione è che gli algoritmi superpolinomiali non possono risolvere in tempo ragionevole che problemi di dimensioni estremamente piccole. Per esempio, un algoritmo di complessità $\Theta(2^n)$ richiede un numero di secoli con oltre 75 cifre per risolvere un problema di dimensione $n = 300$, quando uno $\Theta(n)$ impiega invece circa un decimo di secondo per $n = 100000$. Infine, la situazione non diverrebbe più rosea neanche se nei prossimi anni fossero realizzati calcolatori elettronici 100 o anche 1000 volte più veloci: per funzioni di complessità superpolinomiale la dimensione massima di un problema risolubile in un dato lasso di tempo non aumenterebbe che di pochissime unità. Per esempio, se D è la dimensione massima di un problema risolubile in un giorno con un algoritmo di complessità $\Theta(2^n)$, con un calcolatore 1000 volte più veloce la nuova dimensione massima D' non supera $D + 10$, perché da $2^{D'} = 1000 \cdot 2^D$ segue passando ai logaritmi che $D' = D + \log 1000$. In altri termini, moltiplicando la funzione di complessità per una costante, la dimensione massima risolvibile praticamente non cambia (questo spiega intuitivamente perché si possono trascurare le costanti nel valutare la complessità). La conclusione che se ne trae è univoca: algoritmi superpolinomiali sono e resteranno sempre estremamente inefficienti e non utilizzabili in pratica. Naturalmente, un algoritmo con complessità polinomiale di grado elevato, per esempio uno che richieda tempo $\Theta(n^{1000})$, pur essendo più veloce di uno con complessità $\Theta(2^n)$ per n che tende all'infinito, sarebbe in pratica inefficiente quanto uno superpolinomiale. Con l'eccezione di appositi casi costruiti artificiosamente, però, i problemi risolvibili con algoritmi di complessità

polinomiale richiedono un grado basso del polinomio (raramente più di 3) per cui è sensato considerare efficienti gli algoritmi polinomiali.

I.10 Complessità di problemi e algoritmi ottimi

Per risolvere un problema, si cerca di scoprire algoritmi di complessità sempre più bassa. Fino a quale punto questa ricerca può essere spinta? In altri termini, esiste una limitazione inferiore alla complessità che dipende solo dal problema in esame, che stabilisca una soglia invalicabile alla complessità di ogni algoritmo, anche non noto, che risolva il problema? La risposta è sì. Se si riesce a dimostrare che qualunque algoritmo per il problema in esame deve avere complessità $\Omega(f(n))$, allora si è stabilita una limitazione inferiore alla complessità del problema. Se invece si trova un particolare algoritmo di complessità $O(g(n))$, allora si è stabilita una limitazione superiore alla complessità del problema. Se $f(n) = g(n)$, allora l'algoritmo è detto ottimo, perché la sua complessità è, in ordine di grandezza, la migliore possibile. In generale, trovare limitazioni inferiori significative è molto più difficile che trovare algoritmi efficienti. Questo perché non si deve progettare in modo costruttivo un particolare algoritmo che risolva il problema, ma occorre invece fornire una prova matematica generale (cioè dimostrare un teorema) che valga per qualsiasi algoritmo che possa mai venire ideato anche in futuro. Purtroppo, sono noti pochissimi metodi generali di dimostrazione per limitazioni inferiori, per lo più poco potenti. Vediamone tre molto semplici (altri due più potenti, basati su alberi di decisione e riduzioni, saranno discussi in seguito):

1. *Dimensione dei dati*: se un problema ha in ingresso n dati e richiede di esaminarli tutti, allora una limitazione inferiore della complessità è $\Omega(n)$;
2. *Eventi contabili*: se un problema richiede che un certo evento sia ripetuto almeno n volte, allora una limitazione inferiore della complessità è $\Omega(n)$;
3. *Oracolo*: se un oracolo (o avversario), utilizzando una certa regola che l'algoritmo non conosce e che vale solo per certi dati d'ingresso, "divina" ad ogni opportunità la situazione più sfavorevole e fa lavorare l'algoritmo il più possibile, allora combattendo contro di esso si può individuare una limitazione inferiore della complessità.

Esempio I.11 (Dimensione dei dati). Il problema dell'ordinamento di un array, visto nell'Esempio I.10, richiede di esaminare tutti gli n elementi dell'array. Infatti, anche se l'array in ingresso fosse già ordinato, occorrerebbe comunque verificare l'ordinamento con un confronto tra ogni coppia di elementi adiacenti. Tale problema ha quindi complessità $\Omega(n)$. Questa limitazione non autorizza, per ora, ad affermare che la procedura ORDINA, che ha complessità $\Theta(n^2)$, non sia ottima. Infatti potrebbe essere dimostrabile una limitazione inferiore più elevata di $\Omega(n)$, così come potrebbe essere possibile trovare un algoritmo con complessità inferiore a $\Theta(n^2)$ (si dimostrerà in seguito per altra via una limitazione inferiore $\Omega(n \log n)$ e saranno individuati algoritmi ottimi di complessità $\Theta(n \log n)$). □

Esempio I.12 (Eventi contabili). Il problema della ricerca del minimo elemento in un insieme di n interi, visto nell'Esempio I.5, richiede almeno $n-1$ confronti. Infatti, il minimo può essere uno qualsiasi tra gli n elementi. Ciascuno dei rimanenti $n-1$ elementi deve essere scartato utilizzando almeno un confronto, poiché altrimenti non si potrebbe dire che tale elemento è il minimo. Una limitazione inferiore al numero di confronti necessari per trovare il minimo è quindi $\Omega(n)$. Le due versioni iterativa e ricorsiva di MIN, che usano entrambe $\Theta(n)$ confronti, sono dunque ottime. □

Esempio I.13 (Oracolo). Si consideri il problema di "fondere" due sequenze ordinate, ciascuna di n elementi interi, in un'unica sequenza ordinata di $2n$ elementi. Siano $X[1], X[2], \ldots, X[n]$ ed $Y[1], Y[2], \ldots, Y[n]$ le due sequenze, con $X[1] < X[2] < \cdots < X[n]$ ed $Y[1] < Y[2] < \cdots < Y[n]$, e siano i $2n$ elementi tutti distinti. Si consideri la seguente regola dell'oracolo:

$$X[i] < Y[j] \quad \text{se e solo se} \quad i < j.$$

Tale regola ovviamente non vale per tutti i dati di ingresso, ma solo per alcuni. Un qualsiasi algoritmo che risolva il problema per tutti i possibili dati di ingresso, deve ovviamente poterlo risolvere anche nel caso speciale nel quale valga la regola dell'oracolo. Se l'oracolo risolve il problema, la soluzione *deve* essere:

$$Y[1], X[1], Y[2], X[2], \ldots, Y[n], X[n].$$

Se un qualsiasi algoritmo, nel risolvere il problema, non eseguisse un confronto, p.e. quello tra $X[1]$ e $Y[2]$, allora potrebbe produrre la seguente sequenza, che sarebbe così indistinguibile da quella prodotta dall'oracolo:

$$Y[1], Y[2], X[1], X[2], Y[3], X[3], \ldots, Y[n], X[n].$$

Per produrre la sequenza dell'oracolo, l'algoritmo deve effettuare tutti i $2n-1$ confronti tra elementi adiacenti della sequenza stessa. Pertanto $\Omega(n)$ è una limitazione inferiore al numero di confronti necessari per risolvere il problema (si individui per esercizio una procedura Pascal ottima che usi $\Theta(n)$ confronti). □

Le precedenti tecniche di dimostrazione, ancorché apparentemente semplici, nascondono sempre insidie e sottigliezze. Per esempio, con la dimensione dei dati occorre fare molta attenzione, perché per molti problemi non è affatto necessario esaminare tutti i dati in ingresso. Inoltre, come sempre quando si dimostra un teorema, bisogna aver ben chiare quali siano le ipotesi sotto le quali si agisce. Per esempio, se un insieme è ordinato sequenzialmente, allora il minimo è il primo elemento che può essere reperito in tempo $O(1)$. Questo risultato non è in contrasto con la limitazione inferiore $\Omega(n)$, perché si assume un'ipotesi aggiuntiva (l'ordinamento dell'insieme, appunto).

I.11 Sommario

Lo scopo di questa "Introduzione" è stato quello di presentare alcune nozioni di base dell'informatica, quali algoritmi, procedure, tipi di dato, ricorsione, ordini di grandezza e complessità. Tali nozioni saranno oggetto di uno studio più approfondito nelle successive quattro parti in cui questo testo è suddiviso.

Nella prima saranno trattati metodi di specifica e realizzazione di strutture di dati, cioè particolari tipi di dato formati da aggregati di oggetti elementari (e non) che hanno rilevante interesse in informatica (ad esempio, sono strutture di dati gli array, le liste e gli insiemi). Si vedrà come la stessa struttura di dati possa essere rappresentata in vari modi che influenzano direttamente l'efficienza e la complessità delle operazioni ammesse sui componenti della struttura. Sarà anche illustrato come

l'esecuzione di procedure ricorsive possa essere gestita agevolmente utilizzando una particolare struttura, nota col nome di pila.

Nella seconda parte sarà dapprima data una classificazione dei problemi computazionali più comuni nei quali ci si imbatte nella pratica. Saranno successivamente trattate tecniche per il progetto di algoritmi efficienti per risolvere problemi appartenenti alle suddette classi, quali il divide et impera, il backtrack, la programmazione dinamica, il greedy e la ricerca locale. Sarà anche mostrato come la scelta di una struttura di dati, rispetto ad un'altra, influenzi direttamente l'efficienza di un algoritmo che la usi.

Nella terza parte si introdurranno classi di problemi "intrattabili" di rilevante interesse pratico. Tali problemi, detti NP-completi, sono tra loro computazionalmente correlati: per essi sono conosciuti soltanto algoritmi del tipo "prova tutte le possibilità" e si dubita fortemente che anche uno solo di essi sia risolubile con un algoritmo polinomiale! Si vedranno tecniche per trattare problemi di questa classe, quali la randomizzazione, l'approssimazione ed il branch-&-bound. Si accennerà infine anche all'esistenza di classi di problemi ancora più difficili, per i quali è possibile dimostrare limitazioni inferiori del tempo di calcolo che sono addirittura infinite!

Nella quarta parte si considereranno algoritmi paralleli. Al contrario degli algoritmi sequenziali, trattati nelle prime tre parti, dove è possibile eseguire una singola istruzione per volta, negli algoritmi paralleli è possibile eseguire più istruzioni contemporaneamente, permettendo così di abbassare drasticamente il tempo di risoluzione di molti problemi. Il progetto di algoritmi paralleli, purtroppo, dipende dal modello di calcolatore parallelo a disposizione e, in particolare, dal fatto che ci sia o no sincronismo tra le istruzioni eseguite contemporaneamente e che la memoria sia o no condivisa tra tutti gli esecutori. Verranno tra gli altri considerati algoritmi per modelli PRAM (sincroni con memoria condivisa), reti a grado limitato (sincroni senza memoria condivisa), concorrenti (asincroni con memoria condivisa), e distribuiti (asincroni senza memoria condivisa).

I.12 Esercizi

Esercizio I.1. Si ordinino le seguenti funzioni per ordine di grandezza crescente: $\log^2 n, 3^{n-2}, \pi^n, n^5 - 5n^2, n^4 - 7n^3, n^{\log n}, \log^n n, n/\log n, n^{1/2}, 19$.

$$19, \log^2 n, n^{1/2}, n/\log n, n^4 - 7n^3, n^5 - 5n^2, 3^{n-2}, \pi^n, n^{\log n}, \log^n n.$$

Esercizio I.2. Per dimostrare che una funzione $g(n)$ è $O(f(n))$ si può calcolare il limite per n che tende all'infinito del rapporto $g(n)/f(n)$. Se il limite esiste ed è una costante c (reale), allora $g(n)$ è $O(f(n))$, mentre se il limite è ∞ allora $g(n)$ cresce più velocemente di $f(n)$. Si dimostri che $n \log n$ è $O(n^a)$ per ogni $a > 1$.

Il rapporto $g(n)/f(n)$ è $(n \log n)/n^a = (\log n)/n^{a-1}$. Poiché per n che tende all'infinito sia il limite di $\log n$ che quello di n^{a-1} sono ∞, si applica la regola de L'Hôpital, derivando entrambe le funzioni, ottenendo rispettivamente n^{-1} e $(a-1)n^{a-2}$. Il rapporto tra n^{-1} e $(a-1)n^{a-2}$ è $(a-1)n^{a-1}$ il cui limite per n che tende all'infinito è 0. Pertanto si ha che $n \log n$ è $O(n^a)$ per ogni $a > 1$ (questo spiega perché l'ordine $O(n \log n)$ è detto pseudolineare).

Esercizio I.3. Si mostri che, per stabilire se una funzione $g(n)$ è $\Omega(f(n))$, si può calcolare il limite per n che tende all'infinito del rapporto $g(n)/f(n)$ ed affermare che se il limite esiste ed è una costante $c > 0$ oppure ∞, allora $g(n)$ è $\Omega(f(n))$. Come si può inoltre stabilire se $g(n)$ è $\Theta(f(n))$?

Esercizio I.4. Si dimostri che tutti i polinomi crescono meno di tutti gli esponenziali, cioè che n^k è $O(b^n)$, per qualsiasi $k > 0$ e $b > 1$.

Esercizio I.5. Si provi se le funzioni 2^{n+1}, 2^{2n} e 4^n sono $O(2^n)$.

Essendo $2^{n+1} = 2 \cdot 2^n$, si ha che 2^{n+1} è $O(2^n)$. Poiché invece $2^{2n} = 2^{n+n} = 2^n 2^n$, non è possibile che $2^{2n} < c2^n$ per ogni $n \geq m$, per nessuna costante c, e quindi 2^{2n} non è $O(2^n)$. Lo stesso vale per 4^n, perché $4^n = (2^2)^n = 2^{2n}$.

Esercizio I.6. Si dimostri che $\log(n!)$ è $\Theta(n \log n)$.

Dato che $(n/2)^{n/2} < n! < n^n$, si ha che $n/2 \log(n/2) < \log(n!) < n \log n$. Pertanto $\log(n!)$ è $\Theta(n \log n)$. Da questo segue anche che $\log(n!)$ è $O(n^a)$ per ogni $a > 1$ (si veda l'Esercizio I.2).

Esercizio I.7. Si dimostri, per induzione, che $\sum_{i=1}^{n} i^h$ è $O(n^{h+1})$.

Si proceda per induzione su h. La base dell'induzione è facilmente verificabile per $h = 0$, perché $\sum_{i=1}^{n} i^0 = \sum_{i=1}^{n} 1 = n$. Si assuma che la limitazione sia vera per un generico h, e si consideri la sommatoria per $h+1$:

$$\sum_{i=1}^{n} i^{h+1} = \sum_{i=1}^{n} i^h i \leq \sum_{i=1}^{n} i^h n = n \sum_{i=1}^{n} i^h$$

Poiché per ipotesi induttiva $\sum_{i=1}^{n} i^h$ è $O(n^{h+1})$, segue che $\sum_{i=1}^{n} i^{h+1}$ è $O(n^{h+2})$.

Esercizio I.8. Si considerino le funzioni $f(n) = n$ e $g(n) = n^{1+\operatorname{sen} n}$. Si dimostri che le due funzioni non sono confrontabili in ordine di grandezza, cioè che non vale né che $f(n)$ è $O(g(n))$, né che $f(n)$ è $\Omega(g(n))$.

Esercizio I.9. Dato un vettore di n interi, si vuole decidere se esiste un elemento che compare due volte. Determinare una limitazione inferiore e una superiore alla complessità di questo problema.

Esercizio I.10. Dato un vettore di n interi positivi, si vuole decidere se esistono due elementi che danno per somma 17. Determinare una limitazione inferiore $\Omega(n)$ e una superiore $O(n)$ alla complessità di questo problema.

Esercizio I.11. Dato un intero n, si vogliono stampare tutte le potenze di 5 minori o uguali di n. Si scriva una procedura Pascal di complessità ottima.

Una limitazione inferiore alla complessità del problema si determina con la tecnica degli eventi contabili, dove l'evento è la generazione di una nuova potenza di 5. Poiché bisogna elencare tutte le potenze di 5, la limitazione inferiore è data dal numero di tali potenze, che è $\Omega(\log_5 n)$.

La seguente procedura utilizza un ciclo **while** con una variabile *potenza* che, partendo da $5^0 = 1$, assume via via i valori delle potenze di 5, ciascuna ottenuta dalla precedente moltiplicando *potenza* per 5. Poiché il ciclo **while** è ripetuto $\Theta(\log_5 n)$ volte, la procedura ha complessità ottima.

```
procedure POTENZE5(n: integer);
    var potenza: integer;
    begin
        potenza := 1;
        while potenza ≤ n do begin
            write(potenza);
            potenza := potenza * 5
        end;
    end;
```

Esercizio I.12. Sia dato un vettore di n elementi che possono assumere solo i tre valori "verde", "bianco" e "rosso". Si vuole ordinare il vettore in $\Theta(n)$ tempo in modo che tutti i "verdi" precedano tutti i "bianchi" e tutti i "bianchi" precedano tutti i "rossi". Le uniche operazioni ammesse sono l'esame di un elemento e lo scambio di due elementi, dati i loro indici.

Una limitazione inferiore alla complessità del problema è $\Omega(n)$, poiché un qualsiasi algoritmo deve almeno esaminare tutti gli elementi. La seguente procedura ORDINA inizialmente scandisce il vettore $B[1..n]$ in avanti alla ricerca del primo elemento $B[k]$ non "verde", ed a ritroso alla ricerca dell'ultimo elemento $B[j]$ non "rosso". Successivamente, viene scandita in avanti (con l'indice i) la porzione $B[k..j]$ del vettore. Ogni elemento $B[i]$ "verde" incontrato è spostato indietro, scambiandolo con $B[k]$ e incrementando poi l'indice k, mentre ogni elemento $B[i]$ "rosso" è spostato in avanti scambiandolo con $B[j]$ e decrementando poi l'indice j. La scansione termina quando l'indice i scavalca j. La complessità della procedura, da richiamarsi con l'istruzione ORDINA(B, n), è ovviamente $\Theta(n)$.

```
type
    colore = (verde, bianco, rosso);
    bandiera = array[1..maxlung] of colore;

procedure ORDINA(var B: bandiera; n: integer);
    var i, j, k: integer; temp: colore;
    begin
        k := 1; j := n;
        while k ≤ n and B[k] = verde do k := k + 1;
        while j ≥ 0 and B[j] = rosso do j := j − 1;
        i := k;
        while i ≤ j do begin
            if B[i] = rosso then begin
                temp := B[i]; B[i] := B[j]; B[j] := temp;
                j := j − 1;
            end else
            if B[i] = verde then begin
                temp := B[k]; B[k] := B[i]; B[i] := temp;
                k := k + 1
            end;
            if B[i] = bianco then i := i + 1;
        end;
    end;
```

Parte prima

Strutture di dati

Capitolo 1

Tipi di dato e strutture di dati

I termini dato e tipo di dato, precedentemente introdotti, benché possano sembrare simili, hanno significato diverso:

1. in un linguaggio di programmazione, il dato è un valore che una variabile può assumere;
2. un tipo di dato è un modello matematico, consistente in una collezione di valori sui quali sono ammesse certe operazioni.

Esempio 1.1 (Interi). Il numero sette è un dato, che può essere assegnato ad una variabile di tipo intero. La collezione dei numeri interi, con le usuali operazioni aritmetiche ivi definite, è un tipo di dato. □

Ogni linguaggio di programmazione fornisce direttamente alcuni tipi di dato, detti tipi di dato primitivi. Ad esempio, in Pascal sono primitivi gli interi (con le operazioni: $+$, $-$, $*$, **div**, **mod**), i reali (con: $+$, $-$, $*$, $/$), i booleani (con: **and**, **or**, **not**) e i caratteri. Come si è visto precedentemente, può tuttavia risultare utile definire nuovi tipi di dato. Nel trattare tipi di dato, comunque, è bene distinguere tra i dati stessi e la loro rappresentazione. Ad esempio, un numero naturale è un'entità astratta, che può essere rappresentata in vari modi: in base 10, come nelle istruzioni Pascal, in base 2, come nei circuiti di una macchina elettronica, o addirittura con la notazione degli antichi romani, così che il numero "tredici" sarà rappresentato, rispettivamente, con "13", "1101", o "XIII". In ogni caso, le proprietà dei naturali sono specifiche di tali numeri e non dipendono da come i loro valori sono rappresentati.

In generale, le proprietà che caratterizzano un tipo di dato dipendono esclusivamente dalla specifica del tipo di dato stesso mentre devono essere indipendenti dal modo in cui i dati sono rappresentati. Si parla allora di un tipo di dato astratto, dove l'astrazione è proprio rispetto alla rappresentazione del tipo di dato. In altri termini, la specifica del tipo di dato fornisce le caratteristiche dei valori e delle operazioni per mezzo di definizioni e nozioni puramente matematiche. Una corretta specifica equivale ad un buon "manuale d'uso" chiaro, conciso e non ambiguo. Essa permette ad un progettista di realizzare il tipo di dato, cioè di scegliere come rappresentarne i valori e di scrivere il codice (p.e. in Pascal) di procedure e funzioni che corrispondono alle operazioni indicate nella specifica. Inoltre, permette anche ad un utente di scrivere procedure e funzioni che utilizzano il tipo di dato senza conoscerne la rappresentazione[1].

[1] Il Pascal, pur fornendo meccanismi di costruzione di nuovi tipi di dato, non permette di "nasconderne" la rappresentazione all'utente. Si assume tuttavia un utente "onesto", che si attiene alle specifiche astratte del tipo senza accederne direttamente alla rappresentazione.

1.1 Strutture di dati: specifica e realizzazione

Accade spesso che i dati da elaborare siano riuniti in agglomerati, detti strutture di dati (o strutture informative). Una struttura di dati è un particolare tipo di dato, caratterizzata più dall'organizzazione imposta agli elementi che la compongono, che dal tipo degli elementi stessi. In ultima analisi, il tipo degli elementi di una struttura può essere assunto addirittura parametrico.

Esempio 1.2 (Vettore). Un vettore è una sequenza di elementi dello stesso tipo. Sugli elementi si possono effettuare le operazioni di lettura e scrittura. La lettura consiste nel reperire il valore di un elemento, mentre la scrittura nel sostituire il valore di un elemento con un nuovo valore. In entrambe le operazioni si accede direttamente al generico elemento specificandone la posizione (indice) all'interno della sequenza. □

Esempio 1.3 (Record). Un record è un agglomerato di elementi di tipo diverso. Ogni elemento è individuato con un nome di riferimento e sono possibili le operazioni di lettura e scrittura. In entrambe le operazioni si accede direttamente al generico elemento specificandone il nome di riferimento (campo). □

Più precisamente, una struttura di dati consiste di:

a. un modo sistematico di organizzare i dati;
b. un insieme di operatori che permettono di manipolare elementi della struttura o di aggregare elementi per costruire altri agglomerati.

È comodo fornire una classificazione delle strutture di dati in base alle caratteristiche presentate dalla disposizione dei dati, dal loro numero e dal loro tipo. Di solito si usa distinguerle in:

1. *lineari*, in cui gli agglomerati sono formati da dati disposti in sequenza, tra i quali si individua un primo elemento, un secondo, eccetera;
2. *non lineari*, in cui non è individuata una sequenza;
3. *a dimensione fissa*, in cui il numero di elementi dell'agglomerato rimane sempre costante nel tempo;
4. *a dimensione variabile*, in cui il numero di elementi può aumentare o diminuire nel tempo;
5. *omogenee*, in cui i dati sono tutti dello stesso tipo;
6. *non omogenee*, in cui i dati non sono tutti dello stesso tipo.

Esempio 1.4. Il vettore è una struttura lineare omogenea a dimensione fissa. □

Nel descrivere strutture di dati, è opportuno distinguere, come per tutti i tipi di dato, tra la specifica astratta delle proprietà della struttura e i possibili modi con i quali si può memorizzare la struttura ed eseguire le operazioni.

In particolare, la specifica è distinta in due aspetti fondamentali: la notazione con cui sono indicati sia i tipi di dato che le operazioni (specifica sintattica), e il significato associato ad essi (specifica semantica). La specifica sintattica fornisce l'elenco dei nomi dei tipi di dato utilizzati per definire la struttura, delle operazioni specifiche della struttura stessa, e delle costanti. Inoltre descrive i domini di partenza e di arrivo, cioè i tipi degli operandi e del risultato, per ogni nome di operatore. La specifica semantica invece associa un insieme matematico ad ogni nome di tipo introdotto nella specifica sintattica, un valore ad ogni costante, e una funzione ad ogni nome di operatore. La funzione è definita matematicamente esplicitando una coppia di condizioni

(dette precondizione e postcondizione) sui domini di partenza e di arrivo. La precondizione stabilisce quando l'operatore è applicabile. Se la precondizione manca, l'operatore è sempre applicabile, mentre se è falsa l'operatore non è applicabile. La postcondizione indica come il risultato sia vincolato agli argomenti dell'operatore.

La realizzazione riconduce la specifica ai tipi di dato e agli operatori già disponibili. In particolare, gli operatori della struttura che modificano la struttura stessa, sono di solito simulati con procedure mentre quelli che restituiscono un dato primitivo sono simulati con funzioni. È bene sottolineare che alla stessa specifica possono corrispondere diverse realizzazioni, più o meno efficienti, a seconda dei dati e delle procedure utilizzati. Tali realizzazioni sono "nascoste" all'utente che è interessato soltanto a quali dati e operatori utilizzare, ma non si deve preoccupare di come questi sono realizzati. Al limite, le realizzazioni possono essere addirittura cambiate senza che l'utente neanche se ne accorga!

Esempio 1.5 (Vettore di n componenti). I nomi dei tipi usati nella specifica sintattica sono: vettore (che si va a definire), intero, e tipoelem. I nomi degli operatori e dei relativi domini di partenza e di arrivo sono:

CREAVETTORE: $() \rightarrow$ vettore
LEGGIVETTORE: (vettore, intero) $\rightarrow$ tipoelem
SCRIVIVETTORE: (vettore, intero, tipoelem) $\rightarrow$ vettore

Nella specifica semantica si individua col termine "vettore" l'insieme di tutte le sequenze di n elementi di tipo tipoelem (che è lasciato parametrico in modo da poter definire vettori con elementi di tipo qualsiasi, p.e. interi, reali, ecc.). Si indicano con v, i, ed e tre generici valori dei tipi vettore, intero e tipoelem, rispettivamente. Il significato associato alle operazioni, con relative pre e postcondizioni, è:

CREAVETTORE = v
Post: per ogni i, $1 \leq i \leq n$, l'i-esimo elemento $v(i)$ del vettore è uguale ad un prefissato elemento di tipo tipoelem

LEGGIVETTORE(v, i) = e
Pre: $1 \leq i \leq n$
Post: $e = v(i)$

SCRIVIVETTORE(v, i, e) = v'
Pre: $1 \leq i \leq n$
Post: $v'(i) = e$, $v'(j) = v(j)$ per ogni j tale che $1 \leq j \leq n$ e $j \neq i$

Si noti che la postcondizione di CREAVETTORE specifica come debba essere inizializzato il vettore. LEGGIVETTORE e SCRIVIVETTORE richiedono come precondizione che l'indice i abbia valore compreso tra 1 ed n, altrimenti l'operazione non può essere eseguita. LEGGIVETTORE restituisce il valore contenuto nell'i-esima posizione del vettore, mentre SCRIVIVETTORE restituisce un nuovo vettore v' con tutti i valori degli elementi uguali a quelli di v, tranne il valore dell'i-esima posizione che è posto al valore e.

In Pascal il vettore (array) è un tipo di dato primitivo, e quindi la realizzazione è immediata, a meno di una diversa notazione sintattica. La corrispondenza tra la specifica appena introdotta e quella del Pascal è la seguente:

CREAVETTORE $\Longleftrightarrow$ **var** v **array**$[1..n]$ **of** tipoelem
LEGGIVETTORE(v, i) $\Longleftrightarrow$ $v[i]$
SCRIVIVETTORE(v, i, e) $\Longleftrightarrow$ $v[i] := e$

dove i può essere anche una espressione di tipo intero e l'estremo inferiore dell'array non deve essere necessariamente uguale ad 1. È bene notare che in Pascal l'operazione di scrittura *non* restituisce un nuovo vettore v', bensì modifica il vecchio vettore v scrivendone il valore e nella posizione i-esima. □

Esempio 1.6 (Matrice). La matrice è una diretta estensione del vettore, in cui ci possono essere due o più indici. Una specifica per essa si può ricavare per esercizio estendendo quella vista per il vettore, dove saranno presenti gli operatori CREAMATRICE, LEGGIMATRICE e SCRIVIMATRICE, gli ultimi due con tanti indici quante sono le dimensioni della matrice.

In Pascal la matrice è ancora un array, e di fatto il vettore può essere visto come una particolare matrice monodimensionale. Ad esempio, per tre dimensioni si ha:

CREAMATRICE $\Longleftrightarrow$ **var** w **array**[1..n, 1..m, 1..k] **of** tipoelem
LEGGIMATRICE(w, i, j, h) $\Longleftrightarrow$ $w[i,j,k]$
SCRIVIMATRICE(w, i, j, h, e) $\Longleftrightarrow$ $w[i,j,k] := e$ □

Esempio 1.7 (Record a k campi). Il record è una k-pla sul dominio dato dal prodotto Cartesiano tipoelem1 × tipoelem2 × ··· × tipoelem$k$, con k costante. La specifica sintattica è simile a quella del vettore, con la differenza che sono previste k operazioni di lettura e k di scrittura, una per ogni campo:

CREARECORD: () → record
LEGGICAMPO1: (record) → tipoelem1
...
LEGGICAMPOk: (record) → tipoelem$k$
SCRIVICAMPO1: (record, tipoelem1) → record
...
SCRIVICAMPOk: (record, tipoelem$k$) → record

Indicando con $r = (e_1, \ldots, e_k)$ un dato di tipo record, la specifica semantica è:

CREARECORD = r
Post: $r = (e_1, \ldots, e_k)$, dove ciascun e_h, $1 \leq h \leq k$, è uguale ad un prefissato valore di tipo tipoelem$h$
LEGGICAMPO$h(r)$ = e
Post: $e = e_h$
SCRIVICAMPO$h(r, e)$ = r'
Post: $r' = (e_1, \ldots, e_{h-1}, e, e_{h+1}, \ldots, e_k)$

In Pascal il record, come il vettore, è un tipo primitivo. Al contrario del vettore, però, il campo h-esimo è individuato con un nome. Ad esempio, la definizione di un record "data" con $k = 3$ campi avviene con la dichiarazione:

type data = **record**
giorno: 1..31;
mese: 1..12;
anno: integer
end

L'accesso ad un generico campo avviene con l'operatore "." di selezione e quindi la corrispondenza tra alcune operazioni della specifica appena introdotta e quella del Pascal è banalmente:

CREARECORD $\Longleftrightarrow$ **var** r: data
LEGGICAMPO1(r) $\Longleftrightarrow$ $r.giorno$
LEGGICAMPO3(r) $\Longleftrightarrow$ $r.anno$
SCRIVICAMPO2(r, 6) $\Longleftrightarrow$ $r.mese := 6$ □

Vettori, matrici e record sono semplici strutture di dati che sono fornite dal Pascal come tipi di dato primitivi. Purtroppo il loro uso in Pascal è limitato da due vincoli.

Innanzitutto le dimensioni di array e record devono essere fissate staticamente, cioè non possono essere dichiarati array o record con un numero di elementi (o campi) variabile. Non è possibile, per esempio, leggere una variabile n dall'esterno e poi dichiarare un vettore di n elementi (come invece è ammesso in Algol), ma occorre prevedere una dimensione massima invalicabile. Si è quindi costretti a dichiarare una opportuna costante (diciamo *maxlung*) ed eventualmente ad utilizzare una porzione di n elementi del vettore, badando bene che n non superi mai la soglia *maxlung*. In altri termini, le strutture array e record in Pascal sono forzatamente statiche, con il duplice svantaggio di non poter mai avere $n > maxlung$ elementi e di sprecare in generale $maxlung - n$ spazio di memoria.

In secondo luogo, benché array e record possano essere usati come parametri formali di procedure e funzioni Pascal, essi non possono essere usati come risultato di funzioni Pascal. Ad esempio, un'operazione come SCRIVIVETTORE(v, i, e) va realizzata in Pascal con una procedura anziché una funzione, che quindi non restituisce un nuovo vettore v', come invece dovrebbe secondo la specifica, ma modifica il valore dello stesso vettore v (pertanto, nel resto di questo testo, si adotterà la convenzione di denotare con v', L', ecc. le stesse variabili v, L, ecc. di strutture di dati il cui valore è stato modificato per effetto di operazioni sulle strutture).

1.2 Rappresentazione in memoria

Nel valutare l'efficienza di procedure che utilizzino tipi di dato primitivi, si prescinde solitamente dalle caratteristiche della macchina, o meglio, si assume una organizzazione abbastanza generica, che è soddisfatta da ogni macchina ragionevole:

1. i dati sono contenuti nella memoria;
2. la memoria è divisa in celle (o parole) ognuna delle quali è di ugual lunghezza (p.e. 4 byte = 32 bit) e può contenere un dato elementare;
3. i dati elementari non eccedono la capacità di una cella (p.e. in Pascal gli interi non superano in modulo *maxint*);
4. si accede ad una cella di memoria specificandone un indirizzo;
5. gli indirizzi delle celle di memoria sono interi consecutivi;
6. ogni variabile utilizzata in una procedura (o funzione) è contenuta in una cella il cui indirizzo è conosciuto alla macchina e il cui accesso richiede tempo costante.

Così, per esempio, per eseguire un assegnamento Pascal "$m := B[i]$", si presuppone che i valori delle tre variabili m, i, e $B[i]$ siano contenuti in tre celle, alle quali la macchina deve accedere per eseguire l'istruzione. Vediamo brevemente come sono memorizzati vettori, matrici e record, e quale è l'effettivo tempo di accesso ad un generico elemento.

Esempio 1.8 (Memorizzazione di un vettore). Supponendo che tipoelem sia un dato elementare, il vettore è allocato in memoria in n celle consecutive, a partire da un indirizzo base b, come mostrato nella Fig. 1.1. L'indirizzo dell'i-esimo elemento $v[i]$ del vettore è $b+i-1$, $1 \leq i \leq n$, ed il tempo di accesso al generico elemento è $\Theta(1)$.

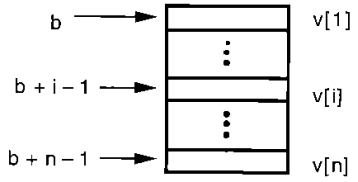


Figura 1.1: Memorizzazione di un vettore.

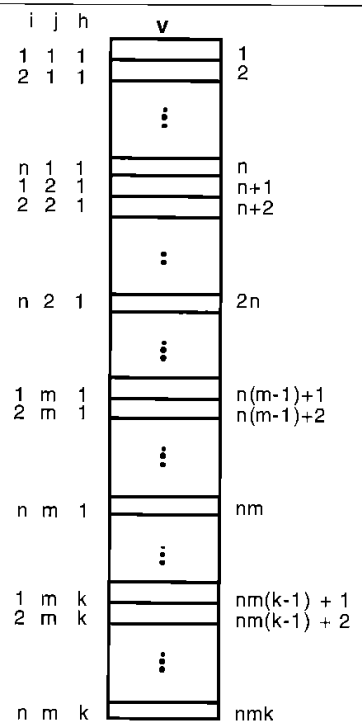


Figura 1.2: Memorizzazione di una matrice a 3 dimensioni.

Il passaggio di v per variabile in una procedura Pascal richiede tempo $\Theta(1)$, poiché viene di fatto passato solo b, ma quello per valore richiede tempo $\Theta(n)$, perché viene creato un nuovo vettore i cui elementi hanno lo stesso valore dei rispettivi elementi di v. □

Esempio 1.9 (Memorizzazione di una matrice). Nel memorizzare una matrice, si presenta subito un problema: la matrice non è una struttura lineare, mentre la memoria dell'elaboratore è lineare (infatti la memoria è in pratica un grande vettore). La matrice viene allora "simulata" con un vettore. Per esempio, una matrice w a tre dimensioni è memorizzata "per colonne", facendo variare l'indice i più in "fretta", poi facendo variare l'indice j e infine l'indice h. In questo modo si mettono in corrispondenza biunivoca gli elementi della matrice w con quelli di un vettore v, che ha mnk elementi ed è memorizzato a partire da un certo indirizzo base b. Si noti che ogni volta che l'indice j è incrementato di uno, sono stati esauriti n valori possibili dell'indice i, mentre ogni volta che l'indice h è incrementato di uno, sono stati esauriti n valori di i ed m di j, per un totale di nm valori. Pertanto, come è mostrato nella Fig. 1.2, il generico elemento $w[i, j, k]$ della matrice corrisponde all'elemento $v[p]$ del vettore, dove $p = i + (j-1)n + (h-1)nm$, il cui indirizzo di memoria è $b + p - 1$. Il tempo di accesso al generico elemento di w dipende essenzialmente dal tempo per valutare p. Poiché il numero di indici della matrice è costante, tale tempo è $\Theta(1)$. Il passaggio per valore costa ovviamente $\Theta(nmk)$ tempo, perché occorre copiare in una nuova matrice tutti gli elementi della matrice passata come parametro. □

Esempio 1.10 (Memorizzazione di un record). Se i k campi di un record hanno tutti la dimensione di una cella di memoria, allora la memorizzazione di un record è analoga a quella di un vettore. Il nome del campo j-esimo è infatti associato proprio all'intero j, e pertanto l'indirizzo del valore del campo j-esimo è proprio uguale a $b + j - 1$, come per il vettore, ed il tempo di accesso è $\Theta(1)$. In generale, però, i vari campi del record possono avere dimensione diversa. Sia n_j il numero (costante) di celle di memoria richieste dal j-esimo campo. Allora la lettura o la scrittura del campo j-esimo richiede l'accesso di n_j celle consecutive a partire dall'indirizzo:

$$b + \sum_{i=1}^{j-1} n_i$$

Poiché sia k che gli n_j sono costanti, le operazioni di lettura e scrittura richiedono tempo $\Theta(1)$. Leggermente più complicato è il caso in cui qualche campo del record sia strutturato, cioè sia a sua volta un altro record, o un vettore, o una matrice. In tal caso la memorizzazione procede in modo "indiretto". È usato un vettore v di k celle, tale che la j-esima cella di v, che si trova al solito indirizzo $b + j - 1$, contiene l'indirizzo di memoria b_j a partire dal quale è memorizzato il campo j-esimo del record. Tale indirizzamento indiretto può ovviamente essere generalizzato a più livelli, qualora per esempio il j-esimo campo del record sia un vettore i cui elementi sono a loro volta record, purché il numero complessivo di livelli sia costante. Il tempo richiesto per raggiungere la cella di memoria opportuna resta $\Theta(1)$, perché il numero di campi e di livelli è costante. Il passaggio per variabile di un record richiede quindi $\Theta(1)$ tempo. Il passaggio per valore però può essere molto costoso, qualora alcuni campi del record siano vettori o matrici. □

1.3 Puntatori

Le opportunità discusse precedentemente, quelle cioè di avere strutture di dati a dimensione variabile e di poter restituire strutture di dati come risultato di funzioni, possono essere ottenute con l'utilizzo di puntatori. I puntatori permettono un accesso indiretto ad un dato, mediante l'indirizzo della cella di memoria a partire dal quale il dato stesso è memorizzato. Per esempio, una variabile di tipo puntatore ad un

intero non ha per valore quello dell'intero a cui si riferisce, ma ha per valore proprio l'indirizzo della cella di memoria che contiene l'intero.

Esempio 1.11 (Puntatori). Si considerino le seguenti dichiarazioni di tipo in Pascal:

type punta-intero = ↑integer;
var *pippo*: integer;
pluto: punta-intero;

Mentre la variabile *pippo* indica un intero, la variabile *pluto* indica un puntatore ad un intero. Se durante l'esecuzione la variabile *pluto* contiene, mettiamo, il valore 666, questo significa che c'è un intero memorizzato nella cella di indirizzo 666. Per accedere a tale intero, si usa l'espressione *pluto*↑, così che se nella cella 666 è memorizzato 17, allora con *pluto*↑ si indica proprio 17. Effettuando l'assegnamento *pippo* := *pluto*↑, si assegna il valore 17 a *pippo*, mentre viceversa con l'assegnamento *pluto*↑ := *pippo* si assegna all'intero indirizzato da *pluto* il valore di *pippo* (se *pippo* valeva 13, allora nella cella di indirizzo 666 non ci sarà più 17 bensì 13). Se anche *paperino* è una variabile di tipo "punta-intero", con *paperino* := *pluto*, si assegna a *paperino* lo stesso valore di *pluto*, che è 666, così che entrambe le variabili *paperino* e *pluto* punteranno alla stessa cella. In tal caso, un eventuale confronto "**if** *paperino* = *pluto* **then** ... " darà esito positivo. □

All'atto di una dichiarazione di variabile (p.e. di tipo intero), viene immediatamente associata alla variabile stessa una cella di memoria alla quale può essere assegnato un valore del tipo puntato (nell'esempio, un intero). È bene notare invece che con una dichiarazione di tipo puntatore (p.e. ad un intero) è associata alla variabile una cella per contenere un indirizzo, ma non è associata alcuna cella per contenere un dato del tipo puntato. Pertanto, per poter utilizzare un intero puntato, occorre acquisire successivamente una cella di memoria per un dato di tipo intero e conservarne l'indirizzo nella variabile di tipo puntatore. È bene inoltre sottolineare che la stessa variabile di tipo puntatore potrà contenere via via gli indirizzi di molte celle differenti di tipo intero. Per questo motivo, i puntatori sono considerati variabili dinamiche, poiché le celle di memoria indirizzate possono variare durante l'esecuzione.

Oltre ad assegnamenti e confronti (per uguaglianza e non uguaglianza), le uniche altre operazioni ammesse sui puntatori sono la creazione e la distruzione. La creazione avviene con l'operazione (o per l'esattezza funzione di libreria) *new*, mentre la distruzione con *dispose*. Con la *new* è possibile acquisire in tempo costante una porzione di memoria della grandezza opportuna per contenere un dato del tipo puntato dal puntatore, mentre con la *dispose* tale porzione di memoria viene liberata in tempo costante.

Esempio 1.12 (continua). Con *new*(*pippo*) viene acquisita una cella di memoria per contenere un intero e l'indirizzo di questa cella è posto nella variabile *pippo*. Solo dopo avere acquisito in tal modo la cella, è possibile operare sull'intero puntato da *pippo*, per esempio con l'assegnamento *pippo*↑ := 17. Con *dispose*(*pippo*) viene liberata la cella puntata da *pippo*, e qualsiasi riferimento successivo a *pippo*↑ è illegale, finché non sarà assegnato a *pippo* un nuovo indirizzo. □

Il valore di un puntatore è un indirizzo assegnato per la prima volta dal sistema con l'esecuzione di una *new* che può essere poi trasferito con un'istruzione di assegnamento ad un altro puntatore. Benché alcuni linguaggi (per esempio il C) permettano di accedere al valore di un puntatore, in generale tali valori sono ignoti al programmatore. L'unica eccezione è rappresentata dallo speciale valore costante **nil**, che non corrisponde ad alcun indirizzo di memoria. **nil** può essere assegnato ad

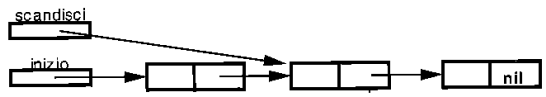


Figura 1.3: Concatenazione di record mediante puntatori.

un puntatore per indicare che non punta a niente ed essere usato in un'istruzione di confronto.

Grazie ai puntatori, è possibile per una funzione Pascal restituire come risultato, anziché direttamente un dato strutturato, l'indirizzo di un dato strutturato. Così si possono avere funzioni che restituiscono un puntatore ad un record oppure ad una matrice. Inoltre, riutilizzando ricorsivamente un puntatore ad un record come campo del record stesso, si possono concatenare più record tra loro tramite puntatori. Tali record possono essere in numero arbitrario e non necessariamente memorizzati in posizioni contigue della memoria.

Esempio 1.13 (Record concatenati). Si consideri la definizione:

type catena = ↑studente;
studente = **record**
matricola: integer;
successivo: catena
end;
var *inizio*, *scandisci*: catena;

Il tipo "catena" individua un puntatore ad un record di tipo "studente", che a sua volta è composto da un campo intero e da un puntatore ad un altro record dello stesso tipo. Si esegua adesso la seguente sequenza di istruzioni Pascal, il cui effetto è illustrato nella Fig. 1.3:

new(*inizio*);
new(*inizio*↑.*successivo*);
scandisci := *inizio*↑.*successivo*;
new(*scandisci*↑.*successivo*);
scandisci↑.*successivo*↑.*successivo* := **nil**;

Con *new*(*inizio*), il puntatore *inizio* conterrà l'indirizzo di un record di tipo "studente". Effettuando *new*(*inizio*↑.*successivo*), nel campo *successivo* del record puntato da *inizio*, ci sarà l'indirizzo ad un secondo record di tipo "studente". Dopo l'assegnamento *scandisci* := *inizio*↑.*successivo*, il puntatore *scandisci* punterà al secondo record e l'ulteriore *new*(*scandisci*↑.*successivo*) porrà nel campo *successivo* del secondo record l'indirizzo di un terzo record. L'assegnamento finale imposterà a **nil** il campo puntatore del terzo record. □

Concludiamo con la raccomandazione di usare i puntatori sempre con la massima cautela, perché con essi si elaborano indirizzi di memoria. Un loro uso scorretto tramite assegnamenti sbagliati o *dispose* azzardate può infatti causare l'impossibilità temporanea di indirizzare intere zone di memoria durante l'esecuzione di un programma.

1.4 Esercizi

Esercizio 1.1 (Specifica numeri complessi). Si fornisca una specifica del tipo di dato numero complesso, che includa i seguenti operatori: parte reale, parte immaginaria, argomento, modulo, costruzione a partire dalla rappresentazione cartesiana, costruzione a partire dalla rappresentazione trigonometrica, addizione, sottrazione, moltiplicazione, coniugato, divisione.

Esercizio 1.2 (Realizzazione numeri complessi). Si fornisca una realizzazione in Pascal del tipo di dato numero complesso specificato nell'Esercizio 1.1, utilizzando record e puntatori.

Esercizio 1.3 (Memorizzazione matrice). Una matrice w a tre dimensioni può essere memorizzata "per righe" in un vettore v, facendo variare più in fretta il terzo indice, indi il secondo, ed infine il primo. Si riscriva la formula che dà la corrispondenza tra $w[i, j, h]$ e $v[p]$.

Esercizio 1.4 (Memorizzazione matrice). Si riscriva la formula che dà la corrispondenza tra un generico elemento $w[i_1, i_2, \ldots, i_k]$ di una matrice a k dimensioni e $v[p]$, assumendo di memorizzare w nel vettore v "per colonne" e che ciascun indice i_j possa variare nell'intervallo min_j e max_j, per $1 \leq j \leq k$.

Esercizio 1.5 (Memoria associativa). Una memoria associativa (o mapping) è una corrispondenza $M : D \rightarrow C$ tra elementi del dominio D e del codominio C, in cui sono ammesse le operazioni:

CREA: $() \rightarrow$ mapping
ASSEGNA: (mapping, dominio, codominio) $\rightarrow$ mapping
CALCOLA: (mapping, dominio) $\rightarrow$ codominio $\times$ booleano

dove:

CREA(M) = Λ (Λ indica mapping vuoto con $M(d)$ *indefinito* per ogni d);
ASSEGNA(M, d, c) = M', con $M'(d) = c$ ed $M'(x) = M(x)$ per $x \neq d$;
CALCOLA(M, d) = (c, b), con b = *falso*, se $M(d)$ è indefinito,
$c = M(d)$ e b = *vero*, altrimenti.

Si fornisca una realizzazione per il tipo di dato mapping che fa uso di un vettore, e si discuta la complessità di ciascuna operazione, assumendo che il dominio D sia un insieme finito.

Capitolo 2

Liste

Una lista è una sequenza di elementi di un certo tipo, in cui è possibile aggiungere o togliere elementi. Per far questo, occorre specificare la posizione relativa all'interno della sequenza nella quale il nuovo elemento va aggiunto o dalla quale il vecchio elemento va tolto. A differenza del vettore, che è una struttura a dimensione fissa dove è possibile accedere direttamente ad ogni elemento specificandone l'indice, la lista è a dimensione variabile e si può accedere direttamente solo ad un ristretto sottoinsieme di elementi (di solito il primo o l'ultimo). Per accedere ad un generico elemento, occorre scandire sequenzialmente gli elementi della lista: partendo da un elemento accessibile direttamente, ci si sposta via via da un elemento ad uno adiacente nella sequenza, fino a raggiungere l'elemento desiderato.

Esempio 2.1 (Liste). Uno schedario contenente le informazioni degli impiegati di un'azienda, in cui è possibile ricercare, aggiungere e togliere schede, è un esempio di lista. □

Sia $L = a_1, a_2, \ldots, a_n$ una lista. La lunghezza di L è pari ad n, il numero dei suoi elementi. Se la lista è vuota, cioè non contiene alcun elemento, allora $n = 0$, e la lista è indicata con Λ. Ogni elemento è caratterizzato da una posizione in L, indicata con pos_i, e da un valore a_i. Anche se viene immediato immaginare che pos_i sia uguale ad i, si considera la posizione come un tipo di dato a sé stante la cui realizzazione, che varia a seconda della realizzazione adottata per il tipo di dato lista, può non essere uguale all'intero i. Per comodità, si assume inoltre l'esistenza di una posizione pos_0 che precede quella del primo elemento e di una posizione pos_{n+1} che segue quella dell'ultimo elemento.

2.1 Specifica

Un tipico insieme di operazioni per il tipo di dato lista comprende l'operazione di creazione CREALISTA, per inizializzare una lista ad un particolare valore (di solito la sequenza vuota), quelle di selezione PRIMOLISTA ed ULTIMOLISTA, per accedere direttamente al primo o all'ultimo elemento della sequenza, e SUCCLISTA e PREDLISTA, per passare da un elemento al successivo o al precedente nella sequenza, quelle di interrogazione LISTAVUOTA e FINELISTA, per verificare che la sequenza sia vuota o che sia stato oltrepassato un estremo della lista, quella di lettura LEGGILISTA per leggere il valore di un elemento, e quelle di modifica SCRIVILISTA per cambiare il valore di un elemento, INSLISTA per inserire un nuovo elemento nella lista, e CANCLISTA per cancellare un vecchio elemento della lista.

Formalmente, la sintassi degli operatori è la seguente.

CREALISTA: () → lista
LISTAVUOTA: (lista) → booleano
PRIMOLISTA: (lista) → posizione
ULTIMOLISTA: (lista) → posizione
SUCCLISTA: (posizione, lista) → posizione
PREDLISTA: (posizione, lista) → posizione
FINELISTA: (posizione, lista) → booleano
LEGGILISTA: (posizione, lista) → tipoelem
SCRIVILISTA: (tipoelem, posizione, lista) → lista
INSLISTA: (tipoelem, posizione, lista) → lista
CANCLISTA: (posizione, lista) → lista

Indicando col termine lista l'insieme di tutte le sequenze di lunghezza arbitraria di elementi di tipo "tipoelem" e con L una generica lista, la semantica degli operatori è data dalle seguenti funzioni, con relative precondizioni e postcondizioni.

CREALISTA() = L'
Post: $L' = \Lambda$, la sequenza vuota
LISTAVUOTA(L) = b
Post: b = vero, se $L = \Lambda$; b = falso, altrimenti
PRIMOLISTA(L) = p
Post: $p = pos_1$ (e quindi $pos_1 = pos_{n+1}$ se $L = \Lambda$)
ULTIMOLISTA(L) = p
Post: $p = pos_n$ (e quindi $pos_n = pos_0$ se $L = \Lambda$)
SUCCLISTA(p, L) = q
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $q = pos_{i+1}$
PREDLISTA(p, L) = q
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $q = pos_{i-1}$
FINELISTA(p, L) = b
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $0 \leq i \leq n+1$
Post: b = vero, se $p = pos_0$ oppure se $p = pos_{n+1}$; b = falso, altrimenti
LEGGILISTA(p, L) = a
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $a = a_i$
SCRIVILISTA(a, p, L) = L'
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $L' = a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n$
INSLISTA(a, p, L) = L'
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n+1$
Post: $L' = a_1, \ldots, a_{i-1}, a, a_i, a_{i+1}, \ldots, a_n$ se $1 \leq i \leq n$,
$L' = a_1, \ldots, a_n, a$, se $i = n+1$ (e quindi $L' = a$, se $i = 1$ e $L = \Lambda$)
CANCLISTA(p, L) = L'
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $L' = a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ (e quindi $L' = \Lambda$ se $i = n = 1$)

Dalla specifica emerge che per accedere ad un elemento occorre conoscerne la posizione. Ciò è possibile solo conoscendo a priori la posizione dell'elemento precedente (o seguente) ed applicando quindi l'operazione SUCCLISTA (o PREDLISTA). Le uniche operazioni che danno per risultato una posizione, senza che gliene venga fornita una in ingresso, sono le operazioni PRIMOLISTA e ULTIMOLISTA. Per accedere ad un generico elemento occorre pertanto scandire la lista in avanti a partire dal primo elemento, o a ritroso partendo dall'ultimo. Se viene oltrepassato un estremo della lista, SUCCLISTA e PREDLISTA restituiscono, rispettivamente, pos_{n+1} e pos_0. È possibile accorgersi di questo fatto con l'interrogazione FINELISTA, che in tal caso restituirà il valore booleano vero.

Poiché inserzioni e cancellazioni "allungano" ed "accorciano" la lista, esse provocano la modifica di tutte le posizioni degli elementi che seguono quello inserito o cancellato. In particolare, INSLISTA(a, p, L) con $p = pos_i$ inserisce il nuovo elemento di valore a come nuovo i-esimo elemento della sequenza, mentre il vecchio i-esimo elemento diventa l'$(i+1)$-esimo, il vecchio $(i+1)$-esimo diventa l'$(i+2)$-esimo, ecc. Se $p = pos_{n+1}$, INSLISTA inserisce il nuovo elemento in fondo alla sequenza, lasciando inalterate le posizioni dei vecchi elementi. Se la lista è vuota e $p = pos_1 = pos_{n+1}$, il nuovo elemento diviene il primo (ed unico) elemento della lista. Invece, CANCLISTA(p, L) con $p = pos_i$ cancella il vecchio i-esimo elemento della sequenza, mentre il vecchio $(i+1)$-esimo elemento diventa l'i-esimo, il vecchio $(i+2)$-esimo diventa l'$(i+1)$-esimo, ecc.

Si noti infine che l'operazione LISTAVUOTA è ridondante, poiché LISTAVUOTA(L) = FINELISTA(PRIMOLISTA(L), L) = FINELISTA(ULTIMOLISTA(L), L).

Esempio 2.2 (Il catalogo di Don Giovanni). Il famoso catalogo delle conquiste di Don Giovanni comprende italiane, tedesche, spagnole, francesi e turche. Tra queste ci sono contadine, cameriere, cittadine, duchesse, baronesse, marchesine e principesse. Il fedele servitore Leporello ha strutturato il catalogo del padrone con una lista ordinata per data di conquista, vorrebbe cancellare dal catalogo tutte le duchesse e copiare in un'altra lista L tutte le spagnole, mantenendone lo stesso ordine che avevano nel catalogo.

```
type tiponazione = (italiana, tedesca, spagnola, francese, turca);
    tiporango = (contadina, cameriera, citttadina, duchessa, baronessa,
        marchesa, principessa);
    conquista = record
        data: tipodata;
        nome: tiponome;
        rango: tiporango;
        nazionalità: tiponazione
    end;

procedure DONGIOVANNI (var catalogo, L: lista);
    var p, q: posizione; temp: conquista;
    begin
        CREALISTA(L);
        q := PRIMOLISTA(catalogo); p := PRIMOLISTA(L);
        while not FINELISTA(q, catalogo) do begin
            temp := LEGGILISTA(q, catalogo);
            if temp.nazionalità = spagnola then
                begin
                    INSLISTA(temp, p, L);
```

```
            p := SUCCLISTA(p, L)
        end;
    if temp.rango = duchessa
        then CANCLISTA(q, catalogo)
        else q := SUCCLISTA(q, catalogo)
    end
end
```

Nella procedura DONGIOVANNI sono utilizzate due posizioni p e q per scandire, rispettivamente, L e *catalogo*. L è inizializzata alla lista vuota, q alla prima posizione del catalogo, e p alla prima posizione di L. Poiché all'inizio L è vuota, $p = pos_1 = pos_{|L|+1}$, la posizione che segue l'ultimo elemento. All'interno del ciclo **while**, la posizione p viene mantenuta sempre uguale a quella che segue l'ultimo elemento di L, in modo da inserire un nuovo elemento in fondo alla sequenza. In questo modo, gli elementi inseriti in L mantengono lo stesso ordine che avevano nel catalogo. Nel ciclo **while**, *prima* è verificata la nazionalità e *poi* è controllato il rango. Così, se una conquista è al tempo stesso sia duchessa che spagnola, viene prima inserita nella lista L e poi cancellata dal catalogo. In accordo alla specifica del tipo di dato lista, la cancellazione di una duchessa dal catalogo non richiede di fare avanzare la posizione q, poiché q assume per definizione la posizione dell'elemento che seguiva quello cancellato. □

Per valutare la complessità di una procedura (come per esempio la DONGIOVANNI) che utilizzi le operazioni appena definite, occorre prima conoscere la complessità di ogni operazione impiegata. In altri termini, occorre conoscere la realizzazione della struttura di dati di tipo "lista".

La prima realizzazione che viene in mente è quella di memorizzare gli elementi della lista in un vettore. In tal caso, la posizione di un elemento è un indice del vettore e pos_i è proprio uguale ad i, per $1 \leq i \leq n$. In altri termini, alla contiguità tra elementi della lista corrisponde una contiguità di memorizzazione. Questa realizzazione permette agevolmente, cioè in tempo O(1), di passare da un elemento al successivo o al precedente, di accorgersi se si tenta di superare un estremo della lista, di leggere o modificare il valore di un elemento. Purtroppo presenta due seri svantaggi. Innanzitutto, il vettore è una struttura a dimensione fissa, mentre la lista è a dimensione variabile. Il numero di elementi della lista non può quindi superare la dimensione del vettore, che è fissata una volta per tutte. Inoltre, l'inserzione di un nuovo elemento o la cancellazione di un vecchio elemento richiedono un inaccettabile tempo $O(n)$, poiché è necessario scalare di una posizione a destra, in caso di inserzione, o a sinistra, in caso di cancellazione, tutti gli elementi che seguono quello inserito o cancellato. Pertanto, non è conveniente realizzare una lista con un vettore.

Vediamo due possibili realizzazioni diverse, che sono basate sull'uso di puntatori e di cursori, dove alla contiguità tra elementi della lista non corrisponde necessariamente una contiguità di memorizzazione. Queste realizzazioni permettono sia di ottenere complessità O(1) per tutte le operazioni, sia di non limitare il massimo numero di elementi presenti nella lista.

2.2 Realizzazione con puntatori

Come mostrato nella Fig. 2.1, ci sono vari modi possibili di realizzare le liste mediante puntatori. L'idea di base è di memorizzare una lista di n elementi in n record, usualmente detti celle, tali che l'i-esima cella contenga il valore dell'i-esimo

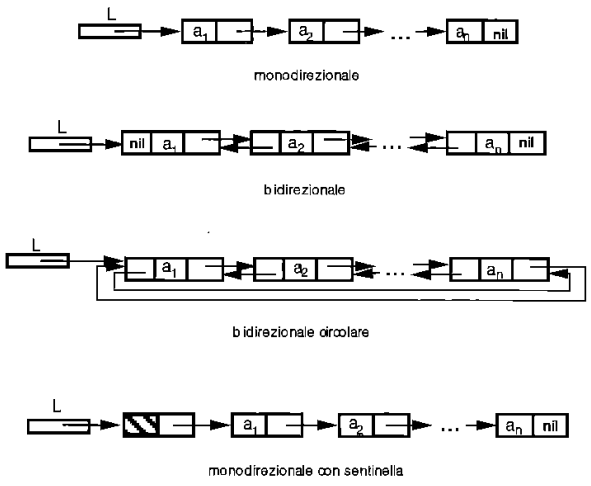


Figura 2.1: Alcune realizzazioni di una lista con puntatori.

elemento della lista e l'indirizzo (puntatore) della cella contenente l'elemento successivo (realizzazione monodirezionale) o gli indirizzi sia della cella successiva che precedente (realizzazione bidirezionale). La prima cella è indirizzata da una variabile L di tipo puntatore, mentre l'ultima cella contiene il valore convenzionale **nil** come indirizzo della cella successiva (ovviamente, nel caso bidirezionale, la prima cella contiene **nil** come indirizzo della cella precedente). Ricordiamo che gli indirizzi delle celle sono noti soltanto alla macchina, ma non lo sono al programmatore.

Entrambe le realizzazioni possono essere rese circolari, facendo sì che l'ultima cella contenga l'indirizzo della prima cella e, nel caso bidirezionale, che la prima cella contenga l'indirizzo dell'ultima. Per rendere più leggibile il codice Pascal delle operazioni INSLISTA e CANCLISTA quando operano sugli elementi estremi della lista, può essere conveniente introdurre una cella in più, detta sentinella, che è direttamente indirizzata da L, ma non contiene il valore di alcun elemento. Si hanno così otto realizzazioni possibili, a seconda che ci siano o no bidirezionalità, circolarità, e sentinella. La Fig. 2.1 illustra quattro di queste otto possibilità. Tutte le otto realizzazioni richiedono $\Theta(n)$ spazio di memoria.

Nel seguito, verrà utilizzata la realizzazione bidirezionale circolare con sentinella che permette di ottenere complessità O(1) per ciascuna operazione semplificandone nel contempo il codice Pascal, a scapito di un lieve spreco di memoria. Va però notato che è possibile ottenere complessità O(1) per quasi tutte le operazioni (con l'esclusione dei soli ULTIMOLISTA e PREDLISTA che hanno complessità $O(n)$) anche con una semplice realizzazione monodirezionale, senza circolarità né sentinelle, usando quindi la minore quantità di memoria possibile ma complicando il codice Pascal delle operazioni.

Con la realizzazione bidirezionale circolare con sentinella, la posizione pos_i è uguale all'indirizzo della cella contenente a_i, per $1 \leq i \leq n$, mentre pos_0 e pos_{n+1}

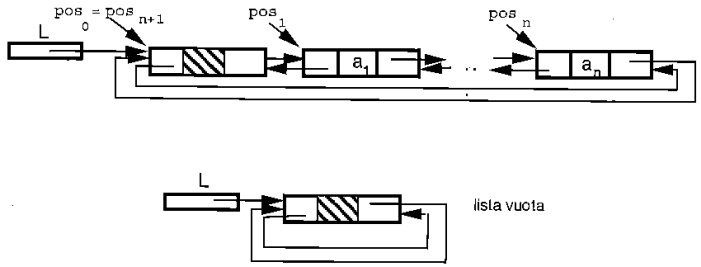


Figura 2.2: Realizzazione bidirezionale circolare con sentinella.

sono uguali all'indirizzo della sentinella (si veda la Fig. 2.2). Si noti che una lista di n elementi è formata da $n + 1$ celle (che contengono gli n elementi più la sentinella) e che quindi una lista vuota contiene esattamente una cella (la sentinella) che ha il proprio indirizzo sia come indirizzo della cella precedente che della cella successiva.

Utilizzando i puntatori del Pascal, si ottiene la seguente realizzazione, dove gli operatori che inizializzano e modificano la lista sono scritti tramite procedure, mentre quelli che restituiscono un valore sono scritti tramite function.

```
type
    lista = ↑cella;
    posizione = lista;
    cella = record
        precedente: posizione;
        elemento: tipoelem;
        successivo: posizione
    end;

procedure CREALISTA(var L: lista);
    begin
        new(L);
        L↑.successivo := L; L↑.precedente := L
    end;

function LISTAVUOTA(var L: lista): boolean;
    begin
        LISTAVUOTA := (L↑.successivo = L) and (L↑.precedente = L)
    end;

function PRIMOLISTA(var L: lista): posizione;
    begin
        PRIMOLISTA := L↑.successivo
    end;

function ULTIMOLISTA(var L: lista): posizione;
    begin
        ULTIMOLISTA := L↑.precedente
    end;
```



```
function SUCCLISTA(p: posizione; var L: lista): posizione;
    begin
        SUCCLISTA := p↑.successivo
    end;

function PREDLISTA(p: posizione; var L: lista): posizione;
    begin
        PREDLISTA := p↑.precedente
    end;

function FINELISTA(p: posizione; var L: lista): boolean;
    begin
        FINELISTA := (p = L);
    end;

function LEGGILISTA(p: posizione; var L: lista): tipoelem;
    begin
        LEGGILISTA := p↑.elemento
    end;

procedure SCRIVILISTA(a: tipoelem; p: posizione; var L: lista);
    begin
        p↑.elemento := a
    end;

procedure INSLISTA(a: tipoelem; var p: posizione; var L: lista);
    var temp: posizione;
    begin
        new(temp);
        temp↑.precedente := p↑.precedente;
        temp↑.successivo := p;
        p↑.precedente↑.successivo := temp;
        p↑.precedente := temp;
        temp↑.elemento := a;
        p := temp
    end;

procedure CANCLISTA(var p: posizione; var L: lista);
    var temp: posizione;
    begin
        temp := p;
        p↑.precedente↑.successivo := p↑.successivo;
        p↑.successivo↑.precedente := p↑.precedente;
        p := p↑.successivo;
        dispose(temp)
    end
```

Si noti che nelle procedure INSLISTA e CANCLISTA la posizione p è passata per variabile ed è modificata all'interno della procedura stessa. Infatti, in accordo alla specifica di INSLISTA, se $p = pos_i$, il nuovo elemento inserito diventa l'i-esimo, mentre quello che era il vecchio i-esimo diventa l'$(i + 1)$-esimo, ecc. Analogamente,

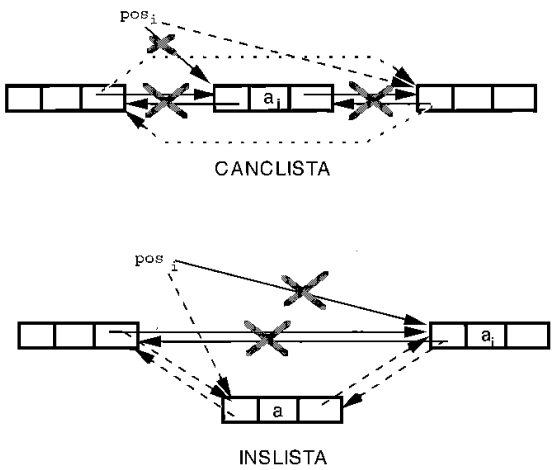


Figura 2.3: Aggiornamento dei puntatori nelle procedure CANCLISTA e INSLISTA

nel caso di CANCLISTA, il vecchio $(i+1)$-esimo elemento diventa l'i-esimo, ecc. In entrambi i casi, p deve contenere l'indirizzo della cella che contiene il nuovo i-esimo elemento (si veda la Fig. 2.3).

Si noti inoltre che utilizzando i puntatori è molto gravoso, se non addirittura impossibile, verificare che siano soddisfatte le condizioni sugli input di ogni operazione. Per esempio, la specifica di LEGGILISTA(p, L) prevede che p sia una posizione di un elemento della lista L; essendo p realizzato con un puntatore, la verifica di questa precondizione richiederebbe la scansione della lista! Per ragioni di efficienza, la condizione non viene verificata, ed il corretto uso delle operazioni è quindi piena responsabilità del programmatore.

2.3 Realizzazione con cursori

Nei linguaggi che non hanno i puntatori come tipi di dato primitivi (p.e. in Fortran o in Algol) è possibile simulare i puntatori con "cursori", cioè con variabili intere il cui valore è interpretato come un indice di un vettore. Questo vettore simula la memoria disponibile per i puntatori, che deve essere gestita esplicitamente nella realizzazione. Per far questo, si definisce un vettore SPAZIO che:

1. contiene tutte le liste, ognuna individuata da un proprio cursore iniziale;
2. contiene tutte le celle libere, organizzate anch'esse in una lista, detta "lista libera".

Esempio 2.3 (Lista libera). Siano $L = 8, 5$ ed $M = 2, 13, 7$ due liste di interi ed il vettore SPAZIO contenga in tutto 10 celle, ciascuna divisa nei campi "precedente", "elemento" e "successivo". Una possibile configurazione del vettore SPAZIO e delle tre liste L, M e *listalibera* è mostrato nella Fig. 2.4. L è memorizzata nelle celle 1, 7 e 4, M nelle celle 3, 5, 2 e 9, mentre *listalibera* comprende le celle 8, 0 e 6. □

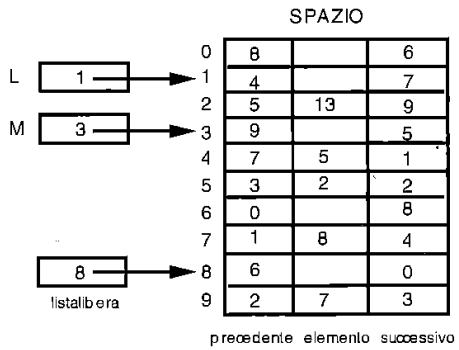


Figura 2.4: Memorizzazione delle liste L = 8, 5 ed M = 2, 13, 7 nel vettore SPAZIO.

Utilizzando i cursori, si può definire la posizione pos_i dell'i-esimo elemento di L uguale al valore del cursore alla cella del vettore SPAZIO che contiene a_i, se $1 \leq i \leq n$, o uguale al valore del cursore alla sentinella, se $i = 0$ oppure $i = n+1$. Assumendo che una opportuna procedura INIZIALIZZALISTALIBERA includa inizialmente tutte le celle del vettore SPAZIO nella lista libera, si ha:

```
const maxlung = ... {opportuna costante intera positiva};
type lista = 0..maxlung − 1;
   posizione = lista;
var listalibera: lista;
   SPAZIO: array[0..maxlung − 1] of record
      precedente: posizione;
      elemento: tipoelem;
      successivo: posizione
   end

procedure INIZIALIZZALISTALIBERA;
   var i: integer;
   begin
      listalibera := 0;
      for i := 0 to maxlung − 1 do begin
         SPAZIO[i].successivo := (i + 1) mod maxlung;
         SPAZIO[i].precedente := (i − 1) mod maxlung
      end
   end
```

La realizzazione delle operazioni usando cursori è analoga a quella mediante puntatori (per esempio, le espressioni $p\uparrow$.*successivo* e $p\uparrow$.*successivo*$\uparrow$.*precedente* diventano SPAZIO[p].*successivo* e SPAZIO[SPAZIO[p].*successivo*].*precedente*). Gli unici cambiamenti si hanno nelle procedure INSLISTA e CANCLISTA, dove occorre "spostare" una cella dalla *listalibera* alla lista L o da L alla *listalibera*.

Per realizzare queste operazioni, è comodo definire una procedura SPOSTA(h, k) che trasferisce la cella puntata da h, spostandola prima della cella puntata da k.

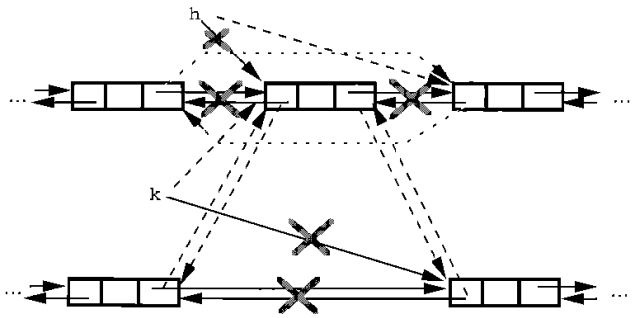


Figura 2.5: Aggiornamento dei cursori nella procedura SPOSTA.

La procedura aggiorna il cursore h in modo che punti alla cella che seguiva quella spostata e il cursore k in modo che punti alla cella spostata (si veda la Fig. 2.5).

Il codice di SPOSTA è analogo a quello di una cancellazione e di una inserzione simultanee (si confronti il codice di SPOSTA con quello delle procedure CANCLISTA e INSLISTA viste precedentemente, sostituendo i cursori ai puntatori).

```
procedure SPOSTA(var h, k: posizione);
    var temp: integer;
    begin
        temp := SPAZIO[h].successivo;
        SPAZIO[SPAZIO[h].precedente].successivo := SPAZIO[h].successivo;
        SPAZIO[SPAZIO[h].successivo].precedente := SPAZIO[h].precedente;
        SPAZIO[h].precedente := SPAZIO[k].precedente;
        SPAZIO[SPAZIO[k].precedente].successivo := h;
        SPAZIO[h].successivo := k; SPAZIO[k].precedente := h;
        k := h; h := temp
    end
```

Le procedure INSLISTA e CANCLISTA chiamano SPOSTA al loro interno e utilizzano *listalibera* come variabile globale.

```
procedure INSLISTA(a: tipoelem; var p: posizione; var L: lista);
    begin
        if LISTAVUOTA(listalibera) then
            {messaggio di errore: listalibera vuota}
        else begin
            SPOSTA(listalibera, p);
            SPAZIO[p].elemento := a
        end
    end;

procedure CANCLISTA(varp: posizione; var L: lista);
    begin
        SPOSTA(p, listalibera)
    end
```


In pratica, in un linguaggio di programmazione che fornisca i puntatori come tipi di dato primitivi, i puntatori sono realizzati proprio riservando un vettore SPAZIO per la porzione di memoria ad essi destinata, organizzato come una lista libera. L'operazione *new* non fa altro che staccare una cella dalla lista libera e restituire l'indirizzo (cursore) che la cella ha nel vettore SPAZIO, mentre l'operazione *dispose* riaggancia alla lista libera la cella non più utilizzata. Uno dei vantaggi dei linguaggi con puntatori consiste proprio nello sgravare il programmatore dalla gestione della memoria comune riservata alle liste, che è effettuata automaticamente dal sistema.

La complessità delle operazioni, usando la realizzazione con cursori, è ovviamente la stessa di quella con i puntatori, e l'occupazione di memoria è O(*maxlung*) per tutte le liste.

Si noti che la complessità della procedura DONGIOVANNI, vista nell'Esempio 2.2, è $\Theta(n)$ sia utilizzando i puntatori sia utilizzando i cursori.

2.4 Esercizi

Esercizio 2.1 (Rango). Il rango di un elemento di una lista è definito come la somma del suo valore e dei valori degli elementi che lo seguono. Scrivere una procedura Pascal ricorsiva di complessità ottima che, data una lista L, modifica L in modo che ogni elemento contenga il proprio rango (per esempio, se $L = 3, 2, 5$, allora si vuole ottenere $L = 10, 7, 5$).

Una limitazione inferiore alla complessità del problema è data dalla lunghezza n della lista L, ed è quindi $\Omega(n)$. La soluzione ricorsiva si basa sulla seguente proprietà:

$$\begin{aligned} rango(a_i) &= rango(a_{i+1}) + a_i && \text{se } i < n \\ rango(a_i) &= a_n && \text{se } i = n. \end{aligned}$$

```
procedure RANGO(p: posizione; var L: lista);
    var a: integer; q: posizione;
    begin
        q := SUCCLISTA(p, L);
        if not FINELISTA(q, L) then begin
            RANGO(q, L);
            a := LEGGILISTA(p, L) + LEGGILISTA(q, L);
            SCRIVILISTA(a, p, L)
        end
    end
```

La chiamata della procedura è:

```
if not LISTAVUOTA(L) then RANGO(PRIMOLISTA(L), L);
```

Il tempo $T(n)$ richiesto dalla procedura verifica le seguenti relazioni di ricorrenza, con c e d costanti,

$$\begin{aligned} T(n) &= c && \text{per } n = 1 \\ T(n) &= T(n-1) + d && \text{per } n > 1 \end{aligned}$$

la cui soluzione è $\Theta(n)$. Pertanto la procedura ha complessità ottima.

Esercizio 2.2 (I primi n interi). Si scriva una procedura Pascal ricorsiva di complessità ottima che, dato un intero $n \geq 1$, costruisca due liste L ed M tali che $L = 1, 2, \ldots, n-1, n$ ed $M = n, n-1, \ldots, 2, 1$.

Una limitazione inferiore alla complessità del problema è $\Omega(n)$ perché un qualsiasi algoritmo deve almeno generare gli n elementi della lista L e della lista M.

La procedura PRIMIENNE effettua un inserimento in testa alla lista L ed uno in fondo alla lista M ad ogni chiamata ricorsiva, in modo da memorizzare gli interi in senso crescente in L e decrescente in M.

Prima di effettuare la chiamata della procedura, occorre aver inizializzato le due liste L ed M:

```
CREALISTA(L);
CREALISTA(M);
PRIMIENNE(n, PRIMOLISTA(L), PRIMOLISTA(M), L, M);

procedure PRIMIENNE(n: integer; p, q: posizione; var L, M: lista);
    begin
        if n ≥ 1 then begin
            INSLISTA(n, p, L);
            INSLISTA(n, q, M);
            q := SUCCLISTA(q, M);
            PRIMIENNE(n − 1, p, q, L, M)
        end
    end
```

La complessità si ottiene risolvendo delle relazioni di ricorrenza simili a quelle dell'Esercizio 2.1. Pertanto la procedura ha complessità $O(n)$ e risulta ottima.

Esercizio 2.3 (Epurazione). Si scriva una procedura Pascal che, data una lista L di interi, restituisca un'altra lista M che contenga solo gli elementi di L i cui valori non compaiono esattamente due volte (p.e., se $L = 5, 7, 3, 2, 2, 1, 2, 3$, allora $M = 5, 7, 2, 2, 1, 2$).

Si effettua una doppia scansione della lista L per selezionare un elemento da L, verificare che non compaia esattamente 2 volte, e inserirlo in M. La posizione p è quella dell'elemento di L da verificare, la posizione q è utilizzata per scandire L alla ricerca delle occorrenze dell'elemento di posizione p, mentre *conta* serve a contare quante volte tale elemento si ripete. La posizione r è quella che segue l'ultimo elemento di M. La complessità è $O(n^2)$.

```
procedure EPURAZIONE(var L, M: lista);
    var p, q, r: posizione; conta: integer;
    begin
        CREALISTA(M); r := PRIMOLISTA(M); p := PRIMOLISTA(L);
        while not FINELISTA(p, L) do begin
            q := PRIMOLISTA(L); conta := 0;
            while not (FINELISTA(q, L)) and (conta ≤ 2) do begin
                if LEGGILISTA(q, L) = LEGGILISTA(p, L) then
                    conta := conta + 1;
            q := SUCCLISTA(q, L)
            end;
            if conta ≠ 2 then begin
                INSLISTA(LEGGILISTA(p, L), r, M);
```



```
                r := SUCCLISTA(r, M)
            end;
            p := SUCCLISTA(p, L)
        end
    end
```

Esercizio 2.4. Si supponga di rappresentare un numero decimale D mediante una lista L tale che il valore dell'elemento i-esimo della lista sia uguale a quello dell'i-esima cifra del numero. Si scriva una funzione Pascal che effettua la somma di due numeri. (Esempio: se $D = 190$ e $D' = 17$ allora $L = 1, 9, 0$, $L' = 1, 7$, e il risultato $D'' = D + D'$ è restituito nella lista $L'' = 2, 0, 7$).

Esercizio 2.5 (Tangentopoli). Dopo essere stati inquisiti per tangentopoli, n politici hanno deciso di suicidarsi disponendosi in cerchio e uccidendo via via una persona ogni k (... ma l'ultimo rimasto si suiciderà poi per davvero?). Si scriva una procedura Pascal che, data la lista L degli aspiranti suicidi, e dato k, $1 \leq k \leq n$, stampi l'ordine delle vittime (p.e. se L = Caio, Sempronio, Tizio, Tullio e $k = 3$, allora l'ordine delle vittime è: Tizio, Sempronio, Tullio, Caio). Si assuma di poter cancellare gli elementi da L.

```
procedure TANGENTOPOLI(var L: lista);
    var p: posizione; i: integer;
    begin
        p := PRIMOLISTA(L);
        while not LISTAVUOTA(L) do begin
            for i := 1 to k − 1 do begin
                p := SUCCLISTA(p, L);
                if FINELISTA(p, L) then p := PRIMOLISTA(L)
            end;
            write(LEGGILISTA(p, L));
            CANCLISTA(p, L)
        end
    end
```

Esercizio 2.6. Si scriva una procedura Pascal ricorsiva di complessità ottima che, data una lista L di interi, riordina L in modo che tutti gli elementi dispari precedano, nello stesso ordine che avevano inizialmente in L, tutti gli elementi pari (p.e., se $L = 3, 7, 8, 1, 4$, allora si ottiene $L = 3, 7, 1, 8, 4$).

Esercizio 2.7. Si scriva una procedura Pascal ricorsiva che data una lista L di interi, la modifichi eliminando ogni elemento pari e replicando ogni elemento dispari tante volte quanti sono gli elementi pari che lo precedono (p.e., $L = 4, 6, 7, 3, 2, 5$, allora si ha $L = 7, 7, 7, 3, 3, 3, 5, 5, 5, 5$).

Esercizio 2.8. Si scriva una procedura Pascal che, data una lista L di interi restituisca un'altra lista M che contiene solo gli elementi di L i cui valori sono uguali alla differenza tra i valori dell'elemento immediatamente precedente e quello immediatamente seguente (p.e., se $L = 5, 2, 3, 1, 2$, allora $M = 2, 1$). Si assuma per semplicità che L contenga almeno tre elementi.

Esercizio 2.9. Si scrivano le procedure Pascal per le operazioni delle liste usando la realizzazione con puntatori monodirezionale non circolare e senza sentinella. Tutte le operazioni (tranne ULTIMOLISTA e PREDLISTA) devono avere complessità $O(1)$. (Suggerimento: si definisca pos_i uguale all'indirizzo della cella contenente a_{i-1}, se $i > 0$, e pos_i = **nil**, se $i = 1$).

Capitolo 3

Pile e code

Una pila è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi soltanto ad un estremo (la "testa") della sequenza. Può essere quindi vista come un caso speciale di lista in cui l'ultimo elemento inserito è anche il primo ad essere rimosso e non è possibile accedere ad alcun elemento che non sia quello in testa. Tale meccanismo di accesso è detto LIFO (dall'inglese "Last In First Out", ovvero "ultimo arrivato, primo servito").

Esempio 3.1 (Pila, o Stack, o Pushdown List). Una catasta di pratiche burocratiche che si accumulano sulla scrivania di un funzionario ministeriale è spesso (ingiustamente) trattata col meccanismo LIFO, in cui l'ultima pratica inserita in testa alla catasta sarà esaurita per prima. Un tubetto di pastiglie ha un meccanismo di accesso analogo, poiché l'ultima pastiglia inserita nel tubetto dal fabbricante è anche la prima ad essere estratta dall'acquirente (si veda la Fig. 3.1). □

Analogamente, una coda è una sequenza di elementi di un certo tipo, in cui è possibile aggiungere elementi ad un estremo (il "fondo") e togliere elementi dall'altro estremo (la "testa"). Può essere considerata come una particolare lista in cui il primo elemento inserito è anche il primo ad essere rimosso e non è possibile accedere ad alcun elemento che non sia quello in testa o quello in fondo. Tale meccanismo di accesso è detto FIFO (dall'inglese "First In First Out", ovvero "primo arrivato, primo servito").

Esempio 3.2 (Coda, o Fila d'attesa). In Inghilterra, una fila di persone ad uno sportello postale è gestita con la disciplina FIFO (si veda la Fig. 3.2). □

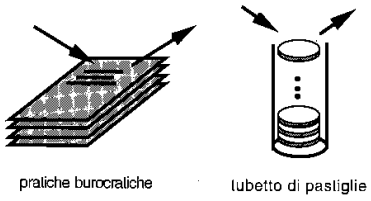


Figura 3.1: Meccanismo di accesso LIFO.

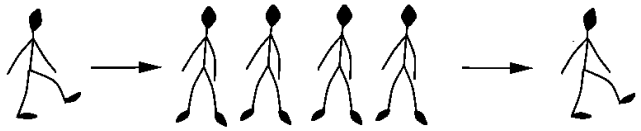


Figura 3.2: Meccanismo di accesso FIFO.

3.1 Specifiche

Le operazioni per il tipo di dato pila comprendono l'operazione di creazione CREAPILA, per inizializzare la pila alla sequenza vuota, quella di interrogazione PILAVUOTA, per verificare che la pila sia vuota oppure no, quella di lettura LEGGIPILA, per leggere il valore dell'elemento in testa alla pila, e quelle di modifica FUORIPILA, per eliminare l'elemento che si trova in testa alla pila, e INPILA, per aggiungere un elemento in testa alla pila. La specifica sintattica del tipo di dato è la seguente.

CREAPILA:	$() \rightarrow$ pila
PILAVUOTA:	(pila) $\rightarrow$ booleano
LEGGIPILA:	(pila) $\rightarrow$ tipoelem
FUORIPILA:	(pila) $\rightarrow$ pila
INPILA:	(tipoelem, pila) $\rightarrow$ pila

Indicando col nome "pila" l'insieme delle sequenze di elementi di tipo "tipoelem" e con P una generica sequenza $a_1, a_2, \ldots, a_n$, la semantica della pila può essere specificata formalmente come segue.

CREAPILA() = P'
Post: $P' = \Lambda$, la sequenza vuota

PILAVUOTA(P) = b
Post: b = vero, se $P = \Lambda$; b = falso, altrimenti

LEGGIPILA(P) = a
Pre: $P = a_1, a_2, \ldots, a_n$ e $n \geq 1$
Post: $a = a_1$

FUORIPILA(P) = P'
Pre: $P = a_1, a_2, \ldots, a_n$ e $n \geq 1$
Post: $P' = a_2, \ldots, a_n$, se $n > 1$, $P' = \Lambda$, se $n = 1$

INPILA(a, P) = P'
Pre: $P = a_1, a_2, \ldots, a_n$ e $n \geq 0$
Post: $P' = a, a_1, a_2, \ldots, a_n$, (e quindi $P' = a$, se $n = 0$)

Le operazioni per il tipo di dato coda comprendono CREACODA, CODAVUOTA, LEGGICODA e FUORICODA, con la stessa specifica delle rispettive operazioni della pila, e INCODA, per aggiungere un nuovo elemento in fondo alla coda. Indicata con Q una generica coda, la specifica semantica di INCODA è ovviamente diversa da quella di INPILA.

INCODA(a, Q) = Q'
Pre: $Q = a_1, a_2, \ldots, a_n$, e $n \geq 0$
Post: $Q' = a_1, a_2, \ldots, a_n, a$, (e quindi $Q' = a$, se $n = 0$)

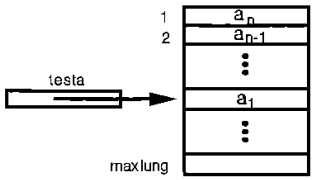


Figura 3.3: Realizzazione di una pila con un vettore.

Poiché pile e code sono casi particolari di liste, ogni realizzazione descritta per le liste funziona anche per le pile e per le code. Per esempio, le operazioni per le pile possono essere simulate con quelle delle liste nel modo seguente:

CREAPILA = CREALISTA
PILAVUOTA(P) = LISTAVUOTA(P)
LEGGIPILA(P) = LEGGILISTA(PRIMOLISTA(P), P)
FUORIPILA(P) = CANCLISTA(PRIMOLISTA(P), P)
INPILA(a, P) = INSLISTA(a, PRIMOLISTA(P), P).

Usando le realizzazioni delle liste con puntatori viste nel capitolo precedente, la complessità di ciascuna operazione per le pile è O(1).

Per simulare INCODA con operazioni delle liste, bisogna distinguere il caso di inserzione in una coda vuota da quello di inserzione in una coda non vuota. La simulazione nel primo caso è identica a quella di INPILA, mentre nel secondo caso occorre inserire l'elemento nella posizione che segue quella dell'ultimo elemento della lista:

INCODA(a, Q) = INSLISTA(a, SUCCLISTA(ULTIMOLISTA(Q), Q), Q).

La complessità di INCODA è O(1) se viene usata una realizzazione bidirezionale circolare (con o senza sentinella) della lista, mentre diviene $\Theta(n)$ usando le altre realizzazioni. Oltre alle realizzazioni per pile e code che si basano su quelle descritte per le liste, ci sono due realizzazioni specifiche, che utilizzano vettori, in cui tutte le operazioni delle pile e delle code continuano ad avere complessità O(1). Tali realizzazioni sono utilizzabili quando si è in grado di limitare superiormente con un opportuno valore il numero massimo di elementi che possono essere presenti nella pila o nella coda.

3.2 Realizzazione di una pila con vettore

Come mostrato nella Fig. 3.3, una realizzazione con vettore del tipo di dato pila consiste nel memorizzare gli n elementi della pila, in ordine inverso, nelle prime n posizioni di un vettore di *maxlung* elementi, mantenendo un cursore alla testa della pila. La pila vuota viene individuata dal valore 0 contenuto nel cursore, mentre la pila satura, cioè che contiene il massimo numero consentito di elementi, è individuata dal valore *maxlung*. FUORIPILA è realizzata decrementando il valore del cursore, mentre INPILA è realizzata incrementando il valore del cursore ed inserendo il nuovo elemento nella posizione del vettore individuata dal cursore. Eventuali tentativi di

inserzione in una pila satura o di estrazione da una pila vuota possono sorgere in caso di errori di programmazione e non sono ammessi. In particolare, il tentativo di inserzione in una pila satura è segnalato con un opportuno messaggio di errore.

La realizzazione dettagliata con procedure Pascal è la seguente:

```
type
    pila = record
        testa: 0..maxlung;
        elementi: array[1..maxlung] of tipoelem;
    end;

procedure CREAPILA(var P: pila);
    begin
        P.testa := 0
    end;

function PILAVUOTA(var P: pila): boolean;
    begin
        PILAVUOTA := (P.testa = 0)
    end;

function LEGGIPILA(var P: pila): tipoelem;
    begin
        if not PILAVUOTA(P) then
            LEGGIPILA := P.elementi[P.testa]
    end;

procedure FUORIPILA(var P: pila);
    begin
        if not PILAVUOTA(P) then
            P.testa := P.testa - 1
    end;

procedure INPILA(a: tipoelem; var P: pila);
    begin
        if P.testa = maxlung then
            {messaggio di errore: pila satura}
        else begin
            P.testa := P.testa + 1;
            P.elementi[P.testa] := a
        end
    end;
```

Con questa realizzazione, ogni operazione della pila richiede tempo costante per essere eseguita. Per ottenere la stessa complessità anche per tutte le operazioni della coda, si utilizza un vettore "circolare".

3.3 Realizzazione di una coda con vettore circolare

Si supponga di avere un array di *maxlung* elementi, indicati da 0 a *maxlung* − 1, in cui l'elemento di indice 0 è considerato come successore di quello di indice

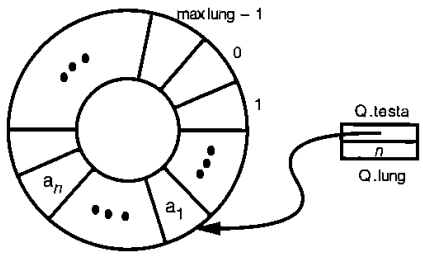


Figura 3.4: Realizzazione di una coda con un vettore circolare.

maxlung – 1. La coda è contenuta in n posizioni consecutive del vettore, per esempio secondo il verso orario, e i suoi elementi estremi sono individuati da due cursori. Per cancellare o per inserire un elemento, viene incrementato di uno (modulo *maxlung*) il rispettivo cursore di testa o di fondo. In questo modo la coda "migra" lungo il vettore circolare e, ad ogni istante, si trova da qualche parte nel vettore, sempre in posizioni consecutive secondo il verso orario.

Anziché utilizzare due cursori, per individuare la coda è sufficiente conoscere l'indice del primo elemento e la lunghezza della coda stessa, come mostrato nella Fig. 3.4. Infatti, se i è la posizione dell'elemento in testa alla coda, allora quella dell'ultimo elemento della coda è $(i + n - 1)$ modulo *maxlung*.

```
type
    coda = record
        elementi: array[0..maxlung − 1] of tipoelem;
        testa: 0..maxlung − 1;
        lung: 0..maxlung;
    end;

procedure CREACODA(var Q: coda);
    begin
        Q.testa := 0;
        Q.lung := 0
    end;

function CODAVUOTA(var Q: coda): boolean;
    begin
        CODAVUOTA := (Q.lung = 0)
    end;

function LEGGICODA(var Q: coda): tipoelem;
    begin
        if not CODAVUOTA(Q) then
            LEGGICODA := Q.elementi[Q.testa]
    end;
```



```
procedure FUORICODA(var Q: coda);
    begin
        if not CODAVUOTA(Q) then begin
            Q.testa := (Q.testa + 1) mod maxlung;
            Q.lung := Q.lung − 1
        end
    end;

procedure INCODA(a: tipoelem; var Q: coda);
    begin
        if Q.lung = maxlung then
            {messaggio di errore: coda satura}
        else begin
            Q.elementi[(Q.testa + Q.lung) mod maxlung] := a;
            Q.lung := Q.lung + 1
        end
    end;
```

La complessità di ciascuna operazione della coda è O(1).

3.4 Pile e procedure ricorsive

Una importante applicazione delle pile consiste nella gestione dell'esecuzione di procedure ricorsive. Infatti, l'esecuzione di una procedura ricorsiva prevede il "salvataggio" dei dati sui quali lavora la procedura al momento di una nuova chiamata ricorsiva interna. Tali dati sono successivamente "riesumati" quando la computazione interna termina e viene ripresa la computazione che era stata sospesa. Questo meccanismo è ovviamente di tipo LIFO, poiché la computazione "più interna" di una procedura ricorsiva, cioè quella iniziata per ultima, è sempre la prima a terminare. Per chiarire come procede il calcolo in una procedura ricorsiva, si consideri un problema "leggendario", divenuto ormai un classico della programmazione ricorsiva:

TORRI DI HANOI: "In un monastero tibetano ci sono n dischi d'oro, forati al centro, tutti di diametro diverso. I dischi sono infilati in un piolo verticale, accatastati per diametro decrescente a partire dal basso. I monaci devono spostare tutti i dischi dal piolo in cui si trovano in un altro piolo, formando una catasta identica a quella di partenza. È possibile usare un terzo piolo di appoggio per effettuare i trasferimenti. Si può sfilare da un piolo soltanto un disco per volta, e il disco sfilato deve essere subito infilato in un altro piolo evitando di sovrapporlo ad un disco di diametro più piccolo."[1]

Questo rompicapo si presta ad una semplice risoluzione ricorsiva. Indichiamo i tre pioli coi nomi *origine*, *destinazione* e *intermedio*. Se c'è un solo disco, la soluzione è banale: si trasferisce direttamente il disco dal piolo *origine* al piolo *destinazione*. Se ci sono n dischi, con $n > 1$, basta spostare i primi $n - 1$ dischi della catasta dal piolo *origine* al piolo *intermedio*, indi trasferire direttamente il disco rimasto dal piolo *origine* al piolo *destinazione*, e infine spostare gli $n - 1$ dischi dal piolo *intermedio* al piolo *destinazione*. Supponendo che *origine*, *intermedio* e *destinazione* siano tre interi distinti (p.e. 1, 2 e 3) si ottiene la seguente procedura Pascal:

[1] Narra la leggenda che il mondo finirà prima che i monaci abbiano spostato tutti i dischi!

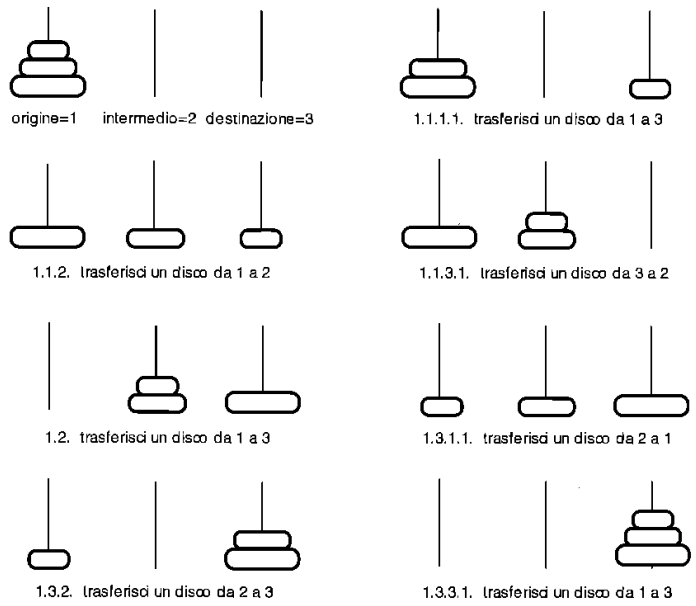


Figura 3.5: Sequenza di trasferimenti di dischi eseguiti dalla procedura ricorsiva TORRIDIHANOI per $n = 3$.

```
procedure TORRIDIHANOI(n, origine, destinazione, intermedio: integer);
    begin
        if n = 1 then
            {'trasferisci un disco da' origine 'a' destinazione}
        else begin
            TORRIDIHANOI(n − 1, origine, intermedio, destinazione);
            {'trasferisci un disco da' origine 'a' destinazione}
            TORRIDIHANOI(n − 1, intermedio, destinazione, origine)
        end
    end;
```

Benché sia facile dimostrare la correttezza di questa procedura usando il principio d'induzione, è molto difficile capire quale sia l'effettiva sequenza di trasferimenti eseguiti dalla procedura stessa.

Esempio 3.3 (Torri di Hanoi). Si assuma $n = 3$, *origine* = 1, *intermedio* = 2 e *destinazione* = 3. L'esecuzione della procedura TORRIDIHANOI con questi dati provoca la seguente sequenza di chiamate ricorsive e di trasferimenti di dischi (si veda la Fig. 3.5):

1. TORRIDIHANOI(3,1,3,2)
    1.1. TORRIDIHANOI(2,1,2,3)
        1.1.1. TORRIDIHANOI(1,1,3,2)

        1.1.1.1. trasferisci un disco da 1 a 3
    1.1.2. trasferisci un disco da 1 a 2
    1.1.3. TORRIDIHANOI(1,3,2,1)
        1.1.3.1. trasferisci un disco da 3 a 2
1.2. trasferisci un disco da 1 a 3
1.3. TORRIDIHANOI(2,2,3,1)
    1.3.1. TORRIDIHANOI(1,2,1,3)
        1.3.1.1. trasferisci un disco da 2 a 1
    1.3.2. trasferisci un disco da 2 a 3
    1.3.3. TORRIDIHANOI(1,1,3,2)
        1.3.3.1. trasferisci un disco da 1 a 3 □

La complessità della procedura TORRIDIHANOI è esponenziale. Infatti, il numero $T(n)$ di trasferimenti di dischi è dato da:

$$\begin{aligned} T(n) &= 1, && \text{se } n = 1, \\ T(n) &= 2T(n-1) + 1, && \text{se } n \geq 2. \end{aligned}$$

Sostituendo successivamente si ottiene:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 = \\ &= 2(2T(n-2) + 1) + 1 = 4T(n-2) + 3 = \\ &\vdots \\ &= 2^{n-1}T(1) + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

Assumendo un trasferimento ogni microsecondo, per $n = 100$ occorrono ben 400.000 miliardi di secoli: nessun dubbio che il mondo finirà prima!

Vediamo come sia possibile eliminare manualmente la ricorsione dalla procedura, trasformandola in una iterativa equivalente. Innanzitutto, la seconda chiamata ricorsiva nella procedura TORRIDIHANOI può essere facilmente eliminata, poiché non è seguita da alcuna istruzione. In altri termini, quando termina la computazione della seconda chiamata ricorsiva termina anche quella del chiamante. La seconda chiamata può essere così sostituita dagli opportuni assegnamenti dei valori specificati nella chiamata ai parametri della procedura, seguiti da un **goto**, cioè da salto ad un'istruzione specificata da una "etichetta".

```
procedure TORRIDIHANOI(n, origine, destinazione, intermedio: integer);
    label 0, 1;
    var temp: integer;
    begin
0:      if n = 1 then begin
            {'trasferisci un disco da' origine 'a' destinazione};
            goto 1
        end;
        TORRIDIHANOI(n − 1, origine, intermedio, destinazione);
        {'trasferisci un disco da' origine 'a' destinazione};
        n := n − 1; temp := origine;
        origine := intermedio;
        intermedio := temp;
        goto 0
1:  end;
```

Per eliminare l'altra chiamata ricorsiva, occorre utilizzare una pila. Infatti tale chiamata ricorsiva provoca un cambiamento dei valori dei parametri ed è seguita da altre istruzioni. Al ritorno della computazione "interna" i vecchi valori dei parametri sarebbero distrutti, a meno che non siano stati precedentemente salvati nella pila. Si ottiene la seguente procedura iterativa:

```
procedure TORRIDIHANOI(n, origine, destinazione, intermedio: integer);
    label 0, 1, 2, 3;
    var S: pila; temp: integer;
    begin
        CREAPILA(S);
0:      if n = 1 then begin
            {'trasferisci un disco da' origine 'a' destinazione};
            goto 1
        end;
        INPILA(n, S); INPILA(origine, S);
        INPILA(destinazione, S); INPILA(intermedio, S);
        n := n − 1; temp := destinazione;
        destinazione := intermedio; intermedio := temp;
        goto 0;
3:      {'trasferisci un disco da' origine 'a' destinazione};
        n := n − 1; temp := origine;
        origine := intermedio; intermedio := temp;
        goto 0;
1:      if PILAVUOTA(S) then goto 2;
        intermedio := LEGGIPILA(S); FUORIPILA(S);
        destinazione := LEGGIPILA(S); FUORIPILA(S);
        origine := LEGGIPILA(S); FUORIPILA(S);
        n := LEGGIPILA(S); FUORIPILA(S);
        goto 3;
2:  end;
```

Si noti come i parametri siano recuperati dalla pila nell'ordine inverso a quello in cui sono stati salvati. Questa versione iterativa è piena di istruzioni **goto** che la rendono alquanto incomprensibile. Per ottenerne una versione "strutturata", si osservi che il brano di istruzioni comprese tra l'etichetta 0 e il primo "**goto** 0" possono essere sostituite con un **while**, seguito dalle istruzioni racchiuse nel primo **if**. Poiché queste terminano con un "**goto** 1" e 1 è l'etichetta del secondo **if**, si può eliminare il **goto** e spostare il secondo **if**. Poiché il secondo **if** termina con un "**goto** 3", si spostano dopo tale **if** le istruzioni comprese tra l'etichetta 3 e il successivo "**goto** 0". Negando la condizione dell'**if** e introducendo una variabile booleana, si elimina il **goto** introducendo un ciclo **while** più esterno. La versione iterativa priva di istruzioni di salto è la seguente:

```
procedure TORRIDIHANOI(n, origine, destinazione, intermedio: integer);
    var S: pila; temp: integer; finedelmondo: boolean;
    begin
        CREAPILA(S); finedelmondo := false;
        while not finedelmondo do begin
            while n ≠ 1 do begin
                INPILA(n, S); INPILA(origine, S);
                INPILA(destinazione, S); INPILA(intermedio, S);
```



```
                n := n − 1; temp := destinazione;
                destinazione := intermedio;
                intermedio := temp;
            end;
            {'trasferisci un disco da' origine 'a' destinazione};
            if PILAVUOTA(S) then
                finedelmondo := true
            else begin
                intermedio := LEGGIPILA(S); FUORIPILA(S);
                destinazione := LEGGIPILA(S); FUORIPILA(S);
                origine := LEGGIPILA(S); FUORIPILA(S);
                n := LEGGIPILA(S); FUORIPILA(S);
                {'trasferisci un disco da' origine 'a' destinazione};
                n := n − 1; temp := origine;
                origine := intermedio; intermedio := temp;
            end
        end
    end;
```

Se ci sono molte chiamate ricorsive, non è sufficiente salvare nella pila i parametri (e le eventuali variabili locali) del chiamante, ma occorre salvare anche il punto di ritorno, cioè l'etichetta dell'istruzione a partire dalla quale riprendere l'esecuzione al termine della computazione "interna". In pratica, un modo "automatico" di trasformare *ogni* procedura ricorsiva in una iterativa equivalente consiste nell'effettuare le seguenti trasformazioni:

a. Creare una pila subito dopo il **begin** iniziale;
b. Sostituire ogni chiamata ricorsiva con una sequenza di istruzioni che salvano nella pila i valori dei parametri, delle variabili locali e l'etichetta dell'istruzione immediatamente seguente alla chiamata ricorsiva, assegnano ai parametri gli opportuni valori, ed effettuano un salto all'istruzione che segue la creazione della pila;
c. Introdurre prima dell'**end** finale istruzioni che, se la pila non è vuota, estraggono dalla pila i valori salvati, ripristinano i valori dei parametri (e delle variabili locali) e saltano all'istruzione la cui etichetta è uguale al punto di ritorno.

Tali regole sono valide se tutti i parametri sono passati per valore (si consideri per esercizio cosa accade quando ci sono parametri passati per variabile). Per semplicità, risulta conveniente memorizzare nella pila un puntatore ad un record, detto record di attivazione, che contiene lo stato della computazione sospesa, cioè i valori dei parametri, delle variabili locali e del punto di ritorno. Per esempio, applicando le predette regole di trasformazione ed utilizzando un record di attivazione, si ottiene la seguente procedura iterativa per risolvere il problema delle Torri di Hanoi:

```
type tipoelem = ↑attivazione;
    attivazione = record
        dischi: integer;
        piolorig, piolodest, pioloint: integer;
        ritorno: integer
    end
```

```
procedure TORRIDIHANOI(n, origine, destinazione, intermedio: integer);
    label 1, 2, 3;
    var S: pila; temp: integer; stato: tipoelem;
    begin
        CREAPILA(S);
1:      if n = 1 then begin
            {'trasferisci un disco da' origine 'a' destinazione};
            goto 3
        end;
        new(stato);
        stato↑.dischi := n;
        stato↑.piolorig := origine;
        stato↑.piolodest := destinazione;
        stato↑.pioloint := intermedio;
        stato↑.ritorno := 2;
        INPILA(stato, S);
        n := n − 1;
        temp := destinazione;
        destinazione := intermedio;
        intermedio := temp;
        goto 1;
2:      {'trasferisci un disco da' origine 'a' destinazione};
        new(stato);
        stato↑.dischi := n;
        stato↑.piolorig := origine;
        stato↑.piolodest := destinazione;
        stato↑.pioloint := intermedio;
        stato↑.ritorno := 3;
        INPILA(stato, S);
        n := n − 1;
        temp := origine;
        origine := intermedio;
        intermedio := temp;
        goto 1;
3:      if not(PILAVUOTA(S)) then begin
            stato := LEGGIPILA(S);
            FUORIPILA(S);
            n := stato↑.dischi;
            origine := stato↑.piolorig;
            destinazione := stato↑.piolodest;
            intermedio := stato↑.pioloint;
            temp := stato↑.ritorno;
            dispose(stato);
            case temp of
                2: goto 2;
                3: goto 3
            end
        end
    end;
```

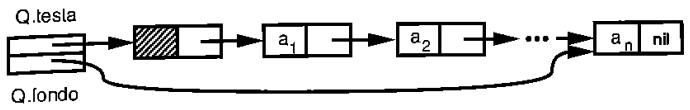



Figura 3.6: Realizzazione monodirezionale con sentinella di una coda.

3.5 Esercizi

Esercizio 3.1. È possibile memorizzare due pile in un unico vettore comune, qualora se ne faccia "crescere" una a partire dal primo elemento del vettore e l'altra a partire dall'ultimo. Si assuma di avere in memoria un solo vettore disponibile per memorizzare le pile. In tal caso, non potranno essere create più di due pile. Si riscrivano le procedure Pascal relative alle usuali operazioni sulle pile, includendo le opportune condizioni di errore qualora si cerchi di creare più di due pile.

Esercizio 3.2. Utilizzando le tecniche di trasformazione illustrate in questo capitolo, si fornisca una versione iterativa della procedura ricorsiva RANGO, che computa il rango di una lista, mostrata nell'Esercizio 2.1.

Esercizio 3.3. Utilizzando le tecniche di trasformazione illustrate in questo capitolo, si fornisca una versione iterativa della procedura ricorsiva PRIMIENNE, che computa la lista dei primi n interi, mostrata nell'Esercizio 2.2.

Esercizio 3.4. Il tipo di dato dequeue (double ended queue) è una sequenza modificabile ad entrambi gli estremi, in cui è possibile inserire un elemento in testa o inserirlo in fondo, e cancellare un elemento dalla testa o cancellarlo dal fondo. Si fornisca una specifica ed una realizzazione, con programmi Pascal, mediante puntatori.

Esercizio 3.5. Si fornisca una realizzazione, con programmi Pascal, del tipo di dato coda in cui si utilizza un vettore circolare con un cursore al primo elemento e un cursore all'ultimo elemento, *senza mantenere la lunghezza della coda*. Si presti particolare attenzione a distinguere il caso di "coda vuota" da quello di "coda piena".

Esercizio 3.6. Si realizzino con procedure Pascal le operazioni della coda, modificando la realizzazione monodirezionale non circolare di una lista, ottenendo complessità $O(1)$ anche per INCODA. Si usino n celle in modo che la prima cella sia indirizzata da un puntatore "testa" e l'ultima da un puntatore "fondo", come mostrato nella Fig. 3.6.

Capitolo 4

Alberi

Un albero ordinato è formato da un insieme finito di elementi, detti nodi. Se tale insieme non è vuoto, allora un particolare nodo è designato come radice, ed i rimanenti nodi, se esistono, sono a loro volta partizionati in insiemi ordinati disgiunti, ciascuno dei quali è un albero ordinato.

L'albero ordinato (spesso chiamato più semplicemente "albero") è una struttura informativa fondamentale, che si presta a rappresentare svariate situazioni, quali:

1. partizioni successive di un insieme in sottoinsiemi disgiunti;
2. organizzazioni gerarchiche di dati;
3. procedimenti enumerativi o decisionali.

Esempio 4.1 (Albero genealogico). L'albero genealogico di una dinastia, a partire dal progenitore, divide gerarchicamente i suoi membri in successive generazioni, fino ad interrompersi in attesa della nascita di nuovi discendenti. La Fig. 4.1 mostra la dinastia dei re merovingi, dove per semplicità ne sono stati indicati solo i principali. Verifichiamo come la struttura di Fig. 4.1 sia costruita in accordo alla definizione ricorsiva di albero ordinato appena introdotta. I nodi sono 16: Meroveo, Childerico I, Clodoveo I, Thierry I, Clodomiro, Childeberto, Clotilde, Clotario I, Sigisberto, Chilperico, Clotario II, Dagoberto I, Sigisberto III, Clodoveo II, Dagoberto II, e Childerico III. Meroveo è la radice dell'albero, ed i rimanenti 15 nodi sono ripartiti in un unico albero con radice Childerico I. Analogamente, sistemato Childerico I, i rimanenti 14 nodi sono ripartiti nell'unico albero con radice Clodoveo I. Si noti che, una volta sistemati i primi tre nodi, gli altri 13 sono ripartiti in 5 alberi, dei quali se ne individua il primo (quello con radice Thierry I), il secondo (con radice Clodomiro), e così via. Ciò spiega perché l'albero sia chiamato "ordinato". □

Esempio 4.2 (Indice di un testo). La suddivisione di un testo in libri, parti, capitoli, paragrafi, sottoparagrafi e capoversi è un valido esempio sia di strutturazione gerarchica che di partizione di dati. La Fig. 4.2 illustra la suddivisione della "Parte Prima" del presente testo fino al capitolo precedente. □

Esempio 4.3 (Classificazioni botaniche e zoologiche). Le classificazioni delle specie animali e vegetali usate nelle scienze naturali rappresentano un altro ben noto esempio di strutturazione gerarchica. La Fig. 4.3 mostra una delle tante classificazioni del Regno Vegetale. □

Esempio 4.4 (Albero sintattico). Si consideri la grammatica libera data dalle seguenti regole di sostituzione: S ::= aB, B ::= b, B ::= bS, B ::= aBB. La sequenza di simboli (stringa) aabb può essere derivata a partire dal simbolo iniziale S sostituendo

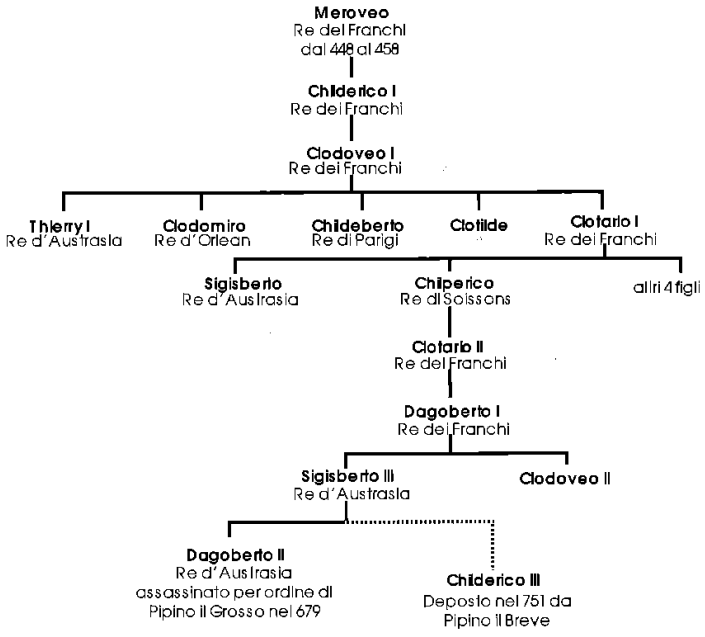


Figura 4.1: La dinastia dei re merovingi.

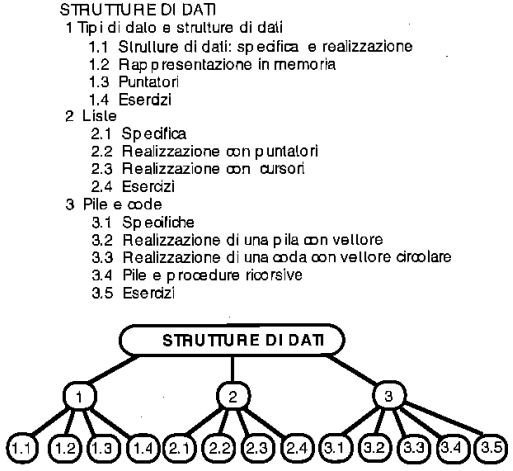


Figura 4.2: Struttura di un testo.

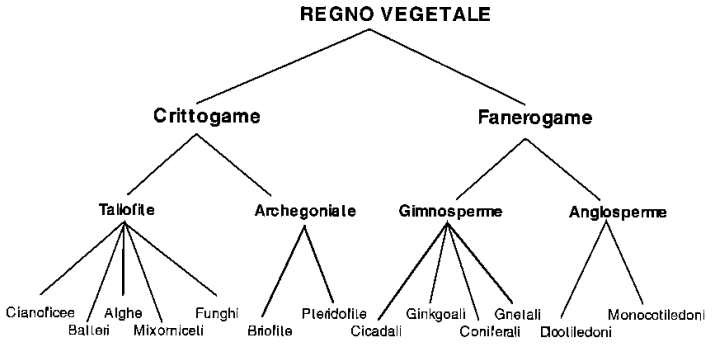


Figura 4.3: Classificazione del regno vegetale.

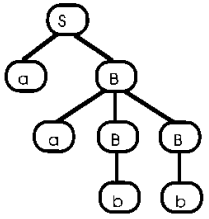


Figura 4.4: Albero sintattico per la derivazione della stringa aabb.

dapprima S con aB, poi B con aBB, indi il primo B con b e infine il secondo B con b, oppure scambiando l'ordine delle due ultime sostituzioni. È possibile rappresentare il procedimento di derivazione di aabb a partire da S indipendentemente dall'ordine di sostituzione, per mezzo di un albero ordinato, detto di derivazione (o sintattico), come mostrato nella Fig. 4.4. □

La terminologia usata nel trattare gli alberi è uno strano misto di termini matematici, botanici e di parentela. Sia T un albero ordinato di n nodi di radice r. Siano $T_1, \ldots, T_k$ gli insiemi disgiunti non vuoti in cui sono partizionati i rimanenti $n-1$ nodi di T, aventi come radici, rispettivamente, i nodi $r_1, \ldots, r_k$. Ciascun T_i è detto sottoalbero di T radicato in r_i, mentre ciascun r_i è detto figlio di r. I nodi $r_1, \ldots, r_k$ sono tra loro fratelli, ed r è il loro padre. Un discendente di un nodo u è un figlio di u oppure un figlio di un discendente di u; analogamente, un antenato di u è suo padre oppure il padre di un suo antenato. Un nodo senza figli è detto foglia, mentre la radice dell'albero è l'unico nodo senza padre. In pratica, in un albero ordinato è stabilito un doppio ordinamento tra i nodi. In primo luogo c'è un ordinamento "verticale" per livelli, cioè in base alla distanza dei nodi dalla radice. La radice r è a livello 0, tutti i figli di r stanno al livello 1, i figli dei figli di r stanno al livello 2, e così via fino all'esaurimento dei nodi. In secondo luogo, è stabilito un ordinamento "orizzontale" tra i nodi che stanno allo stesso livello, distinguendo così tra il primo figlio di un nodo

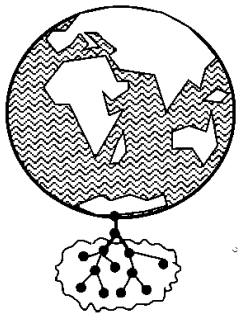


Figura 4.5: L'albero nel mondo informatico.

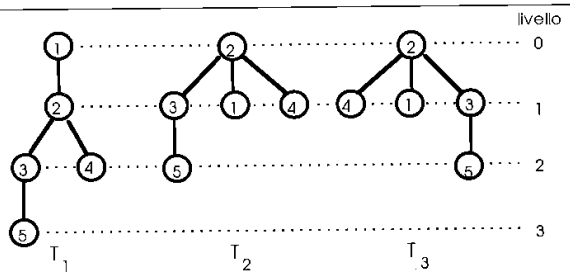


Figura 4.6: Alberi ordinati.

(il figlio maggiore), il suo secondo figlio, e così via fino all'ultimo (il figlio minore). Come visto negli esempi precedenti, i nodi si disegnano spesso con cerchietti, mentre coppie padre-figlio si visualizzano con linee che collegano coppie di cerchietti. La terminologia impiegata pare di origine patriarcale (si usano i termini padre, figlio e fratello, anziché madre, figlia e sorella). Inoltre gli alberi sono usualmente rappresentati rovesciati, con la radice in alto e le foglie in basso (si veda la Fig. 4.5).

Esempio 4.5 (Alberi ordinati). La Fig. 4.6 mostra tre alberi ordinati T_1, T_2, e T_3. Pur essendo formati dagli stessi 5 nodi, i tre alberi sono diversi tra loro. Infatti, la radice di T_1 (il nodo 1) è diversa da quella di T_2 e T_3 (il nodo 2). Inoltre, considerando i nodi allo stesso livello ordinati da sinistra verso destra, T_2 e T_3 sono diversi tra loro. Infatti, pur avendo la stessa radice (il nodo 2), gli stessi figli della radice (i nodi 1, 3, 4), e lo stesso figlio del nodo 3 (il nodo 5), l'ordine dei figli di 2 non è lo stesso in T_2 e in T_3. □

4.1 Specifica

L'albero ordinato può essere interpretato come una generalizzazione di una sequenza lineare, in cui ciascun elemento ha ancora un solo predecessore (il padre), ma

in generale può avere più di un successore (i figli). Infatti, una lista è un particolare albero in cui ogni nodo ha al più un figlio e c'è una sola foglia. Analogamente alle operazioni sulle liste, quelle definite sugli alberi ordinati prevedono la possibilità di spostarsi da un nodo ad un suo parente, p.e. figlio, fratello o padre, e inoltre di inserire o cancellare interi sottoalberi.

Un tipico insieme di operazioni per gli alberi ordinati comprende l'operazione di creazione CREAALBERO, quelle di selezione RADICE, per accedere direttamente alla radice, e PADRE, PRIMOFIGLIO e SUCCFRATELLO, per accedere, rispettivamente, al padre, al primo figlio, o al fratello successivo di un nodo, quelle di interrogazione ALBEROVUOTO, per verificare che l'albero sia vuoto, FOGLIA e FINEFRATELLI, per controllare che un nodo non abbia figli o che siano stati considerati tutti i fratelli, e quelle di modifica INSRADICE, per inserire la radice in un albero vuoto, CANCSOTTOALBERO e INSSOTTOALBERO, per cancellare o inserire un sottoalbero.

CREAALBERO:	() → albero
ALBEROVUOTO:	(albero) → booleano
INSRADICE:	(nodo, albero) → albero
RADICE:	(albero) → nodo
PADRE:	(nodo, albero) → nodo
FOGLIA:	(nodo, albero) → booleano
PRIMOFIGLIO:	(nodo, albero) → nodo
FINEFRATELLI:	(nodo, albero) → booleano
SUCCFRATELLO:	(nodo, albero) → nodo
INSSOTTOALBERO:	(nodo, albero, albero) → albero
CANCSOTTOALBERO:	(nodo, albero) → albero

Indicando per semplicità col termine "albero" l'insieme di tutti gli alberi ordinati, come definiti precedentemente, e con T un generico albero ordinato, la specifica semantica è la seguente.

CREALBERO() = T'
Post: $T' = \Lambda$, l'albero vuoto

ALBEROVUOTO(T) = b
Post: b = vero, se $T = \Lambda$; b = falso, altrimenti

INSRADICE(u, T) = T'
Pre: $T = \Lambda$
Post: T' è formato dalla sola radice u

RADICE(T) = u
Pre: $T \neq \Lambda$
Post: u è la radice di T

PADRE(u, T) = v
Pre: $T \neq \Lambda$, u è un nodo di T, ma non la sua radice
Post: v è il padre di u in T

FOGLIA(u, T) = b
Pre: $T \neq \Lambda$, u è un nodo di T
Post: b = vero, se non esiste alcun figlio di u; b = falso, altrimenti

PRIMOFIGLIO(u, T) = v
Pre: $T \neq \Lambda$, u è un nodo di T, FOGLIA(u, T) = falso
Post: v è il primo figlio di u secondo la relazione di precedenza stabilita tra i figli di u

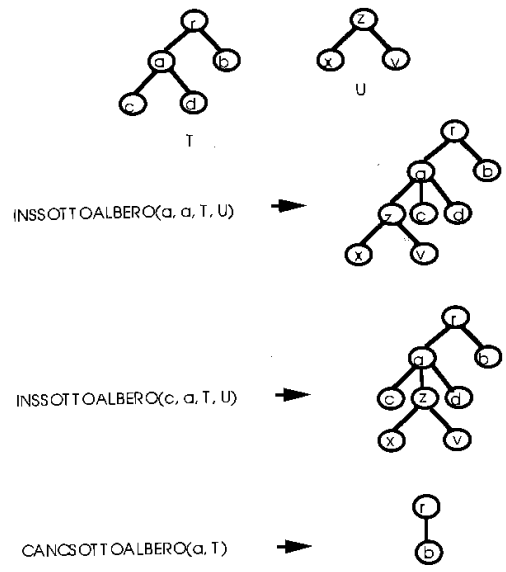


Figura 4.7: Inserzioni e cancellazioni di sottoalberi.

FINEFRATELLI(u, T) = b
Pre: $T \neq \Lambda$, u è un nodo di T o un nodo "sentinella" s che segue l'ultimo fratello di ciascun nodo
Post: b = vero, se $u = s$; b = falso, altrimenti

SUCCFRATELLO(u, T) = v
Pre: $T \neq \Lambda$, u è un nodo di T
Post: v è il fratello di u che lo segue immediatamente u nella relazione di precedenza (e quindi v è la sentinella s se u è l'ultimo fratello)

INSSOTTOALBERO(u, v, T, U) = T'
Pre: $T \neq \Lambda$, $U \neq \Lambda$, u e v sono nodi di T, v è o un figlio di u oppure u stesso, e T e U non hanno nodi in comune
Post: T' è ottenuto da T aggiungendovi U come sottoalbero: la radice z di U diventa il nuovo fratello che segue immediatamente u nella relazione di precedenza, se $u \neq v$, oppure il nuovo primo figlio di u, se $u = v$

CANCSOTTOALBERO(u, T) = T'
Pre: $T \neq \Lambda$, u è un nodo di T
Post: T' è ottenuto da T togliendovi il sottoalbero di radice u, cioè u stesso e tutti i discendenti di u (e quindi $T' = \Lambda$ se u è la radice di T)

Esempio 4.6 (Inserzione e cancellazione di sottoalberi). Gli effetti delle operazioni INSSOTTOALBERO e CANCSOTTOALBERO sono chiariti nella Fig. 4.7. □

4.2 Visite

La visita di un albero ordinato consiste nel seguire una "rotta" di viaggio che consenta di esaminare ogni nodo dell'albero esattamente una volta. La visita può essere eseguita in vari modi, detti "ordini", tra cui i 3 più importanti sono: ordine anticipato (o previsita), differito (o postvisita), e simmetrico (o invisita).

Sia T un albero non vuoto di radice r. Se r non è una foglia ma ha $k > 0$ figli, indichiamo con $T_1, \ldots, T_k$ i k sottoalberi di T radicati nei k figli di r. I predetti ordini di visita sono definiti ricorsivamente come segue:

1. la previsita di T consiste nell'esaminare r e poi nell'effettuare, nell'ordine, la previsita di $T_1, \ldots, T_k$;
2. la postvisita di T consiste nell'effettuare, nell'ordine, la postvisita di $T_1, \ldots, T_k$ e poi nell'esaminare r;
3. Fissato $i \geq 1$, la invisita di T consiste nell'effettuare, nell'ordine, la invisita di $T_1, \ldots, T_i$, nell'esaminare r, e poi nell'effettuare, nell'ordine, la invisita di $T_{i+1}, \ldots, T_k$.

Esempio 4.7 (Visita di alberi ordinati). I tre ordini di visita dei nodi dell'albero della Fig. 4.3 sono:

Previsita: Regno vegetale, Crittogame, Tallofite, Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Archegoniate, Briofite, Pteridofite, Fanerogame, Gimnosperme, Cicadali, Ginkgoali, Coniferali, Gnetali, Angiosperme, Dicotiledoni, Monocotiledoni;

Postvisita: Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Tallofite, Briofite, Pteridofite, Archegoniate, Crittogame, Cicadali, Ginkgoali, Coniferali, Gnetali, Gimnosperme, Dicotiledoni, Monocotiledoni, Angiosperme, Fanerogame, Regno vegetale;

Invisita ($i = 1$): Cianoficee, Tallofite, Batteri, Alghe, Mixomiceti, Funghi, Crittogame, Briofite, Archegoniate, Pteridofite, Regno vegetale, Cicadali, Gimnosperme, Ginkgoali, Coniferali, Gnetali, Fanerogame, Dicotiledoni, Angiosperme, Monocotiledoni. □

In pratica, l'ordine di visita anticipato coincide con quello del diritto di successione al trono nell'albero genealogico di una dinastia regnante. Per illustrare la differenza tra gli ordini di visita anticipato e differito, c'è un modo pittoresco che non tira in ballo la ricorsione. Si immagini che l'albero sia una sorta di isola fatta a fiordo: i cerchietti rappresentano fari e le linee che collegano coppie di fari rappresentano alte scogliere. Una nave fa la circumnavigazione dell'isola, partendo dalla radice e costeggiando sempre le scogliere. Il capitano segna sul libro di bordo il nome di ciascun faro che viene incontrato nella navigazione, ma solo quando lo incontra per la prima volta. È facile verificare che se la circumnavigazione procede in senso antiorario, allora si ottiene proprio l'ordine di visita anticipato, mentre se procede in senso orario si ottiene l'ordine di visita differito inverso (cioè letto da destra verso sinistra, ovvero dall'ultimo nodo visitato al primo). Il seguente schema di procedura Pascal serve per effettuare sia la visita anticipata che quella differita.

```
procedure VISITA(u: nodo; var T: albero);
    var v: nodo;
    begin
(1)     {esame "anticipato" del nodo u};
        if not (FOGLIA(u, T)) then begin
            v := PRIMOFIGLIO(u, T);
            while not (FINEFRATELLI(v, T)) do begin
                PREVISITA(v, T);
                v := SUCCFRATELLO(v, T)
            end
        end;
(2)     {esame "differito" del nodo u};
    end;
```


La chiamata nel programma principale è:

if not ALBEROVUOTO(T) **then** VISITA(RADICE(T), T).

Se l'esame del nodo u viene effettuato solo nella riga (1), ignorando la riga (2), si ottiene la procedura PREVISITA, mentre se viene effettuato solo nella riga (2), ignorando la riga (1), allora si ottiene la procedura POSTVISITA. La procedura INVISITA invece segue uno schema diverso. Per $i = 1$ lo schema è il seguente:

```
procedure INVISITA(u: nodo; var T: albero);
    var v: nodo;
    begin
        if FOGLIA(u, T) then
            {esamina il nodo u}
        else begin
            v := PRIMOFIGLIO(u, T);
            INVISITA(v, T);
            {esame "simmetrico" del nodo u};
            v := SUCCFRATELLO(v, T);
            while not (FINEFRATELLI(v, T)) do begin
                INVISITA(v, T);
                v := SUCCFRATELLO(v, T)
            end
        end
    end;
```

Nelle tre procedure di visita descritte, viene effettuata esattamente una chiamata ricorsiva per ciascun nodo, per un totale di n chiamate, dove n è il numero di nodi dell'albero. Pertanto, la complessità delle tre procedure di visita è $\Theta(n)$, purché si fornisca una realizzazione in cui gli operatori utilizzati nelle procedure di visita richiedano ciascuno $O(1)$ tempo.

4.3 Realizzazione con vettore dei padri

Si supponga che i nodi dell'albero T siano numerati da 1 ad n. La più semplice realizzazione fa uso di un vettore contenente, per ogni nodo i, $1 \leq i \leq n$, il cursore al padre: $T[i] = j$, se j è il padre di i; $T[i] = 0$, se i è la radice.

Esempio 4.8 (Vettore dei padri). Il contenuto del vettore dei padri per l'albero T_1 della Fig. 4.6 è: $T[1] = 0$, $T[2] = 1$, $T[3] = 2$, $T[4] = 2$, $T[5] = 3$. □

Con questa realizzazione è facile visitare i nodi lungo percorsi che vadano dal basso verso l'alto, cioè dalle foglie verso la radice. In particolare, l'operazione PADRE richiede tempo $O(1)$. Viceversa, è costoso passare da un nodo ai suoi figli, individuare il livello di un nodo, e inserire e cancellare sottoalberi. Infine, non è

chiara la relazione di precedenza tra fratelli, a meno che non se ne assuma una implicitamente, per esempio imponendo che un figlio abbia sempre un numero maggiore del padre e che nodi fratelli siano numerati in modo crescente da sinistra verso destra (ciò implica che la radice è il nodo 1).

Le procedure Pascal che realizzano gli operatori degli alberi sono lasciate per esercizio. Assumendo la suddetta relazione di precedenza tra fratelli, è qui riportata come esempio la function PRIMOFIGLIO, la cui complessità è $O(n)$.

```
type
    nodo = integer;
    albero = array [1..maxlung] of nodo;

function PRIMOFIGLIO(u: nodo; var T: albero): nodo;
    var i: integer; trovato: boolean;
    begin
        trovato := false;
        i := 1;
        while (not trovato) and (i ≤ n) do
            if T[i] = u
                then trovato := true
                else i := i + 1;
        PRIMOFIGLIO := i;
    end;
```

4.4 Realizzazione con liste dei figli

Si supponga ancora che i nodi dell'albero T siano numerati da 1 ad n. Una realizzazione talvolta usata consiste nel mantenere un vettore contenente, per ogni nodo i, $1 \leq i \leq n$, l'identificatore (puntatore o cursore) di una lista di tutti figli del nodo i.

Trascurando la realizzazione del tipo di dato lista, si hanno le seguenti dichiarazioni:

```
type
    nodo = integer;
    albero = record
        testa: array [1..maxlung] of lista;
        radice: nodo
    end;
```

Gli operatori degli alberi possono essere realizzati in termini degli operatori delle liste. Per esempio,

PRIMOFIGLIO = LEGGILISTA(PRIMOLISTA(*T.testa*[*u*]), *T.testa*[*u*]).

Ovviamente, la realizzazione con liste dei figli presenta il vantaggio di poter scorrere velocemente tutti i figli di un dato nodo, ma non permette di realizzare efficientemente altri operatori come PADRE o SUCCFRATELLO. Infatti, per poter individuare il padre o il fratello successivo di un dato nodo u, occorre scandire nel caso pessimo tutte le n liste alla ricerca dell'unica lista nella quale compare il nodo u.


```
function PADRE(u: nodo; var T: albero): nodo;
    var i: nodo;
    begin
        i := 1;
        while (not APPARTIENE(u, T.testa[u]) and (i ≤ n) do i := i + 1
        PADRE := i
    end;

function APPARTIENE(u: nodo; var L: lista): boolean;
    var p: posizione; trovato: boolean;
    begin
        p := PRIMOLISTA(L); trovato := false;
        while (not trovato) and (not FINELISTA(L)) do
            if LEGGILISTA(p, L) = u then trovato := true;
        APPARTIENE := trovato
    end;
```

Per realizzare SUCCFRATELLO, occorre essenzialmente ripetere le stesse operazioni effettuate per PADRE, modificando la funzione APPARTIENE in modo da restituire anche il nodo che segue u nella lista L. Anche assumendo una realizzazione efficiente per le liste, le complessità di PADRE e SUCCFRATELLO non scendono sotto $O(n)$. Infatti, indicato con f_i il numero di figli del nodo i, dato che ogni nodo tranne la radice compare una sola volta e in una sola lista, si ha che $f_1 + \cdots + f_n = n - 1$. Per trovare u, occorre scandire nel caso pessimo tutte le n liste dei figli, per un totale di $O(f_1 + \cdots + f_n) = O(n)$ tempo.

4.5 Realizzazione con puntatori padre/primofiglio/fratello

La realizzazione con liste dei figli non si presta bene ad eseguire operazioni come PADRE, SUCCFRATELLO, o INSSOTTOALBERO. Per realizzare PADRE e SUCCFRATELLO in tempo $O(1)$, si potrebbero introdurre due campi con questi valori in ciascuna cella delle liste, oppure usare, oltre al vettore testa, anche un vettore dei padri e un vettore dei fratelli successivi. Ciò comunque non renderebbe efficiente la INSSOTTOALBERO. Infatti, poiché ciascun albero ha il proprio vettore testa, un'inserzione richiede la copiatura di due alberi in un terzo albero. Per effettuare un'inserzione efficientemente, occorre che tutti gli alberi condividano un'area comune, così come le liste condividono la memoria indirizzabile dai puntatori nella realizzazione con puntatori o il vettore SPAZIO nella realizzazione con cursori. Inoltre, è necessario che tutte le celle dell'area comune abbiano la stessa struttura dove siano immediatamente reperibili le informazioni per raggiungere il padre, il primo figlio, e il fratello successivo di un nodo. Questo scopo è raggiunto imponendo che ogni cella contenga almeno tre puntatori (o cursori): uno al padre, uno al primo figlio e uno al fratello successivo. In tal caso, il nodo diventa un puntatore (o cursore) nell'area comune e non può più essere identificato col valore del nodo stesso, e in ogni cella occorre un campo apposito per il valore del nodo.

A tal fine, si introducono due ulteriori operazioni, una di lettura LEGGINODO, per leggere il valore contenuto in un nodo, e una di modifica SCRIVINODO, per cambiare il valore memorizzato in un nodo, la cui specifica è la seguente.

LEGGINODO: (nodo, albero) → tipoelem
SCRIVINODO: (tipoelem, nodo, albero) → albero

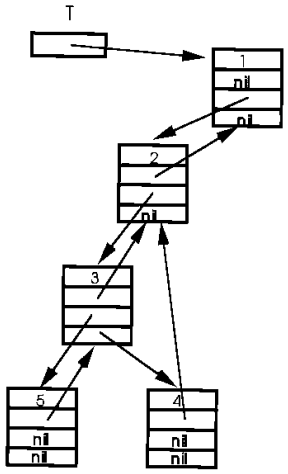


Figura 4.8: Realizzazione con puntatori padre/primofiglio/fratello.

LEGGINODO(u, T) = a
Pre: $T \neq \Lambda$, u è un nodo di T
Post: a è il valore memorizzato nel nodo u
SCRIVINODO(a, u, T)
Pre: $T \neq \Lambda$, u è un nodo di T
Post: T' è ottenuto da T scrivendo il valore a nel nodo u

Con questa nuova realizzazione, detta a puntatori (o cursori) padre/primofiglio/fratello, nodo e valore vengono ad assumere, rispettivamente, ruoli analoghi a quelli di posizione e elemento nelle liste. Infatti, le liste altro non sono che particolari alberi ordinati, in cui ogni nodo ha esattamente un figlio, tranne l'ultimo nodo della sequenza che è l'unica foglia. Nella realizzazione con puntatori, le dichiarazioni sono:

```
type
    albero = ↑cella;
    nodo = albero;
    cella = record
        valore: tipoelem;
        genitore, figlio, fratello: nodo;
    end;
```

Esempio 4.9 (Puntatori padre/primofiglio/fratello). La Fig. 4.8 mostra la realizzazione con puntatori padre/primofiglio/fratello dell'albero T_1 di Fig. 4.6. □

Utilizzando cursori, l'area comune è un vettore AREALIBERA, che deve essere inizializzato in modo da formare una listalibera [cfr. § 2.3]. Le relative dichiarazioni sono:

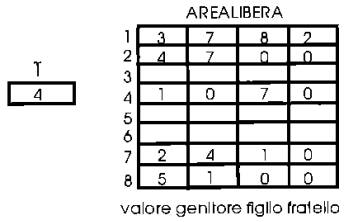


Figura 4.9: Realizzazione con cursori padre/primofiglio/fratello.

```
type
    albero = integer;
    nodo = integer;
var AREALIBERA: array [1..maxlung] of record
        valore: tipoelem;
        genitore, figlio, fratello: nodo;
    end;
```

Esempio 4.10 (Cursori padre/primofiglio/fratello). La Fig. 4.9 mostra una possibile configurazione del vettore comune AREALIBERA, in cui è memorizzato con cursori padre/primofiglio/fratello l'albero T_1 di Fig. 4.6. □

Utilizzando la realizzazione con puntatori (o cursori), ogni operatore, tranne CANCSOTTOALBERO, richiede tempo O(1) per essere eseguito (la realizzazione di tutti gli operatori è lasciata per esercizio). Per realizzare CANCSOTTOALBERO in tempo O(1), occorre che ogni cella, oltre a contenere un puntatore (o cursore) al padre e al fratello successivo, contenga anche un puntatore (o cursore) al fratello precedente. In tal caso, l'aggiornamento dei puntatori (cursori) relativi ai fratelli e al padre della radice del sottoalbero rimosso può essere fatto in tempo costante. Se si vogliono recuperare nella memoria le celle del sottoalbero rimosso, però, il tempo aumenta. Infatti, occorre una visita in ordine differito del sottoalbero per staccare e recuperare un nodo alla volta (tramite *dispose* nel caso dei puntatori). A partire dalle foglie, vengono staccati e recuperati via via i nodi che non hanno (più) figli, fino all'esaurimento dei nodi. Tale recupero richiede tempo lineare nel numero di nodi del sottoalbero.

4.6 Alberi binari

Un albero binario è un particolare albero ordinato in cui ogni nodo ha al più due figli e si fa distinzione tra il figlio sinistro ed il figlio destro di un nodo. Questa proprietà è assai sottile, poiché impone che due alberi T e U aventi gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, siano distinti qualora un nodo u sia designato come figlio sinistro di un nodo v in T e come figlio destro del medesimo nodo in U.

Esempio 4.11 (Alberi binari). La Fig. 4.10 mostra tre alberi $T_1, \ldots, T_3$. Visti come alberi ordinati, essi sono tutti identici tra loro. T_1 non è binario, perché non è chiara la convenzione per distinguere i figli sinistri dai destri. Considerando i figli disegnati

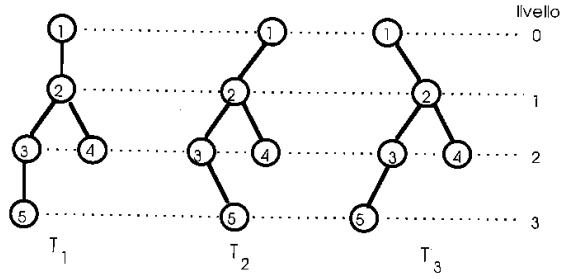


Figura 4.10: Alberi ordinati e alberi binari.

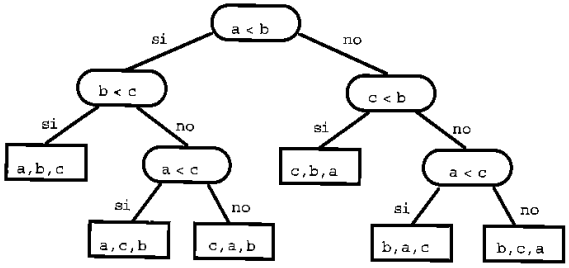


Figura 4.11: Albero di decisione per l'ordinamento di tre numeri.

a sinistra (destra) come figli sinistri (destri), allora T_2 e T_3 sono binari; inoltre essi sono tra loro distinti, poiché non hanno gli stessi figli destri e sinistri. □

L'importanza degli alberi binari risiede sia sulla semplicità della loro struttura che sulle innumerevoli applicazioni in cui possono essere impiegati.

Esempio 4.12 (Albero di decisione). Le possibili sequenze di confronti (a due alternative) effettuate da un algoritmo per risolvere un problema possono essere visualizzate con un albero binario, detto albero di decisione. In tale albero, i nodi con figli rappresentano confronti tra dati del problema, le coppie padri-figli i risultati dei confronti, e le foglie soluzioni possibili del problema. Un percorso radice-foglia rappresenta la sequenza di confronti effettuati per individuare una soluzione e il livello massimo di una foglia dà il numero di confronti effettuati nel caso pessimo. Un albero di decisione che minimizzi il livello massimo delle foglie fornisce con tale valore una limitazione inferiore al numero di decisioni che ogni algoritmo che risolva il problema deve effettuare nel caso peggiore.

La Fig. 4.11 illustra un albero di decisione per il problema di ordinare tre numeri a, b, c. Poiché un qualsiasi algoritmo di ordinamento di n numeri deve fare una sequenza di scelte che permetta di individuare la soluzione corretta tra le $n!$ permutazioni possibili dei dati di ingresso, l'albero di decisione per un qualsiasi algoritmo di ordinamento ha almeno $n!$ foglie. Poiché ad ogni livello dell'albero il numero di nodi al più raddoppia, la lunghezza del percorso radice-foglia più lungo non può essere in-

feriore a $\log_2 n!$ Essendo $n! = n(n-1)\cdots 3\cdot 2\cdot 1 \geq n(n-1)\cdots(n/2) \geq (n/2)^{n/2}$, si ha $\log_2 n! \geq (n/2)\log_2(n/2)$. Pertanto una *limitazione inferiore* alla complessità del problema dell'ordinamento è $\Omega(n \log n)$. □

Una specifica ragionevole per gli alberi binari può includere gli operatori CREABINALBERO, BINALBEROVUOTO, BINRADICE, BINPADRE, CANCSOTTOBINALBERO, LEGGINODO, SCRIVINODO (che sono analoghi ai rispettivi operatori per gli alberi ordinati) ed i nuovi operatori:

FIGLIOSINISTRO:	(nodo, binalbero) → nodo
FIGLIODESTRO:	(nodo, binalbero) → nodo
SINISTROVUOTO:	(nodo, binalbero) → booleano
DESTROVUOTO:	(nodo, binalbero) → booleano
COSTRBINALBERO:	(binalbero, binalbero) → binalbero

la cui specifica semantica è:

FIGLIOSINISTRO(u, T) = v
Pre: $T \neq \Lambda$, u è in T, u ha un figlio sinistro
Post: v è il figlio sinistro di u in T

FIGLIODESTRO(u, T) = v
Pre: $T \neq \Lambda$, u è in T, u ha un figlio destro
Post: v è il figlio destro di u in T

SINISTROVUOTO(u, T) = b
Pre: $T \neq \Lambda$, u è in T
Post: b = vero, se u non ha figlio sinistro; b = falso, altrimenti

DESTROVUOTO(u, T) = b
Pre: $T \neq \Lambda$, u è in T
Post: b = vero, se u non ha figlio destro; b = falso, altrimenti

COSTRBINALBERO(T, U) = T'
Post: T' è ottenuto introducendo un nuovo nodo u, che è la radice di T', che ha come sottoalbero sinistro T e come sottoalbero destro U (se $T = \Lambda$, u non ha figlio sinistro; se $U = \Lambda$, u non ha figlio destro)

Si noti che l'operatore FOGLIA è ridondante, poiché FOGLIA(u, T) = vero se e solo se SINISTROVUOTO(u, T) = DESTROVUOTO(u, T) = vero.

Utilizzando gli operatori introdotti, la visita di un albero binario può essere effettuata molto semplicemente nel modo seguente:

```
procedure BINVISITA(u: nodo; var T: binalbero);
  begin
(1)     {esame "anticipato" del nodo u};
        if not SINISTROVUOTO(u, T) then
            BINVISITA(FIGLIOSINISTRO(u, T), T);
(2)     {esame "simmetrico" del nodo u}
        if not DESTROVUOTO(u, T) then
            BINVISITA(FIGLIODESTRO(u, T), T);
(3)     {esame "differito" del nodo u}
  end;
```

La previsita si ottiene esaminando il nodo u soltanto nella riga (1) e ignorando le righe (2) e (3). Analogamente, la invisita o la postvisita si hanno esaminando il nodo u solo nella riga (2) o in (3), rispettivamente.

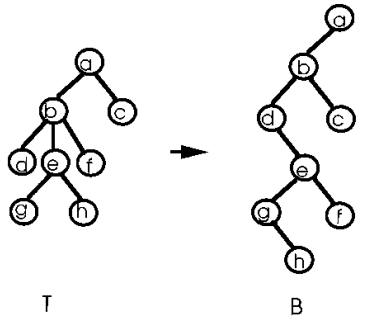


Figura 4.12: Rappresentazione di un albero ordinato T con un albero binario B.

Una realizzazione ragionevole fa uso di un'area comune AREALIBERA, come nel caso degli alberi ordinati, in cui i campi "figlio" e "fratello" sono chiamati, invece, "sinistro" e "destro". La sostanziale equivalenza di queste due realizzazioni non deve sorprendere, poiché nel proporre quella per gli alberi ordinati si è implicitamente trasformato l'albero ordinato in uno binario!

Esempio 4.13 (Rappresentazione binaria di alberi ordinati). Come illustrato nella Fig. 4.12, è sempre possibile rappresentare un albero ordinato T con un albero binario B avente gli stessi nodi e la stessa radice, in cui ogni nodo di B ha come figlio sinistro il primo figlio che ha in T e come figlio destro il fratello successivo che ha in T. È facile verificare che le sequenze dei nodi esaminati su T e su B coincidono se T e B sono entrambi visitati in ordine anticipato, oppure se T è visitato in ordine differito e B è visitato in ordine simmetrico. □

Ovviamente, è possibile usare puntatori anziché cursori. Indicando la mancanza di un figlio col valore **nil** contenuto nell'apposito campo, si ha:

```
type
    binalbero = ↑bincella;
    nodo = binalbero;
    bincella = record
        valore: tipoelem;
        sinistro, destro, genitore: nodo;
    end;

procedure CREABINALBERO(var T: binalbero);
    begin
        T := nil
    end;
function BINALBEROVUOTO(var T: binalbero): boolean;
    begin
        BINALBEROVUOTO := (T = nil)
    end;
```



```
function BINRADICE(var T: binalbero): nodo;
    begin
        BINRADICE := T
    end;

function BINPADRE(u: nodo; var T: binalbero): nodo;
    begin
        BINPADRE := u↑.genitore
    end;

function FIGLIOSINISTRO(u: nodo; var T: binalbero): nodo;
    begin
        FIGLIOSINISTRO := u↑.sinistro
    end;

function FIGLIODESTRO(u: nodo; var T: binalbero): nodo;
    begin
        FIGLIODESTRO := u↑.destro
    end;

function SINISTROVUOTO(u: nodo; var T: binalbero): boolean;
    begin
        SINISTROVUOTO := (u↑.sinistro = nil)
    end;

function DESTROVUOTO(u: nodo; var T: binalbero): boolean;
    begin
        DESTROVUOTO := (u↑.destro = nil)
    end;

procedure COSTRBINALBERO(var T, U: binalbero);
    var u: nodo;
    begin
        new(u);
        u↑.sinistro := T; u↑.destro := U; u↑.genitore := nil;
        if T ≠ nil then T↑.genitore := u;
        if U ≠ nil then u↑.genitore := u;
        T := u
    end;
```

La complessità degli operatori è $O(1)$. La realizzazione dell'operatore CANCSOTTOBINALBERO è lasciata per esercizio.

4.7 Esercizi

Esercizio 4.1 (Profondità). La profondità di un albero ordinato è il massimo livello delle sue foglie. Si fornisca una funzione Pascal che calcoli in tempo ottimo la profondità di un albero ordinato T di n nodi.

Poiché la limitazione inferiore è $\Omega(n)$ per la dimensione dei dati, la funzione deve richiedere tempo $O(n)$. Si utilizza lo schema della visita differita nella quale si

calcola la profondità di ogni sottoalbero di T. La profondità di un sottoalbero radicato in un generico nodo u è zero, se u è una foglia, mentre si ottiene sommando uno alla profondità massima dei sottoalberi radicati nei figli di u, altrimenti. Il tempo richiesto è $\Theta(n)$ e quindi ottimo. La chiamata della funzione è PROFONDITA(RADICE(T), T).

```
function PROFONDITA(u: nodo; var T: albero): integer;
    var v: nodo; max, temp: integer;
    begin
        if FOGLIA(u, T) then
            PROFONDITA := 0
        else begin
            v := PRIMOFIGLIO(u, T);
            max := 0;
            while not FINEFRATELLI(v, T) do begin
                temp := PROFONDITA(v, T);
                if max ≤ temp then max := temp;
                v := SUCCFRATELLO(v, T)
            end
            PROFONDITA := max + 1;
        end
    end;
```

Esercizio 4.2 (Cancella le foglie con somma k). Dato un albero ordinato i cui nodi contengono valori interi, se ne vogliono cancellare tutte le foglie per le quali il percorso radice-foglia ha somma complessiva dei valori uguale a k. Fornire una procedura Pascal di complessità ottima.

Una limitazione inferiore alla complessità del problema è $\Omega(n)$ poiché è necessario esaminare tutte le foglie dell'albero. L'algoritmo risolutivo si basa su una visita anticipata. Durante la discesa calcola per ogni nodo u la somma dei valori contenuti nei nodi incontrati a partire dalla radice fino ad u. Quando giunge ad una foglia, la cancella se tale somma è uguale a k. Si assume una realizzazione con puntatori padre/primofiglio/fratello. Per cancellare la foglia usa l'operatore CANCSOTTOALBERO, che richiede tempo $O(1)$ per cancellare un sottoalbero di un solo nodo. La complessità della procedura è $\Theta(n)$ e la chiamata è FOGLIEK(RADICE(T), T, 0, k).

```
procedure FOGLIEK(u: nodo; var T: albero; somma, k: integer);
    var v: nodo;
    begin
        if not FOGLIA(u, T) then begin
            v := PRIMOFIGLIO(u, T);
            while not FINEFRATELLI(v, T) do begin
                FOGLIEK(v, T, somma + LEGGINODO(u, T), k);
                v := SUCCFRATELLO(v, T)
            end
        end else
            if somma + LEGGINODO(u, T) = k then
                CANCSOTTOALBERO(u, T);
    end;
```

Esercizio 4.3 (Larghezza). La larghezza di un albero ordinato è il numero massimo di nodi che stanno tutti al medesimo livello. Si fornisca una funzione Pascal che calcoli in tempo ottimo la larghezza di un albero ordinato T di n nodi.

Esercizio 4.4 (Visite). Gli ordini di visita di un albero binario di 9 nodi sono i seguenti: A, E, B, F, G, C, D, I, H (anticipato), B, G, C, F, E, H, I, D, A (differito) e B, E, G, F, C, A, D, H, I (simmetrico). Si ricostruisca l'albero binario.

Esercizio 4.5 (Numero nodi per sottoalbero). Dato un albero binario T non vuoto, si vuole memorizzare, in ciascun nodo u, il numero di nodi che si trovano nel sottoalbero radicato in u. Si fornisca un algoritmo lineare nel numero di nodi.

Si procede con una visita differita, memorizzando in ciascun nodo u il valore 1, se u è una foglia, o 1 più la somma dei valori contenuti nel figlio sinistro e nel figlio destro, altrimenti.

```
procedure CONTANODI(u: nodo; var T: binalbero);
    var s, d: integer;
    begin
        if FOGLIA(u, T) then
            SCRIVINODO(u, T) := 1
        else begin
            if not(SINISTROVUOTO(u, T) then begin
                CONTANODI(FIGLIOSINISTRO(u, T), T);
                s := LEGGINODO(FIGLIOSINISTRO(u, T), T)
            end else s := 0;
            if not DESTROVUOTO(u, T) then begin
                CONTANODI(FIGLIODESTRO(u, T), T);
                d := LEGGINODO(FIGLIODESTRO(u, T), T)
            end else d := 0;
            SCRIVINODO(u, T) := s + d + 1;
        end;
    end;
```

Esercizio 4.6 (Cancella foglie). Dato un albero binario i cui nodi contengono interi, si vuole cancellare ogni foglia che sia un figlio sinistro e contenga lo stesso intero del padre. Si scriva una procedura Pascal ricorsiva di complessità ottima assumendo che l'albero binario sia realizzato con puntatori.

Tutte le foglie e quindi tutti i nodi dell'albero devono essere visitati nel caso pessimo. L'algoritmo proposto è basato su una previsita e ha complessità lineare nel numero di nodi e quindi ottima. Conoscendo la realizzazione, l'algoritmo lavora direttamente con i puntatori e le celle, senza usare gli usuali operatori dell'albero binario.

```
procedure CANCELLAFOGLIA(u: nodo; var T: binalbero);
    begin
        if (u↑.sinistro = nil) and (u↑.destro = nil) then begin
            if (u↑.genitore ≠ nil) then
                if (u↑.genitore↑.sinistro = u) and (u↑.genitore ≠.valore = u↑.valore)
                    then begin
                        u↑.genitore↑.sinistro := nil;
                        dispose(u)
                    end
        end else begin
            if (u↑.sinistro ≠ nil) then CANCELLAFOGLIA(u↑.sinistro, T);
            if (u↑.destro ≠ nil) then CANCELLAFOGLIA(u↑.destro, T);
        end
    end;
```

Esercizio 4.7 (Inserisci foglie). Dato un albero binario, si vuole aggiungere ad ogni foglia un figlio destro. Si scriva una procedura Pascal ricorsiva di complessità ottima assumendo che l'albero binario sia realizzato con puntatori.

Valgono le stesse considerazioni dell'esercizio precedente riguardo alla limitazione inferiore e all'uso dei puntatori, ma stavolta l'algoritmo proposto è basato su una postvisita.

```
procedure INSERISCIFOGLIA(u: nodo; var T: binalbero);
  begin
    if (u↑.sinistro ≠ nil) then
      INSERISCIFOGLIA(u↑.sinistro, T);
    if (u↑.destro ≠ nil) then
      INSERISCIFOGLIA(u↑.destro, T);
    if (u↑.sinistro = nil) and (u↑.destro = nil) then begin
      new(u↑.sinistro);
      u↑.sinistro↑.sinistro := nil;
      u↑.sinistro↑.destro := nil;
      u↑.sinistro↑.genitore := u;
    end;
  end;
```

Esercizio 4.8 (Raggruppa le foglie con 0 e 1). Dato un albero binario le cui foglie contengono 0 od 1 e i cui nodi interni contengono solo 0, si vuole cambiare il contenuto delle foglie in modo che, visitandole da sinistra verso destra, si incontrino prima tutti gli 0 e poi tutti gli 1. Si scriva una procedura ricorsiva Pascal di complessità ottima assumendo che l'albero binario sia realizzato con puntatori.

Valgono le stesse considerazioni dell'esercizio precedente riguardo alla limitazione inferiore e all'uso dei puntatori. In particolare, per semplificare ulteriormente il codice della procedura, si sfrutta il fatto che, nella realizzazione Pascal, binalberi e nodi sono entrambi puntatori a celle e che il valore **nil** indica un nodo (e quindi un albero) vuoto. Questo permette di usare un solo parametro, per esempio un "binalbero" T, e di effettuare anche chiamate su sottoalberi vuoti. Per risolvere il problema, sono eseguite 2 visite. Nella prima visita si sommano i valori contenuti nelle foglie, registrando in un parametro *conta* quante sono le foglie ad 1. Nella seconda, le foglie sono rivisitate da destra verso sinistra, ponendo le prime *conta* ad 1. La chiamata è:

```
conta := 0;
CONTA1(T, conta);
SPOSTA1(T, conta)
```

```
procedure CONTA1 (var T: binalbero; var conta: integer);
  begin
    if T ≠ nil then begin
      conta := conta + T↑.valore;
      CONTA1(T↑.sinistro, conta);
      CONTA1(T↑.destro, conta)
    end;
  end;
```



```
procedure SPOSTA1(var T: binalbero; var conta: integer);
  begin
    if T ≠ nil then begin
      if (T↑.sinistro = nil) and (T↑.destro = nil) and (conta > 0)
        then begin
          T↑.valore := 1;
          conta := conta − 1;
        end
      else
        T↑.valore := 0;
      SPOSTA1(T↑.destro, conta);
      SPOSTA1(T↑.sinistro, conta)
    end;
  end;
```

Esercizio 4.9. Dato un albero binario, i cui nodi contengono elementi interi, si vuole modificarlo ottenendo l'albero inverso. Ogni nodo avrà cioè il figlio destro al posto di quello sinistro e viceversa. Si scrivano due procedure Pascal di complessità ottima per invertire un albero binario: una procedura deve usare gli operatori degli alberi binari, mentre l'altra lavora direttamente sulla realizzazione con puntatori.

Esercizio 4.10. Dato un albero binario i cui nodi contengono interi, si vuole aggiungere ad ogni foglia un figlio contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia. Si scriva una procedura Pascal ricorsiva di complessità ottima assumendo che l'albero sia realizzato con puntatori.

Esercizio 4.11 (Numero di foglie). Si dimostri per induzione che in un albero binario il numero delle foglie è uguale al numero dei nodi con due figli più uno.

Sia $T(n)$ un generico albero binario con $n \geq 1$ nodi e si indichi con $N2(T(n))$ il numero di nodi con due figli di $T(n)$ e con $F(T(n))$ il numero di foglie di $T(n)$. Si vuole dimostrare per induzione che $F(T(n)) = N2(T(n)) + 1$.

La base dell'induzione è vera per $n = 1$, perché in tal caso $F(T(1)) = 1$ e $N2(T(1)) = 0$. Si assuma la proprietà vera per $T(n)$. $T(n+1)$ può essere ottenuto da $T(n)$ aggiungendo un figlio o ad una foglia, o ad un nodo con un figlio. Nel primo caso, sia il numero delle foglie sia quello dei nodi con due figli non cambiano e quindi: $F(T(n+1)) = F(T(n)) = N2(T(n)) + 1 = N2(T(n+1)) + 1$. Nel secondo caso, sia il numero dei nodi con due figli sia quello delle foglie aumenta di uno e quindi: $F(T(n+1)) = F(T(n)) + 1 = N2(T(n)) + 1 + 1 = N2(T(n+1)) + 1$.

Esercizio 4.12 (Albero binario completo). Un albero binario completo di profondità k è un albero binario in cui tutti i nodi, tranne le foglie, hanno esattamente due figli, e tutte le foglie si trovano al livello k. Si dimostri per induzione che, in un albero binario completo di profondità k, il numero dei nodi è $2^{k+1} - 1$ ed il numero delle foglie è 2^k.

Esercizio 4.13 (Visite iterative). Utilizzando le pile, si scrivano tre procedure Pascal iterative di complessità ottima per effettuare, rispettivamente, le visite anticipata, differita e simmetrica di un albero binario.

Capitolo 5

Insiemi

Un insieme è una collezione (o famiglia) di elementi distinti (detti anche componenti o membri) dello stesso tipo. L'insieme è la struttura metematica fondamentale, e può essere descritto o elencandone tutti gli elementi o stabilendo una proprietà che ne caratterizzi gli elementi. A differenza delle liste, gli elementi non sono caratterizzati da una posizione, né sono ammesse più copie dello stesso elemento nel medesimo insieme.

Esempio 5.1 (Definizione di insiemi). $A = \{1, 5, 13, 7\}$, B = {studenti del Corso di Laurea in Informatica di Pisa immatricolati nel 1999}, C = {numeri reali compresi tra 0 e 1}. □

Il numero di elementi che compongono un insieme A è detto cardinalità (o dimensione) di A e si indica con $|A|$ (o anche con $\#A$). Se tale numero è finito, allora l'insieme ha cardinalità finita, altrimenti ha cardinalità infinita. Un insieme che non contiene alcun elemento è detto vuoto, e si indica con $\emptyset$.

Esempio 5.2 (Cardinalità di insiemi). La cardinalità dell'insieme A definito nell'Esempio 5.1 è 4, quella di B è finita, mentre quella di C è infinita. La cardinalità di $\emptyset$ è zero. □

La relazione fondamentale per gli insiemi è l'appartenenza di un elemento x ad un insieme A, indicata formalmente con $x \in A$. Da questa si deriva l'inclusione (o contenimento) tra due insiemi A e B: indicata con $A \subseteq B$, significa che ogni elemento x che appartiene ad A appartiene anche a B. Se $A \subseteq B$ e $B \subseteq A$, allora $A = B$, cioè A e B sono lo stesso insieme. Viceversa, le negazioni dell'appartenenza, dell'inclusione e dell'eguaglianza si indicano, rispettivamente, con $x \notin A$, $A \not\subseteq B$ e $A \neq B$. Le operazioni principali tra due insiemi A e B sono l'unione, l'intersezione e la differenza, indicate, rispettivamente, con $A \cup B$, $A \cap B$ e $A - B$. $A \cup B$ denota l'insieme contenente tutti gli elementi di A e tutti gli elementi di B. $A \cap B$ denota l'insieme contenente tutti e soli gli elementi che appartengono sia ad A che a B. Infine, $A - B$ denota l'insieme che contiene tutti e soli gli elementi che appartengono ad A ma non appartengono a B (per cui $A \subseteq B$ se e solo se $A - B = \emptyset$).

Esempio 5.3 (Operazioni tra insiemi). Siano $A = \{x, y, z\}$ e $B = \{y, w\}$. Allora sono vere: $x \in A$, $x \notin B$, $\{x, z\} \subseteq A$, $A = \{z, x, y\}$, $A \neq B$, $\emptyset \subseteq B$. Inoltre $A \cup B = \{x, y, w, z\}$, $A \cap B = \{y\}$, $A - B = \{z, x\}$. □

5.1 Specifica

Una specifica ragionevole per il tipo di dato insieme, oltre agli operatori, dall'ovvio significato, CREAINSIEME, INSIEMEVUOTO, APPARTIENE, UNIONE e INTERSEZIONE, può includere due ulteriori operatori di modifica: INSERISCI, per aggiungere un elemento ad un insieme e CANCELLA, per eliminare un elemento da un insieme.

CREAINSIEME:	() → insieme
INSIEMEVUOTO:	(insieme) → booleano
APPARTIENE:	(tipoelem, insieme) → booleano
INSERISCI:	(tipoelem, insieme) → insieme
CANCELLA:	(tipoelem, insieme) → insieme
UNIONE:	(insieme, insieme) → insieme
INTERSEZIONE:	(insieme, insieme) → insieme
DIFFERENZA:	(insieme, insieme) → insieme

Dalla specifica semantica degli operatori, illustrata di seguito, si nota che eventuali tentativi di inserimento di un elemento già presente o di cancellazione di un elemento non presente lasciano inalterato l'insieme.

CREAINSIEME $= A$
Post: $A = \emptyset$

INSIEMEVUOTO(A) $= b$
Post: b = vero, se $A = \emptyset$; b = falso, altrimenti

APPARTIENE(x, A) $= b$
Post: b = vero, se $x \in A$; b = falso, se $x \notin A$

INSERISCI(x, A) $= A'$
Post: $A' = A \cup \{x\}$ (se $x \in A$ allora $A' = A$)

CANCELLA(x, A) $= A'$
Post: $A' = A - \{x\}$ (se $x \notin A$ allora $A' = A$)

UNIONE(A, B) $= C$
Post: $C = A \cup B$

INTERSEZIONE(A, B) $= C$
Post: $C = A \cap B$

DIFFERENZA(A, B) $= C$
Post: $C = A - B$

Gli operatori INSERISCI e CANCELLA, pur essendo ridondanti, in quanto casi particolari di UNIONE e DIFFERENZA, sono molto utilizzati e di solito compaiono in tutte le specifiche degli insiemi. Va notato che in alcuni linguaggi di programmazione, incluso il Pascal, gli insiemi sono forniti direttamente come tipi primitivi. Per esempio, in Pascal c'è il tipo **set** e gli operatori di appartenenza, inclusione, unione e intersezione sono indicati, rispettivamente, da: **in**, <=, +, *. In luogo delle parentesi graffe, in Pascal sono usate le parentesi quadre, e l'insieme vuoto si indica con [].

Esempio 5.4 (Set). Dopo aver eseguito il seguente brano di programma Pascal, la variabile $c1$, di tipo set, vale [*cremino*, *gianduiotto*, *boero*]:

```
type
    cioccolatino = (cremino, gianduiotto, boero);
    confezione = set of cioccolatino;
var c1, c2: confezione;
        ⋮
    c1 := [boero, cremino];
    c2 := [];
    if (cremino in c1) and (c2 <= [gianduiotto])
        then c1 := c1 + [gianduiotto];                □
```

Purtroppo il Pascal può rappresentare soltanto insiemi con pochi elementi, il cui numero massimo dipende dal compilatore. Inoltre, la realizzazione primitiva non è efficiente, poiché si basa sulla seguente realizzazione con vettore booleano.

5.2 Realizzazione con vettore booleano

Si supponga di dover elaborare insiemi i cui elementi possono essere interi compresi tra 1 ed N. Un insieme A può essere rappresentato con un vettore booleano di N bit, in cui il k-esimo bit è vero se $k \in A$ ed è falso se $k \notin A$. In questo modo, è estremamente semplice verificare se un elemento k appartiene all'insieme, oppure inserire o cancellare un elemento k: basta accedere direttamente al k-esimo bit del vettore! Anche le operazioni di unione, intersezione e differenza di due insiemi si realizzano facilmente con operazioni logiche. Infatti, per l'unione basta porre a vero tutti i bit che sono veri in almeno uno dei due vettori (con un'operazione "or"), per l'intersezione tutti quelli veri in entrambi i vettori (con un "and") e per la differenza tutti quelli veri nel primo vettore e falsi nel secondo.

```
const N = ... {valore opportuno};
type tipoelem = 1..N;
    insieme = packed array[1..N] of boolean;

procedure CREAINSIEME(var A: insieme);
    var i: integer;
    begin
        for i := 1 to N do A[i] := false
    end;

function INSIEMEVUOTO(var A: insieme): boolean;
    var i: integer;
    begin
        i := 1;
        while (i < N) and not A[i] do i := i + 1;
        INSIEMEVUOTO := not A[i]
    end;

function APPARTIENE(i: tipoelem; var A: insieme): boolean;
    begin
        APPARTIENE := A[i]
    end;
```



```
procedure INSERISCI(i: tipoelem; var A: insieme);
    begin
        A[i] := true
    end;

procedure CANCELLA(i: tipoelem; var A: insieme);
    begin
        A[i] := false
    end;

procedure UNIONE(var A, B, C: insieme);
    var i: integer;
    begin
        for i := 1 to N do C[i] := A[i] or B[i]
    end;

procedure INTERSEZIONE(var A, B, C: insieme);
    var i: integer;
    begin
        for i := 1 to N do C[i] := A[i] and B[i]
    end;

procedure DIFFERENZA(var A, B, C: insieme);
    var i: integer;
    begin
        for i := 1 to N do C[i] := A[i] and not B[i]
    end;
```

La complessità di APPARTIENE, INSERISCI e CANCELLA è $O(1)$, mentre quella di CREAINSIEME, INSIEMEVUOTO, UNIONE, INTERSEZIONE e DIFFERENZA è $\Theta(N)$, perché occorre scandire l'intero vettore.

Questa realizzazione, pur avendo il pregio di essere estremamente semplice, presenta molti svantaggi. Innanzitutto vi è un notevole spreco di memoria, poiché ogni insieme richiede sempre $\Theta(N)$ spazio per la sua memorizzazione, indipendentemente dal numero di elementi effettivamente presenti nell'insieme. Inoltre, alcune operazioni sono troppo lente, perché richiedono $O(N)$ tempo, anche per insiemi con pochissimi elementi. Infine, la cardinalità dell'insieme non può superare la costante N.

5.3 Realizzazione con liste non ordinate

Si può rappresentare un insieme con una lista i cui elementi sono quelli dell'insieme. In questo modo, l'insieme non deve necessariamente essere composto da interi compresi tra 1 ed N. Inoltre, l'occupazione di memoria dell'insieme, anzichè essere sempre $\Theta(N)$, cresce linearmente col numero degli elementi effettivamente presenti nell'insieme.

Le operazioni su insiemi si riducono così a procedure che scandiscono liste, inserendovi o cancellandovi elementi. Per esempio, la procedura APPARTIENE è stata implicitamente già descritta in Pascal nel § 4.4 utilizzando gli operatori delle liste. La procedura CANCELLA è analoga alla APPARTIENE dove, dopo aver localizzato

l'elemento scandendo la lista, si procede alla sua eliminazione. La procedura INSERISCI è anch'essa semplice: dopo aver verificato con la APPARTIENE che l'elemento non sia già presente, si può inserirlo direttamente in testa alla lista come fosse una pila. La complessità di CREAINSIEME e INSIEMEVUOTO è O(1), mentre quella di APPARTIENE, INSERISCI e CANCELLA è $O(n)$, dove $n = |A|$, cioè il numero di elementi presenti nell'insieme. La complessità di INSERISCI è dovuta alla verifica preliminare che l'elemento da inserire non sia già presente nell'insieme. Qualora la non appartenenza all'insieme sia nota a priori, la complessità della INSERISCI può essere abbassata ad O(1).

Le procedure Pascal che realizzano UNIONE, INTERSEZIONE e DIFFERENZA sono abbastanza simili tra loro. Occorrono infatti tre variabili di tipo posizione per scandire le liste che rappresentano i tre insiemi A, B, e C. Per l'UNIONE, si inseriscono dapprima tutti gli elementi di B in C, mentre per l'INTERSEZIONE e la DIFFERENZA si parte con C vuoto. Successivamente, per ogni elemento di A, si verifica tramite la APPARTIENE se tale elemento compare anche in B oppure no e lo si inserisce in C nel primo caso (INTERSEZIONE) oppure nel secondo (DIFFERENZA e UNIONE). Per esempio, qui di seguito è riportata la procedura per la DIFFERENZA.

```
procedure DIFFERENZA(var A, B, C: insieme);
    var p, q, r: posizione;
    begin
        CREAINSIEME(C);
        p := PRIMOLISTA(A); q := PRIMOLISTA(B); r := PRIMOLISTA(C);
        while not FINELISTA(p, A) do begin
            if not APPARTIENE(LEGGILISTA(p, A), B)
                then INSLISTA(LEGGILISTA(p, A), r, C);
            p := SUCCLISTA(p, A);
        end;
    end;
```

Poiché APPARTIENE richiede una scansione di B ed è annidata con la scansione più esterna di A, le procedure UNIONE, INTERSEZIONE e DIFFERENZA richiedono tempo $O(nm)$, dove $n = |A|$ ed $m = |B|$, purché le liste siano realizzate con puntatori (o cursori).

5.4 Realizzazione con liste ordinate

Se sugli elementi dell'insieme è definita una relazione "$\leq$" di ordinamento totale, l'insieme stesso può essere rappresentato con una lista ordinata per valori crescenti degli elementi. In questo modo, la complessità degli operatori UNIONE, INTERSEZIONE e DIFFERENZA può essere abbassata a $O(n + m)$. La complessità di INSERISCI, invece, non può scendere sotto $O(n)$, perché quando si inserisce un elemento bisogna rispettare l'ordinamento della lista. Il tempo richiesto da APPARTIENE e CANCELLA resta ovviamente $O(n)$.

A titolo di esempio, l'operatore INTERSEZIONE può essere scritto nel modo seguente. Si usano ancora tre variabili di tipo posizione per scandire A, B e C, ma grazie all'ordinamento si può evitare di utilizzare l'APPARTIENE. Infatti, si considerano il primo elemento di A ed il primo elemento di B. Se sono uguali, si inserisce in C uno dei due e ci si posiziona sugli elementi successivi sia in A che in B. Se invece i due elementi sono diversi, si inserisce in C il minore e ci si posiziona sull'elemento

successivo solo nella lista del minore, senza avanzare nell'altra lista. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Si noti che, per mantenere anche C ordinata, l'inserzione in C va sempre effettuata in fondo.

```
procedure INTERSEZIONE(var A, B, C: insieme);
    var p, q, r: posizione;
    begin
        CREAINSIEME(C);
        p := PRIMOLISTA(A); q := PRIMOLISTA(B); r := PRIMOLISTA(C);
        while not FINELISTA(p, A) and not FINELISTA(q, B) do
            if LEGGILISTA(p, A) = LEGGILISTA(q, B) then
                begin
                    INSLISTA(LEGGILISTA(p, A), r, C);
                    p := SUCCLISTA(p, A);
                    q := SUCCLISTA(q, B);
                    r := SUCCLISTA(r, C)
                end
            else
                if LEGGILISTA(p, A) < LEGGILISTA(q, B)
                    then p := SUCCLISTA(p, A)
                    else q := SUCCLISTA(q, B)
    end;
```

Nella procedura INTERSEZIONE, sia ciascun elemento di A sia ciascun elemento di B è considerato al più una volta. La complessità di INTERSEZIONE è pertanto $O(n + m)$, purché le liste siano realizzate con puntatori (o cursori). Per quanto mostrato nell'Esempio I.13, la fusione di sequenze ordinate richiede $\Omega(n + m)$ tempo. La complessità di UNIONE, INTERSEZIONE e DIFFERENZA è pertanto ottima, relativamente alla realizzazione scelta. Infine, un ulteriore vantaggio della realizzazione con liste ordinate è quello di permettere di realizzare con complessità O(1) un operatore MIN per la ricerca del minimo:

MIN: (insieme) $\rightarrow$ tipoelem
MIN(A) = x
Pre: $A \neq \emptyset$
Post: $x \in A$ e $x \leq y$ per ogni $y \in A$

Realizzando le liste ordinate in modo da accedere direttamente anche all'ultimo elemento (come nel caso bidirezionale circolare) anche l'analogo operatore MAX di ricerca del massimo può essere realizzato con complessità O(1). Si noti che, realizzando l'insieme A con un vettore booleano o con una lista non ordinata, gli operatori MIN e MAX richiedono nel caso pessimo una scansione completa del vettore o della lista e la complessità risultante è, rispettivamente, $O(N)$ e $O(n)$.

5.5 Mfset e unione di insiemi disgiunti

Un'altra struttura che opera su un insieme è il cosiddetto "merge-find-set" (brevemente "mfset", ovvero "insieme fondi-trova"). L'mfset è una partizione di un insieme finito in sottoinsiemi disgiunti, detti comunemente componenti (o parti). Le operazioni ammesse permettono di stabilire a quale componente appartiene un generico

elemento e di unire due componenti distinte in una sola componente, distruggendo le due vecchie componenti ma lasciando inalterate le componenti rimanenti.

La diversità rispetto all'usuale tipo di dato insieme consiste nel fatto che non sono previste cancellazioni o inserzioni di elementi, eccettuato gli inserimenti "multipli" effettuati dall'operatore CREAMFSET per inizializzare l'mfset ad una partizione formata da tante componenti contenenti ciascuna uno ed un solo elemento. Inoltre, l'operatore TROVA restituisce l'identificatore di una componente anziché un booleano, mentre l'operatore FONDI effettua l'unione di due insiemi (componenti) disgiunti. Poiché FONDI unisce due vecchie componenti senza però modificare le altre, dopo la sua esecuzione il numero totale di componenti è diminuito di uno. La specifica dell'mfset è la seguente.

CREAMFSET:	(insieme) $\rightarrow$ mfset
TROVA:	(tipoelem, mfset) $\rightarrow$ componente
FONDI:	(tipoelem, tipoelem, mfset) $\rightarrow$ mfset

CREAMFSET(A) = S
Post: S è una famiglia di $n = |A|$ componenti $C_1, \ldots, C_n$, ciascuna delle quali contiene uno e un solo elemento di A e tali che $\bigcup_{1 \leq i \leq n} C_i = A$

TROVA(x, S) = C
Pre: x appartiene ad una componente di S
Post: C è l'identificatore della componente cui x appartiene

FONDI(x, y, S) = S'
Pre: TROVA(x, S) $\neq$ TROVA(y, S), cioè x e y appartengono a componenti distinte di S
Post: S' è formato da tutte le componenti di S che non contengono x o y, e da una nuova componente data dall'unione delle due componenti contenenti x e y

Esempio 5.5 (Mfset). Sia dato $A = \{1, 2, 3, 4\}$. CREAMFSET restituisce $S = \{\{1\}, \{2\}, \{3\}, \{4\}\}$. Poiché TROVA(1, S) $\neq$ TROVA(3, S), eseguendo FONDI(1, 3, S) si ottiene $S = \{\{1, 3\}, \{2\}, \{4\}\}$. Adesso, TROVA(4, S) $\neq$ TROVA(2, S) ed eseguendo FONDI(4, 2, S), si ha $S = \{\{1, 3\}, \{2, 4\}\}$. Essendo TROVA(3, S) $\neq$ TROVA(4, S), l'esecuzione di FONDI(3, 4) dà per risultato $S = \{\{1, 3, 2, 4\}\}$. D'ora in poi, non sono più possibili ulteriori fusioni. □

Una realizzazione efficiente per l'mfset prevede che gli elementi di ciascuna componente siano ospitati nei nodi di un albero radicato (non necessariamente ordinato) così che la partizione in componenti sia rappresentata da un insieme di alberi radicati (detta foresta). Ciascuna componente è identificata dalla radice dell'albero corrispondente. In questo modo, l'operatore TROVA si riduce a restituire la radice dell'albero, mentre FONDI può essere realizzato combinando i due alberi in un unico albero.

Esempio 5.6 (Realizzazione con foresta). Sia dato un mfset S di 7 componenti, ciascuna contenente il solo intero i, $1 \leq i \leq 7$. S è rappresentato con la foresta di Fig. 5.1a). L'effetto su S dell'esecuzione della sequenza di operazioni FONDI(1, 2, S), FONDI(1, 3, S), FONDI(5, 6, S), FONDI(6, 3, S), FONDI(4, 7, S) e FONDI(3, 7, S) è mostrato nelle Figg. 5.1(b)-(g). □

Se è possibile accedere direttamente ai nodi contenenti gli elementi, l'operazione TROVA si realizza "salendo" attraverso i padri lungo un percorso da un generico nodo

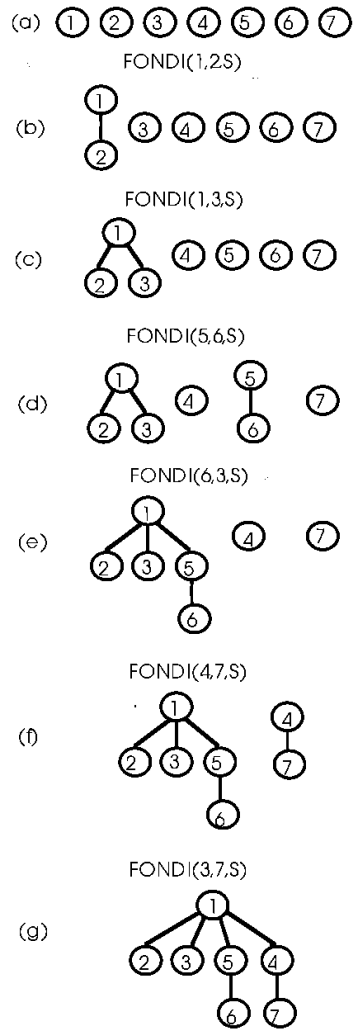


Figura 5.1: Sequenza di fusioni in un mfset.

fino alla radice. La FONDI si realizza in modo analogo, imponendo inoltre che una delle due radici divenga un nuovo figlio dell'altra.

Poiché TROVA e FONDI prevedono di risalire sugli alberi, la complessità può essere abbassata mantenendo il più possibile ridotto il livello massimo delle foglie di ciascun albero. A tal fine, in una FONDI è opportuno scegliere sempre come radice della nuova componente quella relativa alla componente più "grande" (quella dell'albero col maggior numero di nodi). In altri termini, si impone che la radice della componente più "piccola" (quella col minor numero di nodi) divenga un nuovo figlio della radice di quella più grande.

Per giustificare la bontà di questa scelta, si consideri la sequenza di fusioni nelle quali un generico elemento x può essere coinvolto e si analizzi di quanto può aumentare complessivamente il suo livello. Se in una fusione x appartiene all'albero più grande, allora il suo livello non cambia, perché la radice resta la stessa dopo la fusione. Se invece nella fusione x appartiene all'albero più piccolo, allora il suo livello aumenta di uno, perché la sua vecchia radice diviene un nuovo figlio della radice dell'albero più grande. Ma quando l'albero più piccolo che contiene x viene fuso con quello più grande, il numero di nodi del nuovo albero è almeno il doppio del numero di nodi dell'albero cui x apparteneva. Poiché inizialmente vi sono $n = |A|$ componenti, ciascuna formata da un solo elemento, le componenti più piccole alle quali x via via appartiene possono essere coinvolte in $O(\log n)$ fusioni. Siccome ad ogni fusione il livello delle foglie della componente più piccola aumenta di uno e x inizialmente è a livello zero, il livello del nodo contenente x risulta essere $O(\log n)$ nel caso pessimo. La complessità di TROVA e FONDI è pertanto $O(\log n)$.

Si supponga che l'insieme A sia formato da tutti gli interi compresi tra 1 ed n e sia rappresentato per semplicità con una variabile contenente il valore n. La foresta di alberi radicati che rappresenta l'mfset può allora essere realizzata in Pascal con due vettori: un vettore dei padri e un vettore "dimensione" che per ogni radice contiene il numero di nodi dell'albero.

```
type
    tipoelem = 1..maxlung; insieme = 1..maxlung;
    componente = 1..maxlung;
    mfset = array[1..maxlung] of record
        padre: 0..maxlung;
        dimensione: 1..maxlung
    end;
```

La procedura CREAMFSET inizializza la foresta ad $A = n$ alberi di un nodo ciascuno. La funzione TROVA, dato un elemento (nodo), risale da esso fino alla radice che viene restituita come identificativo della componente. La FONDI richiama due volte la TROVA e poi utilizza il vettore *dimensione* per stabilire qual è l'albero più piccolo la cui radice diventa figlio della radice dell'albero più grande. Durante la fusione, FONDI aggiorna il campo "dimensione" della radice del sottoalbero più grande, sommandoci la dimensione di quello più piccolo.

```
procedure CREAMFSET(var A: insieme; var S: mfset);
    var i: integer;
    begin
        for i := 1 to A do begin
            S[i].padre := 0; S[i].dimensione := 1
        end
    end;
```



```
function TROVA(x: tipoelem; var S: mfset): componente;
    begin
        if S[x].padre = 0
            then TROVA := x
            else TROVA := TROVA(S[x].padre, S)
    end;

procedure FONDI(x, y: tipoelem; var S: mfset);
    var i, j: 1..maxlung;
    begin
        i := TROVA(x, S);
        j := TROVA(y, S);
        if i ≠ j then
            if S[i].dimensione < S[j].dimensione then begin
                S[i].padre := j;
                S[j].dimensione := S[j].dimensione + S[i].dimensione
            end else begin
                S[j].padre := i;
                S[i].dimensione := S[i].dimensione + S[j].dimensione
            end
    end;
```

Come discusso precedentemente, la complessità di TROVA e FONDI è $O(\log n)$, mentre quella di CREAMFSET è $\Theta(n)$.

Un accorgimento che viene usato per abbassare la complessità media di TROVA e FONDI consiste nel modificare la TROVA in modo che alteri l'albero durante la sua esecuzione, così da rendere più vantaggiose le esecuzioni successive di TROVA e FONDI. La tecnica usata si chiama compressione dei percorsi, e consiste nel rendere figlio della radice ogni nodo che viene incontrato da TROVA nel percorso di risalita dal generico nodo x alla radice. Per far questo, la chiamata ricorsiva all'interno di TROVA è modificata introducendo un assegnamento al padre del nodo x. In questo modo, durante l'esecuzione, restano "sospesi" gli assegnamenti sui padri dei nodi nel percorso di risalita verso la radice. Quando con la chiamata ricorsiva più interna si giunge alla radice, la ricorsione di TROVA termina restituendo la radice stessa. La radice viene assegnata "a ritroso" ai padri in tutti gli assegnamenti "sospesi", via via che si richiudono tutte le chiamate ricorsive. È importante notare che il numero di operazioni della nuova versione di TROVA aumenta solo per un fattore costante rispetto alla vecchia versione e che pertanto la sua complessità nel caso pessimo non aumenta.

```
function TROVA(x: tipoelem; var S: mfset): componente;
    begin
        if S[x].padre = 0 then
            TROVA := x
        else begin
            S[x].padre := TROVA(S[x].padre, S);
            TROVA := S[x].padre
        end
    end;
```

Esempio 5.7 (Compressione dei percorsi). L'effetto dell'esecuzione di TROVA con la compressione dei percorsi è mostrato nella Fig. 5.2. □

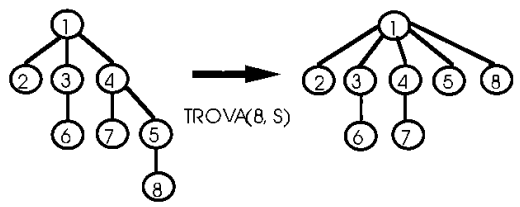


Figura 5.2: Compressione dei percorsi.

Il vantaggio introdotto dalla compressione dei percorsi si ha quando viene effettuata una sequenza di operazioni formata, oltre che da una CREAMFSET, da m TROVA ed m FONDI, con $m \geq n$. Senza la compressione dei percorsi, la complessità dovuta all'esecuzione della sequenza è ovviamente $O(n + m \log n)$, mentre con la compressione dei percorsi è possibile dimostrare che tale costo scende a $O(n + m\alpha(m, n))$, dove $\alpha(m, n)$ è la funzione inversa della funzione di Ackermann.

$\alpha(m, n)$ è una funzione che cresce lentissimamente, tanto che $\alpha(m, n) \leq 4$ per ogni m ed n rappresentabile con un calcolatore elettronico! Ne segue che in media ciascuna delle m operazioni TROVA e FONDI della sequenza ha complessità "ammortizzata" $O((n + m\alpha(m, n))/m)$, che è pressoché $O(1)$ per ogni valore "realistico" di n ed m.

5.6 Esercizi

Esercizio 5.1 (Intersezione di liste non ordinate). Utilizzando gli operatori delle liste, si scriva in Pascal la procedura per INTERSEZIONE, assumendo che gli insiemi siano realizzati con liste non ordinate.

Esercizio 5.2 (Unione di liste non ordinate)). Utilizzando gli operatori delle liste, si scriva in Pascal la procedura per UNIONE, assumendo che gli insiemi siano realizzati con liste non ordinate.

Esercizio 5.3 (Unione di liste ordinate). Utilizzando gli operatori delle liste, si scriva in Pascal la procedura di complessità ottima per UNIONE, assumendo che gli insiemi siano realizzati con liste ordinate.

Nella procedura UNIONE, si usano tre variabili di tipo posizione per scandire A, B e C. Come nella procedura INTERSEZIONE, si considerano il primo elemento di A ed il primo elemento di B. Se sono uguali, si inserisce in C uno dei due e ci si posiziona sugli elementi successivi sia in A che in B. Se invece i due elementi sono diversi, si inserisce in C il minore e ci si posiziona sull'elemento successivo solo nella lista del minore, senza avanzare nell'altra lista. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Se una lista tra A e B non è stata esaurita, si prosegue inserendo in C tutti gli elementi rimasti nella lista che non era stata esaurita. Per mantenere C ordinata, l'inserzione in C è sempre effettuata in fondo. La complessità è $O(n + m)$, come per INTERSEZIONE.


```
procedure UNIONE(var A, B, C: insieme);
   var p, q, r: posizione;
   begin
      CREAINSIEME(C);
      p := PRIMOLISTA(A);
      q := PRIMOLISTA(B);
      r := PRIMOLISTA(C);
      while not FINELISTA(p, A) and not FINELISTA(q, B) do
         if LEGGILISTA(p, A) ≤ LEGGILISTA(q, B) then
            begin
               INSLISTA(LEGGILISTA(p, A), r, C);
               if LEGGILISTA(p, A) = LEGGILISTA(q, B) then
               q := SUCCLISTA(q, B);
               p := SUCCLISTA(p, A);
               r := SUCCLISTA(r, C)
            end
         else
            begin
               INSLISTA(LEGGILISTA(q, B), r, C);
               q := SUCCLISTA(q, B);
               r := SUCCLISTA(r, C)
            end;
      while not FINELISTA(p, A) do
         begin
            INSLISTA(LEGGILISTA(p, A), r, C);
            p := SUCCLISTA(p, A);
            r := SUCCLISTA(r, C)
         end;
      while not FINELISTA(q, B) do
         begin
            INSLISTA(LEGGILISTA(q, B), r, C);
            q := SUCCLISTA(q, B);
            r := SUCCLISTA(r, C)
         end;
   end;
```

Esercizio 5.4 (Differenza con liste ordinate). Utilizzando gli operatori delle liste, si scriva in Pascal la procedura di complessità ottima per DIFFERENZA, assumendo che gli insiemi siano realizzati con liste ordinate.

Come nella procedura INTERSEZIONE, si considerano il primo elemento di A ed il primo elemento di B. Se quello di A è minore, allora lo si inserisce in C e ci si posiziona sull'elemento successivo di A. Se quello di B è minore, allora si avanza di una posizione solo in B, mentre se i due elementi sono uguali, si avanza di una posizione sia in A che in B. In entrambi i casi non si inserisce alcun elemento in C. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Se la lista A non è stata esaurita, si prosegue inserendo in C tutti gli elementi rimasti in A. La complessità è $O(n + m)$, come per INTERSEZIONE.

```
procedure DIFFERENZA(var A, B, C: insieme);
    var p, q, r: posizione;
    begin
        CREAINSIEME(C);
        p := PRIMOLISTA(A);
        q := PRIMOLISTA(B);
        r := PRIMOLISTA(C);
        while not FINELISTA(p, A) and not FINELISTA(q, B) do
            if LEGGILISTA(p, A) < LEGGILISTA(q, B) then begin
                INSLISTA(LEGGILISTA(p, A), r, C);
                p := SUCCLISTA(p, A);
                r := SUCCLISTA(r, C)
            end else begin
                if LEGGILISTA(p, A) = LEGGILISTA(q, B) then
                    p := SUCCLISTA(p, A);
                q := SUCCLISTA(q, B);
            end;
        while not FINELISTA(p, A) do begin
            INSLISTA(LEGGILISTA(p, A), r, C);
            p := SUCCLISTA(p, A);
            r := SUCCLISTA(r, C)
        end;
    end;
```

Esercizio 5.5 (Differenza simmetrica). La differenza simmetrica $A \Delta B$ di due insiemi A e B, è data da tutti gli elementi che stanno nell'unione $A \cup B$ ma non nell'intersezione $A \cap B$. Si scrivano tre procedure Pascal per il nuovo operatore DIFFERENZA-SIMMETRICA, assumendo realizzazioni con vettori booleani, liste non ordinate, e liste ordinate, rispettivamente.

Esercizio 5.6 (Dimensione). Si provi che $S[x].dimensione$ contiene il numero di nodi del sottoalbero radicato in x per ogni nodo, se non si usa la compressione dei percorsi, ma solo per ogni radice x di albero, se si usa la compressione dei percorsi.

Esercizio 5.7 (Compressione senza ricorsione). Si scriva una versione iterativa della funzione Pascal TROVA con compressione dei percorsi senza usare una pila e senza aumentarne la complessità.

Capitolo 6

Dizionari

Il dizionario è un caso particolare di insieme, in cui è possibile verificare l'appartenenza di un elemento, cancellare un elemento e inserire un nuovo elemento. Gli elementi del dizionario sono detti comunemente chiavi.

La specifica è identica a quella del tipo di dato insieme, in cui le uniche operazioni ammesse, oltre alla creazione, sono APPARTIENE, INSERISCI, e CANCELLA. Ogni realizzazione per l'insieme va ovviamente bene anche per il dizionario, ma ci sono realizzazioni particolarmente efficienti mediante vettori ordinati e tabelle hash.

6.1 Realizzazione con vettore ordinato

Si utilizza un vettore con un cursore all'ultima posizione occupata e le chiavi sono memorizzate nel vettore in posizioni contigue e in ordine crescente a partire dalla prima posizione.

Per verificare l'appartenenza di una chiave k, si trae vantaggio dall'ordinamento nel vettore. Si confronta il valore della chiave con il valore v che occupa la posizione centrale del vettore. Se $k = v$, allora la chiave è stata trovata. Se invece $k < v$ (o $k > v$) allora la chiave va ricercata nella prima (seconda) metà del vettore, che contiene solo chiavi minori (maggiori) di k, senza più considerare l'altra metà del vettore. Questo procedimento, detto di "ricerca binaria" (o anche di ricerca dicotomica o logaritmica), viene riapplicato ricorsivamente sulla metà del vettore così determinata, fino ad individuare k o a stabilire che k non appartiene all'insieme perché sono stati esauriti gli elementi da confrontare.

Nella funzione ricorsiva Pascal RICBIN per la ricerca binaria, si utilizzano quattro parametri: il vettore V che contiene le chiavi, la chiave k da ricercare e due cursori i e j che individuano la porzione $V[i..j]$ del vettore sulla quale ricercare k. La posizione centrale m si ricava sommando all'estremo sinistro i la lunghezza dell'intervallo (che è $j - i$) dimezzata, ottenendo $m = i + \lfloor (j - i)/2 \rfloor = \lfloor (i + j)/2 \rfloor$. Se k è uguale a $V[m]$, allora la ricerca termina con risultato "vero". Altrimenti, viene effettuata una chiamata ricorsiva su una delle due metà, $V[i..m-1]$ o $V[m+1..j]$, a seconda che k sia minore o maggiore di $V[m]$, rispettivamente. Nel far ciò, il cursore i può "scavalcare" il cursore j. Infatti, quando $i = j$, risulta $m = \lfloor (i + i)/2 \rfloor = i = j$. Pertanto, se $k \neq V[m]$, la nuova chiamata ricorsiva avviene o su $V[i..i-1]$ o su $V[j+1..j]$. Questo significa che la chiave k non appare nel dizionario e quindi la condizione di chiusura "$i > j$" fa terminare la ricorsione restituendo risultato "falso". Poiché la mancanza di chiavi nel dizionario è indicata dal valore 0 nel cursore "ultimo", una eventuale

ricerca binaria sul vettore vuoto avviene con indici $i = 1$ e $j = 0$ e quindi restituisce correttamente "falso".

```
type
    dizionario = record
        chiavi: array [1..maxlung] of tipoelem;
        ultimo: 0..maxlung
    end;

function APPARTIENE(k: tipoelem; var D: dizionario): boolean;
    begin
        APPARTIENE := RICBIN(D.chiavi, k, 1, D.ultimo)
    end;

function RICBIN(var V: vettore; k: tipoelem; i, j: integer): boolean;
    var m: integer;
    begin
        if i > j
            then RICBIN := false
        else begin
            m := (i + j) div 2;
            if k = V[m]
                then RICBIN := true
                else if k < V[m]
                    then RICBIN := RICBIN(V, k, i, m − 1)
                    else RICBIN := RICBIN(V, k, m + 1, j)
        end
    end;
```

Esempio 6.1 (Ricerca binaria). Applicando la funzione RICBIN al vettore ordinato di 12 chiavi intere $V = 1, 5, 7, 8, 13, 17, 21, 27, 33, 37, 40, 42$ con $k = 13$ si ottiene:

CHIAMATA	i	j	m	$V[m]$
1	1	12	6	17
2	1	5	3	7
3	4	5	4	8
4	5	5	5	13

□

Per analizzare la complessità di RICBIN, è conveniente contare il numero di operazioni "dominanti" effettuate. Sia n il numero di chiavi memorizzate nel dizionario e sia $C(n)$ il numero di confronti tra la chiave k e le chiavi presenti nel vettore. Poiché ad ogni chiamata ricorsiva sono effettuati al più 2 confronti su k (quelli "$k = V[m]$" e "$k < V[m]$"), assumendo per semplicità $n = 2^p$, si ha:

$$\begin{aligned} C(n) &\leq C(n/2) + 2 \\ &\leq C(n/2^2) + 2 \cdot 2 \\ &\vdots \\ &\leq C(n/2^p) + p \cdot 2 = C(1) + p \cdot 2 \\ &\leq 2 + p \cdot 2 = 2(p+1) \end{aligned}$$

Essendo $p = \log n$, risulta che $C(n)$ è $O(\log n)$. Pertanto, realizzando il dizionario con un vettore ordinato ed utilizzando la ricerca binaria la complessità dell'operatore APPARTIENE scende da $O(n)$ ad $O(\log n)$.

La complessità di APPARTIENE può essere ulteriormente abbassata tenendo conto dei valori delle chiavi memorizzate nel vettore e della loro distribuzione di probabilità.

Esempio 6.2 (Elenco telefonico). Volendo ricercare il cognome Cherubini nell'elenco telefonico, non si apre l'elenco a metà, ma all'incirca ad un quarto del numero delle pagine. □

Siano le n chiavi presenti nel vettore numeriche ed uniformemente distribuite nell'intervallo $[k_{\text{MIN}}, k_{\text{MAX}}]$. Dovendo ricercare la chiave k nel vettore $V[1..n]$, anziché provare nella posizione centrale è ragionevole tentare in quella più vicina a:

$$\frac{n(k - k_{\text{MIN}})}{(k_{\text{MAX}} - k_{\text{MIN}})}.$$

Infatti, $k - k_{\text{MIN}}$ fornisce lo scarto tra i valori della chiave da ricercare e di quella più piccola, mentre $k_{\text{MAX}} - k_{\text{MIN}}$ dà lo scarto tra i valori della chiave più grande e quella più piccola. Il loro rapporto indica di quanto il valore di k si discosta dai valori estremi. Se $k = k_{\text{MIN}}$, allora il rapporto è 0 e conviene cercare nella prima posizione del vettore, mentre se $k = k_{\text{MAX}}$, il rapporto è 1 e conviene cercare nell'ultima posizione del vettore (la ricerca binaria, ignorando il valore di k, assume tale rapporto 1/2, e tenta in una posizione centrale). Il metodo di ricerca risultante è detto di "interpolazione". La procedura Pascal di ricerca binaria RICBIN può essere trasformata in una di interpolazione, INTERPOL, semplicemente sostituendo l'istruzione

$m := (i + j)$ **div** 2

con

$m := i + (k - V[i]) * (j - i)$ **div** $(V[j] - V[i])$.

Infatti la ricerca viene effettuata in generale sulla porzione $V[i..j]$ del vettore, che contiene la più piccola chiave in $V[i]$ e la più grande in $V[j]$. La formula per m si ricava da quella della ricerca binaria, che era $i + \lfloor (j-i)/2 \rfloor$, sostituendo $(k - V[i])/(V[j] - V[i])$ ad 1/2.

Esempio 6.3 (Interpolazione). Applicando la procedura INTERPOL al vettore dell'Esempio 6.1, sempre con $k = 13$, si ottiene:

CHIAMATA	i	j	$V[i]$	$V[j]$	m	$V[m]$
1	1	12	1	42	4	8
2	5	12	13	42	5	13

Si noti come, rispetto alla ricerca binaria, siano sufficienti 2 sole chiamate ricorsive anziché 4. □

È possibile dimostrare che, sotto l'ipotesi di uniforme distribuzione delle chiavi, la complessità di INTERPOL è $O(\log \log n)$. La dimostrazione è piuttosto complicata e non viene qui riportata. La funzione $\log \log n$ cresce molto lentamente (per esempio, vale meno di 5 per n uguale a un miliardo), ma è paragonabile a $\log n$ per n piccolo. Pertanto se ci sono poche chiavi oppure se le chiavi non sono uniformemente distribuite è più conveniente usare la ricerca binaria.

Mentre la complessità di APPARTIENE risulta notevolmente abbassata realizzando il dizionario con un vettore ordinato ed utilizzando un metodo di ricerca binaria (o di interpolazione), la complessità degli operatori INSERISCI e CANCELLA resta, purtroppo, $O(n)$. La realizzazione con vettore ordinato risulta pertanto conveniente quando sul dizionario si effettuano principalmente interrogazioni sulle chiavi memorizzate, mentre inserimenti di nuove chiavi e cancellazioni di chiavi già presenti sono eventi rarissimi.

6.2 Realizzazione con tabella hash

La realizzazione del dizionario con tabelle hash si basa sul concetto di ricavare direttamente dal valore della chiave la posizione che la chiave stessa dovrebbe occupare in un vettore. Con questa realizzazione, ciascuna operazione, pur avendo complessità $O(n)$ nel caso pessimo, richiede tempo $O(1)$ nel caso medio. Sia K l'insieme di tutte le possibili chiavi distinte e si memorizzi il dizionario in un vettore V di dimensione m. La soluzione ideale sarebbe quella di possedere una funzione d'accesso

$$H : K \rightarrow \{1, \ldots, m\}$$

che permetta di ricavare la posizione $H(k)$ della chiave k nel vettore V in modo che per ogni $k_1 \in K$ e $k_2 \in K$, con $k_1 \neq k_2$, risulti $H(k_1) \neq H(k_2)$. Per garantire tale biunivocità, ovviamente, m non può essere più piccolo di $|K|$.

Per esempio, utilizzando un vettore con $m = |K|$ posizioni, si potrebbe accedere direttamente alla posizione contenente la chiave in tempo $O(1)$. Se $|K|$ è grande, purtroppo, questo metodo è improponibile perché richiede uno spreco enorme di memoria.

Esempio 6.4 (Tabella dei simboli). Il compilatore del linguaggio Pascal utilizza un dizionario, detto tabella dei simboli, per memorizzare tutti gli identificatori (nomi di variabili, costanti, procedure, ecc.) utilizzati in un programma Pascal. Questi identificatori sono inseriti nel dizionario quando sono compilate le "dichiarazioni", mentre la loro appartenenza è verificata quando sono compilate le "istruzioni eseguibili". Assumendo, come nella maggior parte dei compilatori, $K = \{$stringhe alfanumeriche di al più 8 caratteri di cui il primo è alfabetico$\}$, si ottiene: $|K| = 26 \cdot 36^7 > 2 \cdot 10^{12}$. È chiaramente impensabile utilizzare un vettore con oltre duemila miliardi di posizioni per un programma che potrà contenere, al più, qualche centinaio di identificatori! □

Per evitare inutili sprechi di memoria, la dimensione m del vettore va scelta in base al numero di chiavi attese, cioè che ci si aspetta possano essere effettivamente memorizzate contemporaneamente, e non in base a quello di tutte le chiavi possibili, la maggior parte delle quali potrebbe non apparire mai nel dizionario. Occorre quindi rinunciare alla biunivocità della funzione H ammettendo che chiavi diverse possano avere lo stesso indirizzo nel vettore, cioè che $H(k_1) = H(k_2)$ anche se $k_1 \neq k_2$.

In tal caso, la soluzione estrema che minimizza l'uso della memoria sembra quella in cui c'è un unico indirizzo $H(k)$ per ogni possibile chiave k, per esempio mantenendo tutte le chiavi in un'unica lista. In questo modo, la memoria richiesta è linearmente proporzionale al numero n di chiavi presenti nel dizionario, ma anche il tempo per le operazioni cresce con un inaccettabile $O(n)$. Occorre pertanto una soluzione di compromesso, con m maggiore di 1 ma molto minore di $|K|$, che mantenga bassi sia il tempo di esecuzione delle operazioni (possibilmente $O(1)$, come nel caso $m = |K|$), sia l'occupazione di memoria (possibilmente $O(n)$ come nel caso $m = 1$).

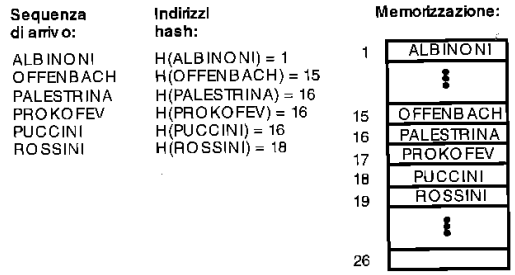


Figura 6.1: Un (cattivo) esempio di tabella hash.

Rinunciando alla biunivocità di H, sorge il nuovo problema di scegliere le posizioni per memorizzare due chiavi k_1 e k_2 che collidono, cioè con $H(k_1) = H(k_2)$, qualora siano entrambe presenti contemporaneamente nel vettore. Un esempio è utile per illustrare le principali problematiche che si presentano.

Esempio 6.5 (Musicisti). Sia $K = \{$cognomi di musicisti$\}$ e si assuma $m = 26$. Si consideri la funzione $H(k) = h$, $1 \leq h \leq 26$, se il carattere iniziale di k è l'h-esima lettera dell'alfabeto inglese. La funzione H non è biunivoca, poiché, p.e., $H(\text{BACH}) = H(\text{BEETHOVEN}) = H(\text{BRAHMS}) = 2$. Sia V inizialmente vuoto e si supponga che le chiavi ALBINONI, OFFENBACH e PALESTRINA si presentino, in quest'ordine, per essere inserite. Poiché $H(\text{ALBINONI}) = 1$, $H(\text{OFFENBACH}) = 15$ e $H(\text{PALESTRINA}) = 16$, le tre chiavi saranno inserite, rispettivamente, nelle posizioni 1, 15, e 16. Se successivamente si presenta PROKOFEV, poiché $H(\text{PROKOFEV}) = 16$ e la posizione 16 è occupata da PALESTRINA, nasce una collisione tra le due chiavi: dove sistemare PROKOFEV? Una scelta ragionevole è quella di cercare la prima posizione libera dopo la 16, in questo caso la 17, e lì memorizzare PROKOFEV. Se poi arriva PUCCINI, un trattamento analogo porta a riservargli la posizione 18. Un successivo arrivo di ROSSINI, per il quale $H(\text{ROSSINI}) = 18$, provoca la memorizzazione del nuovo venuto nella posizione 19 (si veda la Fig. 6.1). La situazione è qui aggravata dal fatto che ROSSINI ha trovato la sua posizione abusivamente occupata da PUCCINI, cioè da una chiave per la quale la funzione H dà un risultato diverso. Inoltre, ROSSINI stesso va ad occupare una posizione che non gli compete. Se ora si vuole verificare l'appartenenza di una chiave già presente nel dizionario, occorre procedere attraverso le stesse posizioni per le quali si è passati durante la sua inserzione: così se si vuole ricercare PUCCINI occorrono ben tre accessi al vettore V, mentre per ricercare PALESTRINA ne basta uno. Questa non uniformità nel numero di accessi necessari a reperire una chiave è dovuta alla presenza di un agglomerato: benchè V contenga solo 6 chiavi su 26 memorizzabili, ben 5 di queste si trovano in posizioni contigue e chiavi k diverse con indirizzo $H(k)$ compreso tra 15 e 19 devono procedere attraverso posizioni comuni per essere reperite. Ovviamente, dopo aver riempito le 26 posizioni disponibili, una eventuale 27-esima chiave non potrà più essere inserita. □

L'Esempio 6.5 è molto istruttivo e focalizza le problematiche che sorgono qualora si voglia realizzare un dizionario con una tabella hash.

Innanzitutto, la funzione H dell'Esempio 6.5 non è una buona funzione di accesso (anzi, è proprio da sconsigliare), sia perché l'indirizzo $H(k)$ dipende pesantemente da

configurazioni di caratteri (in questo caso il primo) che appaiono nella chiave, sia perché non vi è alcuna garanzia che distribuisca uniformemente le chiavi nel vettore. Infatti, ognuna delle $|K|$ possibili chiavi dovrebbe avere la stessa probabilità $1/m$ che il suo indirizzo $H(k)$ sia una qualsiasi delle m posizioni del vettore, ed è ben noto che la distribuzione della lettera iniziale dei cognomi di persone non è affatto uniforme.

In secondo luogo, per inserire chiavi che hanno trovato la loro posizione occupata, occorre assolutamente evitare il formarsi di agglomerati. Nell'Esempio 6.5, 5 delle 6 chiavi complessivamente presenti nel dizionario stanno in posizioni contigue. Ciò è dovuto alla inadeguatezza sia della funzione H, sia del metodo usato per esplorare il vettore (cioè tentare nella posizione successiva, poi in quella ancora successiva, e così via). Se le 6 chiavi fossero state inserite seguendo un criterio uniforme, essendo $\lfloor 26/6 \rfloor = 4$, ci si aspetterebbe in media di trovare nel vettore all'incirca una posizione occupata seguita da tre libere. Purtroppo, la presenza di agglomerati tende a farli espandere ancora di più. Infatti, se una posizione vuota è preceduta da j occupate, allora la probabilità che tale posizione vuota sia la prossima ad essere riempita è $(j+1)/m$, contro una probabilità di $1/m$ nel caso che la posizione precedente fosse anch'essa libera! In particolare, quando il vettore è quasi completamente occupato, diviene pressoché impossibile evitare il formarsi di agglomerati ed il tempo per inserimenti e ricerche di chiavi tende a crescere linearmente con la dimensione m del vettore. Inoltre, se il vettore si riempie completamente, non è ovviamente più possibile alcuna ulteriore inserzione.

Ricapitolando, per realizzare efficientemente un dizionario con una tabella hash:

1. occorre una funzione H, detta appunto funzione hash, che sia calcolabile velocemente (in tempo $O(1)$) e che distribuisca le chiavi uniformemente nel vettore V, al fine di ridurre il numero di collisioni tra chiavi diverse che hanno lo stesso indirizzo hash;
2. occorre un metodo di scansione, per reperire chiavi che hanno trovato la loro posizione occupata, che permetta di esplorare tutto il vettore V e non provochi la formazione di agglomerati di chiavi;
3. la dimensione m del vettore V deve essere una sovrastima (di solito il doppio) del numero di chiavi attese, onde evitare di riempire V completamente.

6.2.1 Funzioni hash

Trovare una buona funzione hash è, in un certo senso, un problema opposto a quello della ricerca binaria. Nella ricerca binaria si vuole ricavare l'output (un booleano) tenendo conto di una relazione sulla struttura dell'input (l'ordine degli elementi). Con la funzione hash, invece, si vuole ricavare l'output (un numero intero) che sia scorrelato dalla struttura dell'input (la chiave stessa). Da qui il termine "hash", che in inglese significa "tritare" (e quindi "polpetta"): si sminuzza la chiave in pezzettini che sono rimescolati in modo da formare un intero che sia il più "casuale" possibile. Ovviamente l'algoritmo che calcola la funzione hash non è casuale, in quanto se si ricalcola più volte $H(k)$ per la stessa chiave k si ottiene sempre lo stesso valore, ma l'indirizzo hash si comporta da un punto di vista statistico come se fosse stato davvero prodotto con uno o più lanci casuali di una moneta.

Si supponga che gli m elementi di V siano indicati da 0 ad $m-1$, anziché da 1 ad m. Per definire funzioni hash, è conveniente considerare la rappresentazione binaria $bin(k)$ della chiave k. Se la chiave k non è numerica, $bin(k)$ è data dalla concatenazione della rappresentazione binaria di ciascun carattere che la compone. È altresì utile denotare con $int(b)$ il numero intero rappresentato da una stringa binaria b. In

questo modo, oltre a poter agevolmente passare da una chiave alfanumerica k alla sua rappresentazione binaria $bin(k)$, si può passare da una qualsiasi stringa binaria b, inclusa $b = bin(k)$, ad un numero intero $int(b)$. In particolare, ciò è possibile anche in Pascal. Infatti la funzione primitiva *ord* del Pascal, applicata ad un carattere, restituisce un intero che è il numero ordinale del carattere nell'insieme dei caratteri ammessi. Attraverso la rappresentazione binaria degli ordinali associati ai caratteri presenti in k si può ricavare $bin(k)$ e anche $int(bin(k))$.

Quattro buoni metodi di generazione di indirizzi hash sono i seguenti. I primi tre presuppongono $m = 2^p$, per qualche p, mentre il quarto presuppone m dispari (meglio se primo):

a. $H(k) = int(b)$, dove b è un sottoinsieme di p bit di $bin(k)$, solitamente estratti nelle posizioni centrali;
b. $H(k) = int(b)$, dove b è dato dalla somma modulo 2, effettuata bit a bit, di diversi sottoinsiemi di p bit di $bin(k)$;
c. $H(k) = int(b)$, dove b è un sottoinsieme di p bit estratti dalle posizioni centrali di $bin(int(bin(k))^2)$;
d. $H(k)$ è uguale al resto della divisione di $int(bin(k))$ per m.

Esempio 6.6 (Funzioni hash). Si supponga che le chiavi siano di 6 caratteri alfanumerici. Chiavi più lunghe sono troncate, mentre chiavi più corte sono espanse a destra con spazi. Si assuma $ord(A) = 1$, $ord(B) = 2, \ldots, ord(Z) = 26$ e $ord(\underline{b}) = 32$, dove $\underline{b}$ indica lo spazio, con rappresentazione di ogni ordinale su 6 bit. Per le chiavi WEBER e WEBERN si ottengono le seguenti rappresentazioni su 36 bit, dove si sono evidenziati per chiarezza i 6 gruppi di 6 bit che rappresentano i caratteri della chiave:

$$\begin{aligned} bin(\text{WEBER}\underline{b}) &= 010111\ 000101\ 000010\ 000101\ 010010\ 100000 \\ bin(\text{WEBERN}) &= 010111\ 000101\ 000010\ 000101\ 010010\ 001110 \end{aligned}$$

Calcoliamo gli indirizzi hash ottenuti con le funzioni definite in (a), (b) e (d):

a. Sia $m = 256 = 2^8$. Estraendo gli 8 bit dalla posizione 15 alla 22 di $bin(k)$, $H(\text{WEBER}) = H(\text{WEBERN}) = int(00100001) = 33$. Le due chiavi danno pertanto luogo ad una collisione.
b. Sia ancora $m = 256 = 2^8$ e si calcoli b raggruppando $bin(k)$ in cinque gruppi di 8 bit, dopo aver espanso a destra $bin(k)$ con quattro zeri in modo che $bin(k)$ sia formato da 40 bit. $H(\text{WEBER}) = int(11000011) = 195$ mentre $H(\text{WEBERN}) = int(00100001) = 33$. Infatti, indicando con $\oplus$ la somma bit a bit modulo 2, si ottiene:

$$\begin{aligned} 11000011 &= 01011100 \oplus 01010000 \oplus 10000101 \oplus 01001010 \oplus 00000000 \\ 00100001 &= 01011100 \oplus 01010000 \oplus 10000101 \oplus 01001000 \oplus 11100000 \end{aligned}$$

Benché $H(\text{WEBERN})$ sia uguale a 33, come nel caso (a), la collisione è stata eliminata.
d. Sia $m = 383$. Poiché si usano 6 bit per rappresentare ciascun carattere della chiave, $int(bin(k))$ è interpretato come un numero in base $2^6 = 64$.

$$\begin{aligned} int(bin(\text{WEBERN})) &= 23 \cdot 64^5 + 5 \cdot 64^4 + 2 \cdot 64^3 + \\ &\quad + 5 \cdot 64^2 + 18 \cdot 64^1 + 14 \cdot 64^0 = \\ &= 64(64(64(64(23 \cdot 64) + 5) + 2) + 5) + 18) + 14. \end{aligned}$$

H(WEBERN) è ottenuto prendendo il resto della divisione

$$int(bin(\text{WEBERN}))/383.$$

Il calcolo completo di H(WEBERN) e H(WEBER) è lasciato per esercizio. □

La scelta in (a) e (c) di estrarre i p bit dalle posizioni centrali è dovuta al fatto che molte chiavi alfanumeriche potrebbero iniziare con la stessa sequenza di caratteri, così come molte chiavi potrebbero terminare con degli spazi perché espanse a destra in quanto inizialmente troppo brevi (come visto nell'Esempio 6.6).

La funzione hash definita in (d) presuppone che m sia dispari, perché se m fosse uguale a 2^p allora due numeri con gli stessi p bit finali darebbero sempre luogo ad una collisione. Questa funzione è la migliore dal punto di vista probabilistico e fornisce un'eccellente distribuzione degli indirizzi $H(k)$ nell'intervallo $[0, m-1]$, specie se m è un numero primo non troppo vicino ad una potenza di 2 (o a qualsiasi altra base con la quale il calcolatore utilizzato rappresenti i numeri). Se il calcolatore rappresenta numeri usando la base 2, allora alcuni valori adeguati per m sono: 13, 23, 47, 97, 193, 383, 769, 1531, 3067, 6143, 12289, 24571.

Nel calcolare la funziona hash definita in (d), è conveniente applicare la cosiddetta "regola di Horner" per la valutazione di un polinomio in un punto, come già anticipato implicitamente nell'Esempio 6.6. Infatti, un qualsiasi polinomio

$$p(x) = a_q x^q + a_{q-1}x^{q-1} + \cdots + a_1 x + a_0$$

può essere riscritto come

$$p(x) = x(\cdots(x(a_q x) + a_{q-1})\cdots) + a_1) + a_0$$

Per valutare p nel punto x, si procede secondo tale parentesizzazione, dalla parentesi più interna verso quella più esterna. In questo modo il polinomio di grado q è valutato in x utilizzando soltanto q prodotti e q addizioni. La seguente procedura Pascal calcola la funzione hash utilizzando la regola di Horner ed effettuando l'operazione **mod** ad ogni iterazione, per evitare che b divenga troppo grande e provochi "overflow" (supero della capacità di una parola di memoria).

```
const dimchiave = ... ; m = ... ;
type tipoelem = array[1..dimchiave] of character;

function H(k: tipoelem): integer;
    var b, j: integer;
    begin
        b := ord(k[1]);
        for j := 2 to dimchiave do
            b := ((b*64) + ord(k[j])) mod m;
        H := b
    end;
```

Ovviamente, per chiavi definite su alfabeti di dimensione diversa si userà 32 o 128 in luogo di 64. La complessità della funzione Pascal H è $O(1)$, ma la costante dipende da *dimchiave*, la lunghezza della chiave. Per chiavi molto lunghe è opportuno usare una combinazione dei metodi (b) e (d), al fine di ridurre il tempo richiesto dal calcolo della funzione hash.

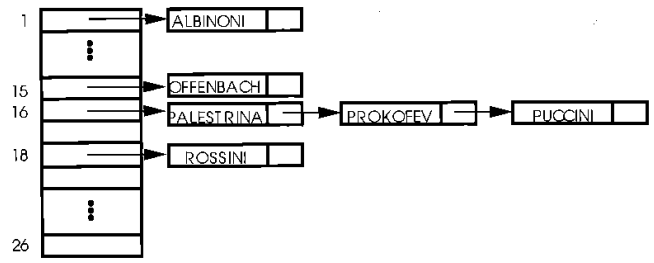


Figura 6.2: Liste di trabocco.

6.2.2 Metodi di scansione

I metodi di scansione si distinguono in esterni e interni, a seconda che le chiavi siano memorizzate all'esterno o all'interno del vettore V. Il metodo esterno per eccellenza si basa sulle liste di trabocco, mentre quelli interni più comuni sono la scansione lineare, la scansione quadratica, la scansione pseudocasuale, e l'hashing doppio. Il metodo esterno è il più semplice e, come vedremo, anche il più efficiente. Il vettore V contiene, in ogni posizione, l'identificatore di una lista, detta appunto "lista di trabocco", e tutte le chiavi che collidono sullo stesso indirizzo hash vengono inserite nella stessa lista.

Esempio 6.7 (Liste di trabocco). La Fig. 6.2 illustra la memorizzazione delle chiavi dell'Esempio 6.5 utilizzando liste di trabocco (monodirezionali senza sentinella). □

Il metodo di scansione con liste di trabocco presenta il duplice vantaggio di non imporre alcun limite alla capacità del dizionario e di evitare completamente il formarsi di agglomerati. Le operazioni APPARTIENE, INSERISCI e CANCELLA per una chiave k si riducono, rispettivamente, a procedure di ricerca, inserzione in testa e cancellazione di k nella lista $V[H(k)]$ e richiedono tempo lineare nella lunghezza della lista, purché sia scelta una realizzazione efficiente per le liste. Per esempio, una realizzazione adeguata si può basare su un vettore di puntatori a liste bidirezionali (usando cursori, invece, si rischia che la memoria riservata alle liste col vettore SPAZIO si esaurisca). Ovviamente la dimensione m di V, cioè il numero di liste, va scelta in modo oculato, così da evitare sia troppe liste, la maggior parte delle quali sarebbero vuote, sia poche liste, che tenderebbero ad essere troppo lunghe.

Se non è possibile l'utilizzo di puntatori, allora si ricorre ad un metodo di scansione interna. In questo caso non ci sono liste esterne a V che vengono scandite seguendo puntatori, ma piuttosto si calcola esplicitamente la sequenza di scansione all'interno di V (come si è visto nell'Esempio 6.5).

Sia f_i la funzione che viene utilizzata, in un metodo di scansione interna, l'i-esima volta che una posizione del vettore V è trovata già occupata, con $i \geq 1$ (per $i = 0$, $f_0 = H$). La funzione f_i va scelta in modo da toccare tutte le posizioni del vettore V. Così, se il vettore V è completamente occupato e la chiave k da ricercare non vi è contenuta, la ricerca si può arrestare non appena verrà toccata per la seconda volta la posizione di partenza $H(k)$.

Nella scansione lineare, $f_i = (H(k) + hi) \bmod m$, dove h è un intero primo con m. La costante h rappresenta la distanza tra due successive posizioni esaminate nella scansione (se $h = 1$, si ottiene la scansione a passo unitario vista nell'Esempio 6.5).

La divisione in modulo rende la scansione "circolare", così da continuare dall'inizio di V se si tenta di uscire oltre la fine di V. L'ipotesi di primalità tra m e h garantisce che la scansione tocchi tutte le posizioni di V prima di riconsiderare posizioni già esaminate. Questo metodo di scansione ha lo svantaggio di non ridurre la possibilità di formazione di agglomerati di chiavi. Infatti, se due chiavi k e k' hanno indirizzi hash $H(k') = H(k) + h$, le due sequenze di scansione $H(k)$, $H(k) + h$, $H(k) + 2h$, $H(k) + 3h, \ldots$ e $H(k')$, $H(k') + h$, $H(k') + 2h, \ldots$ collidono a partire dalla posizione $H(k) + h$. In questo caso, il vantaggio di avere due indirizzi hash distinti per k e k' è di fatto annullato.

Con la scansione quadratica, $f_i = (H(k) + hi + i(i\text{-}1)/2) \bmod m$, dove m è un numero primo. Questo metodo riduce l'effetto di agglomerazione dovuto alla scansione lineare poiché la distanza tra due successive posizioni nella sequenza di scansione non è più costante ma variabile. Purtroppo, data k, la sequenza di scansione non include tutte le posizioni di V, ma solo un sottoinsieme. Può non essere dunque possibile l'inserimento di k perché tutte le posizioni considerate nella sequenza di scansione sono occupate, mentre possono esistere altre posizioni libere non considerate nella sequenza. Questo svantaggio, comunque, è talvolta trascurabile, specie per m non troppo piccolo.

La scansione pseudocasuale usa $f_i = (H(k) + r_i) \bmod m$, dove r_i è l'i-esimo numero generato da un generatore di numeri pseudo-casuali (vedi Appendice A). Tale generatore genera tutti gli interi tra 1 ed $m - 1$ esattamente una volta e in un ordine qualunque. In altri termini, la sequenza generata $r_1, r_2, \ldots, r_{m-1}$ è una permutazione di $1, 2, \ldots, m - 1$ che dal punto di vista statistico è come se fosse "casuale". Se il generatore è ben progettato, la differenza tra due interi r_i e r_{i+1} generati successivamente è continuamente variabile e questa proprietà elimina entrambi gli svantaggi visti per la scansione lineare e quella quadratica.

Infine, l'hashing doppio impiega $f_i = (H(k) + iF(k)) \bmod m$, dove F è un'altra funzione hash diversa da H. In questo modo, anche se per due chiavi k e k' si ha $H(k) = H(k')$, vi è un'alta probabilità che le due ampiezze del passo di scansione $F(k)$ ed $F(k')$ siano diverse. Ciò evita, in pratica, la formazione di agglomerati. Affinché la sequenza di scansione possa includere tutte le posizioni, occorre che il passo di scansione $F(k)$ sia primo rispetto ad m. Questa condizione può essere soddisfatta imponendo che F generi solo numeri dispari, qualora m sia una potenza di 2, oppure che F generi interi minori di m, qualora m sia un numero primo.

Usando i metodi di scansione interna e permettendo la cancellazione di chiavi, però, sorge un nuovo problema. Una volta raggiunta una posizione "vuota" nella ricerca di una chiave k, non si è mai sicuri che la chiave stessa non si trovi da qualche altra parte del vettore, perché la posizione attualmente vuota poteva essere occupata quando k è stata inserita.

Esempio 6.8 (Cancellazione). Cancellando PROKOFEV dalla tabella della Fig. 6.1 ed utilizzando la scansione lineare a passo unitario, la successiva ricerca di PUCCINI fallisce. Infatti tale ricerca si arresta sulla prima posizione vuota dopo la posizione 16, cioè sulla 17, mentre PUCCINI è memorizzato nella posizione 18. □

Con i metodi di scansione interna, una posizione in cui è avvenuta una cancellazione, pur essendo disponibile per una eventuale inserzione, non può essere considerata nella scansione come una posizione mai riempita! Occorre pertanto scandire anche le posizioni in cui si è cancellato ed arrestarsi o sulla prima posizione mai riempita o dopo essere ritornati su una posizione già scandita. Poiché ciò comporta un aumento nel tempo di ricerca, se sono previste molte cancellazioni è più conveniente utilizzare il metodo di scansione esterna.

La seguente procedura Pascal SCANDISCI effettua la scansione lineare a passo unitario ($h = 1$) e si arresta su una posizione del vettore V contenente la chiave k, oppure sulla prima posizione "vuota", oppure dopo aver esaurito tutto V. La posizione su cui si arresta è restituita nel parametro j. In un altro parametro c è invece restituita l'eventuale prima posizione "cancellata" o "vuota" che è stata incontrata nella scansione.

```
const vuoto = ... ; cancellato = ... ;
type dizionario = array[0..m − 1] of tipoelem;

procedure SCANDISCI(k: tipoelem; var V: dizionario; var j, c: integer);
    var i: integer; libero, trovato, esaurito: boolean;
    begin
        i := H(k); j := i; c := i;
        libero := false; trovato := false; esaurito := false;
        while (V[j] ≠ vuoto) and not (trovato or esaurito) do
            if V[j] = k then
                trovato := true
            else begin
                j := (j + 1) mod m;
                if not libero and ((V[j] = cancellato) or (V[j] = vuoto))
                    then begin
                        c := j;
                        libero := true
                    end;
                esaurito := (i = j)
            end
    end;
```

Utilizzando la SCANDISCI è facile realizzare le procedure APPARTIENE, INSERISCI e CANCELLA. Infatti, dopo la scansione, se $V[j] = k$ allora la chiave appare nel dizionario (e può essere eventualmente cancellata), mentre se $V[j] \neq k$ la chiave k non compare nel dizionario (e può essere eventualmente inserita in $V[c]$ se $V[c] = vuoto$ oppure $V[c] = cancellato$). Se infine $V[j] \neq k$, $V[c] \neq vuoto$, e $V[c] \neq cancellato$, allora il vettore è stato completamente scandito (e non è possibile l'eventuale inserzione di k).

```
function APPARTIENE(k: tipoelem; var V: dizionario): boolean;
    var j, c: integer;
    begin
        SCANDISCI(k, V, j, c);
        APPARTIENE := (V[j] = k);
    end;

procedure INSERISCI(k: tipoelem; var V: dizionario);
    var j, c: integer;
    begin
        SCANDISCI(k, V, j, c);
        if (V[j] ≠ k) and ( (V[c] = vuoto) or (V[c] = cancellato))
            then V[c] := k;
    end;
```

```
procedure CANCELLA(k: tipoelem; var V: dizionario);
    var j, c: integer;
    begin
        SCANDISCI(k, V, j, c);
        if (V[j] = k) then V[j] := cancellato
    end;
```

6.2.3 Complessità media

Benché nel caso pessimo la ricerca di un elemento possa richiedere la scansione di tutta la tabella, nel caso medio tale ricerca è drasticamente più veloce. È stato infatti dimostrato che il tempo medio di ricerca dipende soltanto dal cosiddetto fattore di carico della tabella, cioè dal rapporto tra il numero di chiavi memorizzate nella tabella e la dimensione della tabella stessa. In altri termini, tabelle di dimensioni diverse presentano lo stesso tempo di ricerca qualora la percentuale di posizioni occupate sia la stessa. Siano:

n = numero di chiavi memorizzate nella tabella hash,
m = dimensione della tabella hash,
α = n/m (fattore di carico),
$I(\alpha)$ = numero medio di accessi alla tabella per la ricerca di una chiave non presente nella tabella (ricerca con insuccesso),
$S(\alpha)$ = numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella (ricerca con successo).

Le grandezze $I(\alpha)$ ed $S(\alpha)$ possono adeguatamente rappresentare la complessità media degli operatori APPARTIENE, INSERISCI e CANCELLA. Infatti, per inserire un elemento non presente nella tabella occorre innanzitutto una ricerca con insuccesso, seguita dall'inserimento vero e proprio, mentre per cancellare un elemento presente nella tabella occorre una ricerca con successo, seguita dalla cancellazione vera e propria. Pertanto, i tempi medi di inserzione e cancellazione dipendono, a meno di costanti, da $I(\alpha)$ ed $S(\alpha)$.

Le grandezze $I(\alpha)$ ed $S(\alpha)$ sono state valutate matematicamente, assumendo una distribuzione uniforme delle chiavi. Le funzioni ottenute al variare di α per i vari metodi di scansione sono, con buona approssimazione, le seguenti:

SCANSIONE	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \le \alpha < 1$	$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$	$\frac{1-\alpha/2}{1-\alpha}$
Quadratica, Pseudocasuale, Hashing doppio	$0 \le \alpha < 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$
Liste di trabocco	$\alpha \ge 0$	$1+\alpha$	$1+\alpha/2$

Il risultato per la scansione lineare è alquanto difficile da dimostrare e non può essere qui riportato. In contrasto, il risultato per le liste di trabocco si dimostra facilmente nel modo seguente. Il fattore di carico α rappresenta anche la lunghezza media di ciascuna lista di trabocco. In caso di successo, mediamente viene scandita

metà lista, mentre in caso di insuccesso viene scandita un'intera lista. In entrambi i casi, è necessario un accesso al vettore V che contiene il puntatore alla lista.

Per i rimanenti metodi di scansione interna, la difficoltà principale consiste nel dimostrare che non si creano agglomerati. Per semplicità, si assuma che ciò sia stato dimostrato e si consideri una ricerca con insuccesso. Se si denota con p_i la probabilità di avere esattamente i collisioni, il costo di una ricerca con insuccesso è

$$1 + \sum_{i=1}^{\infty} i p_i$$

dove il termine 1 è il costo per l'accesso all'unica posizione vuota di V che non dà collisione, e $p_i = 0$ per $i > n$, essendoci n chiavi memorizzate. Per valutare la serie, si usa la relazione

$$\sum_{i=1}^{\infty} i p_i = \sum_{i=1}^{\infty} q_i$$

dove q_i denota la probabilità di avere almeno i collisioni. Per ricavare le probabilità q_i, si osservi che $q_1 = n/m = \alpha$, perché si ha almeno una collisione accedendo ad una delle n posizioni occupate sulle m totali. Invece, $q_2 = (n/m)((n-1)/(m-1))$ perché, dopo aver trovato la prima posizione occupata, si ha almeno un'altra collisione accedendo ad una delle rimanenti $n-1$ posizioni occupate tra le $m-1$ posizioni non ancora accedute. Iterando il procedimento, si ricava $q_i = (n/m)((n-1)/(m-1)) \cdots ((n-i+1)/(m-i+1))$. Per m ed n grandi, quest'ultima quantità si può maggiorare con $(n/m)^i = \alpha^i$. Pertanto il costo di una ricerca con insuccesso è, per $0 \le \alpha < 1$:

$$I(\alpha) \le 1 + \sum_{i=1}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

D'altro canto, il costo di una ricerca con successo è pari al costo di una ricerca con insuccesso quando la chiave è stata inserita, poiché in entrambi i casi viene effettuata la stessa sequenza di scansione. Al momento dell'inserzione, però il fattore di carico era diverso da quello che si ha al momento della ricerca con successo.

Sia β il fattore di carico all'atto dell'inserzione ed α quello all'atto della ricerca con successo. Facendo la media sui possibili valori di β, che sono compresi tra 0 ed α, si ottiene:

$$S(\alpha) = \frac{1}{\alpha}\int_0^{\alpha} \frac{1}{1-\beta} d\beta = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} = -\frac{1}{\alpha}\ln(1-\alpha)$$

Un accurato confronto delle funzioni riportate nella Fig. 6.3 mostra la netta superiorità del metodo delle liste di trabocco, per $\alpha < 1$. Per $\alpha \ge 1$, questo metodo non è confrontabile con gli altri, che sono utilizzabili soltanto per $\alpha < 1$.

Tra i metodi di scansione interna, quello lineare è il più lento, specie per valori di α vicini ad 1. Per valori contenuti di α, anche la scansione lineare può talvolta risultare conveniente (p.e. imponendo $\alpha \le 0.5$, allora $I(\alpha) \le 4$ e $S(\alpha) \le 2$). È

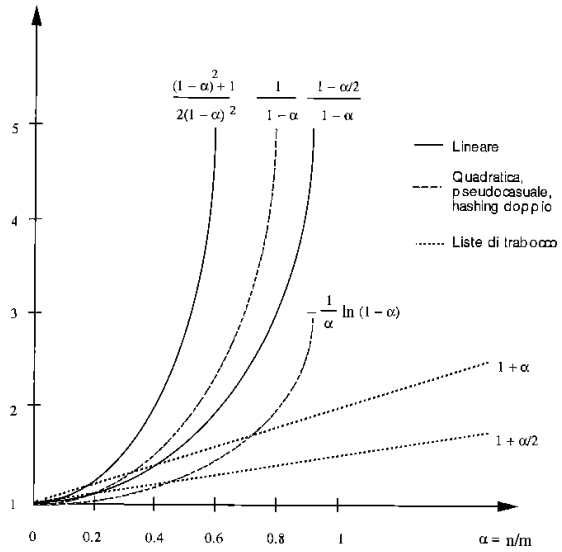


Figura 6.3: Numero medio di accessi per ricerche in tabelle hash.

chiaro, comunque, che anche per gli altri metodi interni conviene evitare di spingere a livelli elevati la percentuale di occupazione della tabella. Infatti il tempo di ricerca cresce rapidamente all'aumentare di α. Inoltre, le cancellazioni di chiavi non fanno diminuire il tempo di ricerca, poiché nella scansione occorre passare anche sulle posizioni che contengono la marca *cancellato*. In ogni caso, qualora siano previste molte cancellazioni, è conveniente utilizzare le liste di trabocco.

Comunque, vi è un caso in cui la scansione lineare a passo unitario risulta più vantaggiosa degli altri metodi. Ciò accade quando la tabella non risieda in memoria principale, ma su dispositivi di memoria secondaria, siano essi ad accesso sequenziale, o anche ad accesso casuale ma con chiavi raggruppate in pagine. In quest'ultimo caso, poiché il trasferimento di una pagina dalla memoria secondaria a quella principale è lento, la scansione lineare risulta conveniente. Infatti permette di effettuare la scansione in memoria principale all'interno della stessa pagina, riducendo così il numero di trasferimenti di pagine dalla memoria secondaria a quella principale.

6.2.4 Ristrutturazione

Usando la scansione interna, non è conveniente che la tabella si riempia troppo. In pratica, quando a causa di un'inserzione α diviene maggiore di una prefissata soglia (di solito 0.5), è vantaggioso ristrutturare la tabella. La tabella viene ridimensionata con m raddoppiato e sono reinserite nella nuova tabella tutte le chiavi presenti in quella vecchia (cambiando funzione hash, ovviamente, visto che m è raddoppiato). In questo modo, la nuova tabella ha fattore di carico al più dimezzato (cioè non superiore a 0.25) e non ci sono più posizioni "cancellate" (cioè sono tutte o occupate

o mai usate). Chiaramente il costo di tale ristrutturazione è $O(m)$ nel caso pessimo, ma è possibile dimostrare che il costo "ammortizzato" delle operazioni APPARTIENE, INSERISCI e CANCELLA resta $O(1)$. Per semplicità, vediamo la dimostrazione considerando solo inserzioni e trascurando le cancellazioni. Assumiamo $m = 2^p$. Sia data una sequenza di j operazioni INSERISCI e sia c_i il numero di inserzioni di chiavi nella tabella dovute all'i-esima INSERISCI. Poiché le ristrutturazioni avvengono solo quando i supera una potenza di 2 maggiore o uguale ad $m/2$ (cioè per $i - 1 = m/2$, m, $2m$, $4m$, ecc.) si ha:

$$c_i = \begin{cases} i & \text{se } i-1 = 2^s \geq 2^{p-1}, \quad \text{(perché avviene una ristrutturazione)} \\ 1 & \text{altrimenti.} \quad \text{(perché non c'è ristrutturazione)} \end{cases}$$

Il numero complessivo $C(j)$ di inserzioni di chiavi causate dalla sequenza di j INSERISCI è quindi:

$$C(j) = \sum_{i=1}^{j} c_i \leq j + \sum_{i=0}^{\log j} 2^i < j + 2j = 3j$$

Poiché la sequenza di j INSERISCI costa $C(j) = 3j$ inserzioni di chiavi, il costo ammortizzato di ciascuna INSERISCI è $C(j)/j = 3$. Pertanto, la complessità media di ciascuna INSERISCI resta $O(1)$ anche impiegando ristrutturazioni della tabella.

Il risultato precedente si può estendere al caso in cui la sequenza di operazioni includa sia INSERISCI sia CANCELLA e si usi ristrutturare la tabella sia espandendola quando è troppo piena (come visto precedentemente), sia contraendola quando è quasi vuota (ridimensionandola con m dimezzato quando α scende sotto una certa soglia, per esempio 0.125).

Analoghe ristrutturazioni in espansione o contrazione sono convenienti anche per le liste di trabocco o per ogni struttura realizzata con un vettore che si "saturi". Per esempio, con le liste di trabocco è conveniente espandere se α supera 2, mentre nel caso di pile [cfr. § 3] o heap [cfr. § 7] si espande il vettore quando α raggiunge 1.

6.3 Esercizi

Esercizio 6.1 (Conta gli zeri). Un vettore V di n numeri binari contiene tutti gli zeri nelle prime posizioni, seguiti da tutti gli uni. Si scrivano due funzioni Pascal, basate sulla ricerca binaria, per determinare il numero h di zeri in tempo $O(\log n)$ e $O(\log h)$, rispettivamente.

Si assume che V abbia indici nell'intervallo $1..n$, con $n \geq 1$. La funzione di complessità $O(\log n)$ esegue una ricerca binaria considerando il valore $V[m]$ dell'elemento centrale. Se $V[m] = 1$ viene fatta una chiamata ricorsiva sulla prima metà $V[i..m-1]$, mentre se $V[m] = 0$ è effettuata una chiamata sulla seconda metà $V[m+1..j]$. Per evitare lo "scavalcamento" degli indici i e j, la ricorsione è arrestata quando la porzione ancora da esaminare comprende o un solo elemento (caso $i = j$) oppure due elementi (caso $i = j - 1$). In entrambi i casi, se l'elemento sinistro $V[i]$ è 1, allora l'ultimo 0 si trova in posizione $h = i - 1$ (se non c'è alcuno 0 allora deve essere $i = 1$ e correttamente $h = i - 1 = 0$); se invece l'elemento destro $V[j]$ è 0, allora $h = j$ (se ci sono tutti zeri, allora deve essere $j = n$ e quindi $h = j = n$). Infine, la terza possibilità occorre solo se ci sono esattamente due elementi (cioè quando $i = j - 1$) dei quali il

sinistro non è 1 ed il destro non è 0; ma in tal caso deve essere $V[i] = 0$ e $V[j] = 1$, ed il numero h di zeri è i. La chiamata nel programma principale è CONTA0(V, 1, n).

```
function CONTA0(var V: vettore; i, j: integer): integer;
    var m: integer;
    begin
        if (i = j) or (i = j − 1) then
            if V[i] = 1 then
                CONTA0 := i − 1
            else if V[j] = 0
                then CONTA0 := j
                else CONTA0 := i
        else begin
            m := (i + j) div 2;
            if V[m] = 1
                then CONTA0 := CONTA0(V, i, m − 1)
                else CONTA0 := CONTA0(V, m+1, j)
        end
    end;
```

Per contare il numero h di zeri in tempo $O(\log h)$, si scandisce con un ciclo **while** il vettore V da sinistra verso destra per potenze di due. Se l'elemento considerato $V[i]$ è uguale a 0, si procede nella scansione, raddoppiando i, ma evitando di superare la dimensione n del vettore. Se al termine della scansione l'ultimo elemento $V[i]$ esaminato vale 0, allora si chiama la CONTA0 tra tale elemento e l'ultimo elemento $V[n]$ del vettore. Altrimenti si chiama la CONTA0 tra l'ultimo elemento $V[i]$ esaminato (che vale 1) e l'elemento $V[i$ **div** $2]$ esaminato in precedenza, se i **div** 2 non è inferiore ad 1, oppure $V[1]$. La chiamata è TROVAh(V, n).

```
function TROVAh(var V: vettore; n: integer): integer;
    var i: integer;
    begin
        i := 1;
        while (V[i] = 0) and (2 * i ≤ n) do i := 2 * i;
        if V[i] = 0 then
            TROVAh := CONTA0(V, i, n)
        else if i div 2 > 1
            then TROVAh := CONTA0(V, i div 2, i)
            else TROVAh := CONTA0(V, 1, i)
    end;
```

Poiché la scansione procede raddoppiando ogni volta i finché non è individuato un intervallo tra due potenze di 2 che include h (diciamo $2^s \leq h \leq 2^{s+1}$), il ciclo **while** costa $O(\log 2^s) = O(\log h)$. La CONTA0 viene poi eseguita sugli elementi appartenenti a tale intervallo, il cui numero è $O(2^s) = O(h)$, e quindi la sua complessita è $O(\log 2^s) = O(\log h)$. Ovviamente, questo metodo è vantaggioso quando h è molto minore di n.

Esercizio 6.2 (Chi manca?). Sia dato un vettore ordinato $V[1..n]$ contenente n elementi interi distinti appartenenti all'intervallo $1..n+1$. Si scriva una procedura Pascal, basata sulla ricerca binaria, per individuare in tempo $O(\log n)$ l'unico intero dell'intervallo $1..n+1$ che non compare in V.

Esercizio 6.3. Si indichi il contenuto di una tabella hash di dimensione 13, inizialmente vuota, dopo l'inserzione, nell'ordine, delle chiavi: D, O, M, A, N, D, A, F, A, C, I, L, E. Si usi la funzione hash $H(k) = k \mod 13$ per la k-esima lettera dell'alfabeto inglese e il metodo di scansione lineare a passo unitario. Si indichi successivamente il contenuto della tabella dopo avervi cancellato, nell'ordine: D, I, F, F, I, C, I, L, E, e poi inserito, nell'ordine: B, A, S, T, A.

Esercizio 6.4. Si definisca la realizzazione di un dizionario contenente in media 256 elementi, in cui la chiave richiede 8 bit, in modo da minimizzare il tempo di accesso.

Esercizio 6.5. Si forniscano procedure Pascal per realizzare i metodi di scansione "quadratica" ed "hashing doppio".

Esercizio 6.6. Un dizionario è realizzato mediante una tabella hash con liste di trabocco. Alcune chiavi sono ricercate molto più frequentemente delle altre, ma non si sa a priori quali siano tali chiavi. Quale accorgimento si può adottare per sperare di abbassare il tempo di accesso a queste chiavi?

Capitolo 7

Code con priorità

Una coda con priorità è un particolare insieme, sugli elementi del quale è definita una relazione "$\leq$" di ordinamento totale, in cui è possibile inserire un nuovo elemento o estrarre l'elemento "minimo". La specifica è simile a quella del tipo di dato "insieme", in cui sono ammesse, oltre alla creazione, le operazioni INSERISCI, MIN [cfr. § 5.4] e la nuova operazione CANCELLAMIN, che rimuove l'elemento minimo:

CANCELLAMIN: (prioricoda) $\rightarrow$ prioricoda

CANCELLAMIN(A) = A'
Pre: $A \neq \emptyset$
Post: $A' = A - \{x\}$, con $x =$ MIN(A)

In termini degli operatori del tipo di dato insieme, risulta che

$$\text{CANCELLAMIN}(A) = \text{CANCELLA}(\text{MIN}(A), A).$$

Il nome deriva dall'interpretazione in cui A è una coda di elementi che devono essere serviti rispettandone la priorità. Per semplicità, in questo capitolo si fanno coincidere le priorità con gli elementi stessi e si assumono, in accordo alla specifica del tipo di dato insieme, priorità distinte (in generale, le priorità si possono intendere come proprietà associate agli elementi, alcuni dei quali possono avere la stessa priorità).

Esempio 7.1 (Pronto Soccorso). Nella sala d'attesa del pronto soccorso di un ospedale, un paziente colpito da un incidente che può risultargli fatale sarà soccorso prima di altri pazienti, affetti da malanni meno gravi, che erano già in attesa prima di lui. □

Si può realizzare una coda con priorità di n elementi utilizzando liste, ordinate o non ordinate. Nel caso di liste ordinate, gli operatori MIN e CANCELLAMIN hanno complessità $O(1)$, mentre INSERISCI è $O(n)$. Nel caso di liste non ordinate, la complessità di INSERISCI può scendere a $O(1)$, sapendo che l'elemento da inserire non è già presente, ma quella di MIN e CANCELLAMIN sale a $O(n)$.

Se il numero di elementi presenti nella coda con priorità può essere limitato superiormente da una costante, allora è possibile rendere uniforme la complessità di CANCELLAMIN e INSERISCI con una realizzazione sequenziale efficiente e particolarmente semplice, che usa un vettore chiamato "heap".

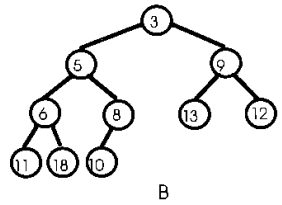


Figura 7.1: Un albero binario B che rappresenta uno heap.

7.1 Realizzazione con heap

Gli elementi della coda con priorità C possono essere disposti in un vettore H (detto appunto heap) che può essere interpretato come un albero binario B. Ciascun nodo di B corrisponde ad una posizione di H che memorizza un elemento di C. L'albero B deve verificare le seguenti proprietà:

1. se h è il livello massimo delle foglie, allora ci sono esattamente $2^h - 1$ nodi di livello minore di h;
2. tutte le foglie di livello h sono addossate a sinistra;
3. ogni nodo contiene un elemento di C che è maggiore di quello contenuto nel padre.

Esempio 7.2 (Albero B). L'albero B di Fig. 7.1 contiene gli elementi della coda con priorità $C = \{5, 10, 8, 11, 13, 12, 9, 18, 3, 6\}$ e verifica le proprietà (1)-(3) suddette. Il livello massimo delle foglie è 3, ci sono $2^3 - 1 = 7$ nodi di livello al più 2, e le 3 foglie di livello 3 sono addossate a sinistra. □

Grazie alle proprietà (1) e (2), ciascun livello k (tranne l'ultimo) di B contiene tutti i possibili 2^k nodi, $0 \leq k < h$, mentre i nodi di livello massimo h stanno a sinistra. In altri termini, B è "quasi pieno", cioè non ha nodi "mancanti" tranne le foglie a destra di livello massimo. Pertanto gli n nodi di B corrispondono alle posizioni $1..n$ del vettore H nell'ordine in cui si incontrano i nodi visitando B per livelli crescenti ed esaminando da sinistra verso destra i nodi allo stesso livello. Poiché ad ogni livello il numero di nodi raddoppia, si ha che

a. la radice di B corrisponde alla posizione 1 di H;
b. il figlio sinistro e il figlio destro di i corrispondono alle posizioni $2i$ e $2i+1$ di H.

È facile verificare che il figlio sinistro (destro) di i non esiste se e solo se $2i > n$ ($2i + 1 > n$). Inoltre, per $1 < i \leq n$, il padre di i è $\lfloor i/2 \rfloor$. Ovviamente, in $H[i]$ è memorizzato l'elemento di C contenuto nel corrispondente nodo di B. Così, per la proprietà (3), se il figlio sinistro (destro) di i esiste, allora $H[2i] > H[i]$ ($H[2i + 1] > H[i]$).

Esempio 7.3 (Heap H). L'albero B di Fig. 7.1 rappresenta il seguente heap: $H[1] = 3$, $H[2] = 5$, $H[3] = 9$, $H[4] = 6$, $H[5] = 8$, $H[6] = 13$, $H[7] = 12$, $H[8] = 11$, $H[9] = 18$, $H[10] = 10$. □

La realizzazione degli operatori MIN, CANCELLAMIN e INSERISCI è molto semplice se si immagina di operare sull'albero B anziché sul vettore H. Per eseguire MIN

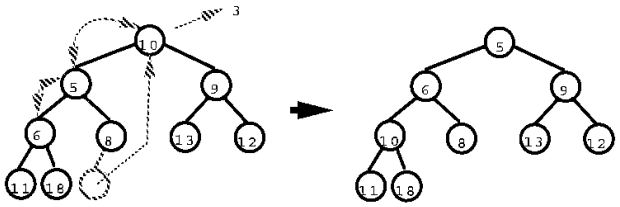


Figura 7.2: Cancellazione del minimo.

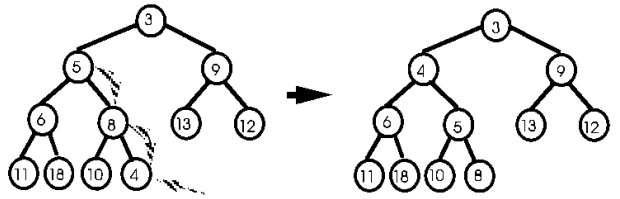


Figura 7.3: Inserimento di 4.

basta leggere l'elemento contenuto nella radice, mentre per eseguire CANCELLAMIN si cancella la foglia di livello massimo più a destra, sì da mantenere verificate le proprietà (1) e (2), se ne copia l'elemento nella radice, e si fa "scendere" tale elemento lungo un percorso radice-foglia, scambiandolo col minimo degli elementi contenuti nei figli, fino a verificare la proprietà (3).

Esempio 7.4 (CANCELLAMIN). Volendo cancellare 3 dall'albero B di Fig. 7.1, si cancella la foglia contenente 10, si ricopia 10 nella radice, si scambia 10 con 5 e successivamente 10 con 6 (si veda la Fig. 7.2). □

Per eseguire INSERISCI, si aggiunge una foglia col nuovo elemento al livello massimo più a sinistra, per verificare le proprietà (1) e (2), e si fa "salire" tale elemento lungo un percorso foglia-radice scambiandolo col padre, fino a verificare la proprietà (3).

Esempio 7.5 (INSERISCI). Volendo inserire l'elemento 4 nell'albero B di Fig. 8.1, si aggiunge un figlio, con 4, al nodo contenente 8, si scambia 4 con 8 e successivamente 4 con 5 (si veda la Fig. 7.3). □

La proprietà (1) di B permette di mantenere il livello massimo h delle foglie limitato da $\lfloor \log n \rfloor$. Infatti il numero n di nodi risulta compreso tra 2^h e $2^{h+1} - 1$, e quindi $h = \lfloor \log n \rfloor$. Poiché CANCELLAMIN e INSERISCI provocano scambi di elementi lungo un solo percorso radice-foglia, entrambi possono essere realizzati in modo da richiedere $O(\log n)$ tempo. Vediamo in dettaglio le procedure Pascal che realizzano la coda con priorità usando il vettore heap. Gli n elementi sono memorizzati in *heap*[1..n] dove *heap*[1] contiene l'elemento minimo. Il cursore *ultimo* registra n, cioè l'ultima posizione occupata, che corrisponde alla foglia di B che sta più a destra tra quelle di livello massimo. Al solito, il valore 0 in *ultimo* indica una coda con priorità vuota.


```
type
    prioricoda = record
        heap: array[1..maxlung] of tipoelem;
        ultimo: 0..maxlung;
    end;

procedure CREAPRIORICODA(var C: prioricoda);
    begin
        C.ultimo := 0
    end;

function MIN(var C: prioricoda): tipoelem;
    begin
        if C.ultimo = 0
            then {errore: prioricoda vuota}
            else MIN := C.heap[1]
    end;
```

Nella INSERISCI, si incrementa il valore del cursore *ultimo*, così da far spazio per il nuovo elemento x. Questo corrisponde ad aggiungere una foglia al livello massimo più a sinistra in B. Si usano due cursori i e k per risalire lo heap. Inizialmente i indica la foglia che è stata aggiunta. Il cursore $k = \lfloor i/2 \rfloor$ indica il padre del nodo i, che esiste solo se $i > 1$, cioè se i non è la radice. Se non è verificata la proprietà (3) tra *heap*[i] e *heap*[k], allora avviene lo scambio e si continua a salire, con i che assume il vecchio valore di k. Nel caso pessimo, la risalita ha termine quando è raggiunta la radice, cioè quando $i = 1$.

```
procedure INSERISCI(x: tipoelem; var C: prioricoda);
    var i, k: integer; temp: tipoelem;
    begin
        if C.ultimo = maxlung then
            {errore: prioricoda piena}
        else begin
            C.ultimo := C.ultimo + 1;
            C.heap[C.ultimo] := x;
            i := C.ultimo;
            if i > 1 then k := i div 2;
            while (i > 1) and (C.heap[i] < C.heap[k]) do begin
                temp := C.heap[i]; C.heap[i] := C.heap[k]; C.heap[k] := temp;
                i := k;
                if i > 1 then k := i div 2;
            end
        end
    end;
```

Nella CANCELLAMIN, si decrementa il valore del cursore *ultimo*, così da rimuovere la foglia al livello massimo più a destra in B, il cui valore è copiato nella radice. Si usano ancora due cursori i e k per discendere nello heap, ma stavolta i indica inizialmente la radice. Il cursore k indica il figlio sinistro $2i$ o quello destro $2i + 1$, se esistono, a seconda di chi contiene l'elemento minimo. Se non è verificata la proprietà (3) tra *heap*[i] e *heap*[k], allora avviene lo scambio e si continua a scendere, con i che assume il vecchio valore di k. Nel caso peggiore, la discesa ha termine

quando è raggiunta una foglia, cioè quando $i > \lfloor n/2 \rfloor$, dove n è il valore contenuto in *ultimo*.

```
procedure CANCELLAMIN(var C: prioricoda);
    var i, k: integer; temp: tipoelem; scambio: boolean;
    begin
        if C.ultimo = 0 then
            {errore: prioricoda vuota}
        else begin
            C.heap[1] := C.heap[C.ultimo];
            C.ultimo := C.ultimo − 1;
            i := 1; scambio := true;
            while (i ≤ C.ultimo div 2) and (scambio) do begin
                k := 2 * i;
                if k < C.ultimo then
                    if C.heap[k] > C.heap[k + 1] then k := k + 1;
                if C.heap[k] < C.heap[i] then begin
                    temp := C.heap[i]; C.heap[i] := C.heap[k]; C.heap[k] := temp;
                    i := k
                end else
                    scambio := false;
            end
        end
    end;
```

La complessità di MIN è $O(1)$, mentre quella di INSERISCI e CANCELLAMIN è $O(\log n)$. Infatti, queste due procedure si basano, rispettivamente, su una "salita" in un percorso foglia-radice e su una "discesa" in un percorso radice-foglia, la cui lunghezza è $O(\log n)$.

7.2 Heapsort

L'impiego più famoso di uno heap permette di abbassare drasticamente la complessità dell'algoritmo di ordinamento di un vettore basato sull'iterata ricerca del minimo introdotto nell'Esempio I.10.

Esempio 7.6 (Ordinamento). Si supponga che la sequenza di n elementi da ordinare sia memorizzata nel vettore A. È possibile ordinare gli elementi di A utilizzando una coda con priorità C, nella quale sono dapprima inseriti in sequenza tutti gli elementi, e dalla quale sono poi estratti, sempre in sequenza, gli elementi minimi.

```
procedure ORDINA(var A: vettore; n: integer);
    var C: prioricoda; i: integer;
    begin
        CREAPRIORICODA(C);
        for i := 1 to n do INSERISCI(A[i], C);
        for i := 1 to n do begin
            A[i] := MIN(C);
            CANCELLAMIN(C)
        end
    end;
```


Se la coda con priorità è realizzata con uno heap, la complessità della procedura ORDINA è $O(n \log n)$. Per quanto visto nell'Esempio 4.12, una limitazione inferiore alla complessità del problema dell'ordinamento utilizzando confronti è $\Omega(n \log n)$. La procedura ORDINA ha quindi complessità ottima. □

Benché la procedura ORDINA abbia complessità ottima se si usa uno heap, essa presenta lo svantaggio di richiedere un doppio trasferimento di elementi dal vettore A allo heap e dallo heap di nuovo al vettore A, con conseguente spreco di memoria. È possibile però effettuare l'ordinamento "in loco", cioè direttamente nel vettore A senza un secondo vettore d'appoggio, dopo aver "ristrutturato" A stesso in modo che verifichi le proprietà di uno heap. Poiché si vuole ordinare A in ordine crescente, è opportuno che dopo la ristrutturazione A contenga l'elemento massimo nella radice anziché il minimo (la ragione di questa scelta sarà più chiara in seguito). Ciò implica ovviamente che dopo la ristrutturazione $A[i]$ sia maggiore di $A[2i]$ e $A[2i+1]$, se esistono. Anche se l'ordinamento vero e proprio richiede sempre $O(n \log n)$ tempo, la ristrutturazione iniziale di A può essere fatta in tempo $\Theta(n)$.

La ristrutturazione di A avviene richiamando più volte una opportuna procedura RESTAURAHEAP(A, i, j) che ricostruisce la sezione di heap compresa tra le posizioni i e j del vettore A qualora solo l'elemento in posizione i violi le proprietà dello heap, cioè $A[i]$ sia minore di almeno uno dei suoi figli. La RESTAURAHEAP è una procedura ricorsiva che verifica innanzitutto se i ha un figlio nella porzione di vettore $A[i..j]$. Se $i > \lfloor j/2 \rfloor$, allora i è una foglia e la procedura termina. Altrimenti si individua $A[k]$ come $A[2i]$ o il massimo tra $A[2i]$ e $A[2i+1]$, se esistono entrambi. Se $A[i] < A[k]$ allora vengono scambiati tra loro $A[i]$ e $A[k]$ ed avviene una chiamata ricorsiva nella porzione $A[k..j]$.

```
procedure RESTAURAHEAP(var A: vettore; i, j: integer);
    var k: integer, temp: tipoelem;
    begin
        if i ≤ j div 2 then begin
            k := 2 * i;
            if k < j then
                if A[k] < A[k + 1] then k := k + 1;
            if A[k] > A[i] then begin
                temp := A[i];
                A[i] := A[k];
                A[k] := temp;
                RESTAURAHEAP(A, k, j)
            end
        end
    end;
```

La complessità di RESTAURAHEAP dipende da i e j. Sia m l'altezza del nodo i nel solito albero che rappresenta lo heap $A[i..j]$, definita come segue: l'altezza di una foglia è zero, mentre l'altezza di un nodo che non è una foglia è uguale al massimo delle altezze dei figli più uno. Poiché la procedura si basa su una discesa dal nodo i fino ad una foglia, essa richiede tempo $O(m)$ dove m è l'altezza del nodo i.

Dato il vettore $A[1..n]$ in cui gli elementi sono disposti in un ordine qualunque, è possibile permutarli richiamando RESTAURAHEAP più volte, dalle foglie verso la radice dell'albero B, in modo che alla fine il vettore A soddisfi la proprietà dello heap. La procedura risultante è la seguente, dove le chiamate a RESTAURAHEAP

sono effettuate solo per $i \leq \lfloor n/2 \rfloor$ perché la porzione $A[\lfloor n/2 \rfloor + 1..n]$ è formata da foglie che per definizione sono già tanti heap di un solo elemento ciascuno.

```
procedure COSTRUISCIHEAP(var A: vettore; n: integer);
    begin
        for i := n div 2 downto 1 do
            RESTAURAHEAP(A, i, n);
    end;
```

Per valutare la complessità di COSTRUISCIHEAP, si noti anzitutto che in uno heap di n elementi ci sono al più $\lceil n/2^{m+1} \rceil$ nodi di altezza m. Poiché RESTAURAHEAP richiede tempo $O(m)$ quando è richiamata su un nodo i di altezza m, la complessità di COSTRUISCIHEAP è

$$O\left(\sum_{m=0}^{\log n} \frac{n}{2^{m+1}} \cdot m\right),$$

che è dello stesso ordine di grandezza di

$$O\left(n \sum_{m=0}^{\log n} \frac{m}{2^m}\right).$$

Dato che

$$\sum_{m=0}^{\log n} \frac{m}{2^m} < \sum_{m=0}^{\infty} \frac{m}{2^m} = \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} = 2$$

la complessità di COSTRUISCIHEAP è $O(n)$.

Terminata la ristrutturazione, l'elemento massimo di $A[1..n]$ si trova in $A[1]$ e può essere sistemato nella posizione n che gli compete nella sequenza ordinata scambiandolo con $A[n]$. Dopo lo scambio, si considera la porzione $A[1..n-1]$, che può violare le proprietà dello heap a causa del valore di $A[1]$. Con una chiamata a RESTAURAHEAP$(A, 1, n-1)$ è possibile ripristinare la struttura di heap in $A[1..n-1]$. A questo punto $A[1]$ può essere scambiato con $A[n-1]$, ed il procedimento iterato su $A[1..n-2]$, $A[1..n-3]$, fino ad $A[1..2]$. La procedura di ordinamento risultante, detta HEAPSORT e dovuta a Williams (1964), è descritta in Pascal come segue:

```
procedure HEAPSORT(var A: vettore; n: integer);
    var i: integer; temp: tipoelem;
    begin
        COSTRUISCIHEAP(A, n);
        for i := n downto 2 do begin
            temp := A[i];
            A[i] := A[1];
            A[1] := temp;
            RESTAURAHEAP(A, 1, i - 1)
        end
    end;
```


La chiamata iniziale a COSTRUISCIHEAP costa $\Theta(n)$. Poiché lo heap $A[1..n]$ ha n elementi, l'altezza della radice (il nodo 1) è $O(\log n)$. Ciascuna chiamata a RESTAURAHEAP richiede pertanto $O(\log n)$ tempo. Essendoci n chiamate a RESTAURAHEAP, la complessità di HEAPSORT è $O(n \log n)$.

7.3 Esercizi

Esercizio 7.1. Si forniscano realizzazioni per le pile e per le code utilizzando una coda con priorità.

Esercizio 7.2. Si specifichi, passo dopo passo, il contenuto dello heap eseguendo la procedura ORDINA per ordinare alfabeticamente gli 11 elementi: O, R, D, I, N, A, M, E, N, T, O. Sorgono difficoltà dalla presenza di elementi con lo stesso valore?

Esercizio 7.3. Si dimostri che la procedura RESTAURAHEAP ha complessità $O(1 + \log(j/i))$.

Esercizio 7.4. Si dimostri che nell'albero B che rappresenta uno heap di n elementi ci sono al più $\lceil n/2m + 1 \rceil$ nodi di altezza m.

Capitolo 8

Alberi bilanciati di ricerca

Le varie realizzazioni di strutture di dati di tipo "insieme" viste finora, hanno privilegiato l'efficienza di certi operatori rispetto ad altri. Per esempio, la tabella hash permette di eseguire molto velocemente le operazioni specifiche dei dizionari (APPARTIENE, INSERISCI e CANCELLA), mentre lo heap è vantaggioso per eseguire le operazioni delle code con priorità (MIN, CANCELLAMIN e anche INSERISCI, se è noto a priori che l'elemento da inserire non è già presente nello heap). Né la tabella hash né lo heap, però, si prestano ad eseguire efficientemente sia le operazioni dei dizionari che quelle delle code con priorità. Infatti, è facile verificare che MIN e CANCELLAMIN hanno complessità $O(max\{n, m\})$ qualora si usi una tabella hash, mentre APPARTIENE e CANCELLA (e anche INSERISCI, nel caso che si debba verificare se l'elemento da inserire sia già presente oppure no) hanno complessità $O(n)$ utilizzando uno heap. In questo capitolo, si vedranno altri metodi alternativi per realizzare strutture che ammettono sia le operazioni dei dizionari che delle code con priorità. Queste realizzazioni si basano sull'impiego di "alberi bilanciati di ricerca", come gli "alberi 2-3" e i "B-alberi binari", con i quali tutte le operazioni hanno complessità $O(\log n)$. La complessità, sia nel caso medio che in quello pessimo, degli operatori APPARTIENE, INSERISCI, CANCELLA, MIN e CANCELLAMIN per tutte le realizzazioni considerate in questo testo è così riassunta nella tabella seguente:

	APPARTIENE	INSERISCI	CANCELLA	MIN	CANCELLAMIN
vettore booleano	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$
lista non ordinata	$O(n)$	$O(1)$*	$O(n)$	$O(n)$	$O(n)$
lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
vettore ordinato	$O(\log n)$**	$O(n)$	$O(n)$	$O(1)$	$O(n)$
hash (caso medio)	$O(1)$	$O(1)$	$O(1)$	$O(\max\{m,n\})$	$O(\max\{m,n\})$
hash (caso pessimo)	$O(\max\{m,n\})$	$O(\max\{m,n\})$	$O(\max\{m,n\})$	$O(\max\{m,n\})$	$O(\max\{m,n\})$
heap	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(\log n)$
alberi bilanciati di ricerca	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

*Sapendo a priori che l'elemento da inserire non è già presente, altrimenti la complessità è $O(n)$.

**Se è utilizzabile l'interpolazione allora la complessità è $O(\log \log n)$.

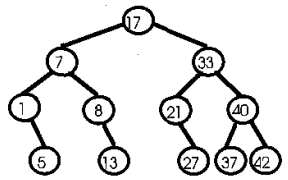


Figura 8.1: Un albero binario di ricerca.

8.1 Alberi binari di ricerca

Gli elementi di un insieme A possono essere contenuti nei nodi di un albero binario B, detto albero binario di ricerca, che verifica le seguenti proprietà:

1. per ogni nodo u, tutti gli elementi contenuti nel sottoalbero radicato nel figlio sinistro di u sono minori dell'elemento contenuto in u;
2. per ogni nodo u, tutti gli elementi contenuti nel sottoalbero radicato nel figlio destro di u sono maggiori dell'elemento contenuto in u.

Esempio 8.1 (Albero binario di ricerca). Un albero binario di ricerca è mostrato nella Fig. 8.1. Si noti che con una visita simmetrica dell'albero si esaminano gli elementi in ordine crescente. □

Le proprietà (1) e (2) hanno lo scopo di agevolare la ricerca di un elemento. È sufficiente confrontare l'elemento da ricercare con quello contenuto nella radice e, se sono diversi, riapplicare il procedimento al sottoalbero radicato nel figlio sinistro o destro, secondo l'esito del confronto. Questo procedimento è molto simile a quello effettuato nella ricerca binaria. Come il nome suggerisce, tale ricerca genera infatti partizioni successive degli elementi secondo uno schema coincidente con un albero binario.

Esempio 8.2 (Ricerca binaria). L'albero binario di ricerca di Fig. 8.1 corrisponde ad una ricerca binaria effettuata sui dati dell'Esempio 6.1. □

Assumendo che l'albero binario sia realizzato con puntatori [cfr. § 4], una funzione Pascal per l'operazione APPARTIENE può essere la seguente.

```
type insieme = binalbero;

function APPARTIENE(x: tipoelem; var u: insieme): boolean;
  begin
    if u = nil then
        APPARTIENE := false
    else if x = u↑.elemento then
        APPARTIENE := true
    else if x < u↑.elemento
        then APPARTIENE := APPARTIENE(x, u↑.sinistro)
        else APPARTIENE := APPARTIENE(x, u↑.destro)
  end;
```

Poiché la funzione APPARTIENE si basa su una discesa lungo un percorso radice-foglia, la sua complessità è $O(h)$, dove h è il livello massimo delle foglie.

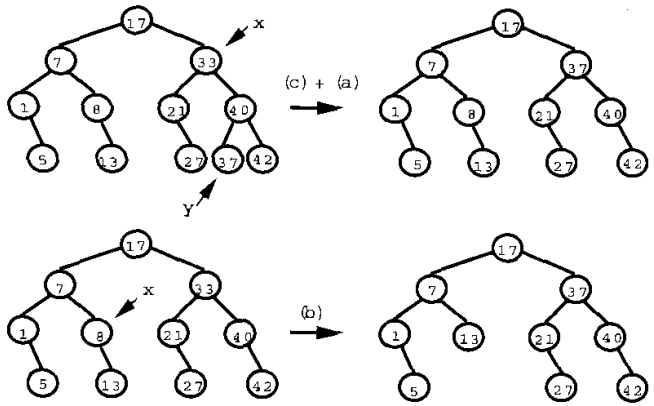


Figura 8.2: I tre casi di cancellazione di un elemento x da un albero binario di ricerca.

L'operatore MIN è facilissimo da realizzare in tempo $O(h)$. Basta infatti effettuare una ricerca in cui si segue sempre il figlio sinistro, fino a trovare un nodo che non ha figlio sinistro.

Per inserire e/o cancellare elementi, occorre innanzitutto effettuare una ricerca, in modo simile a quanto visto per la APPARTIENE, per localizzare il nodo contenente l'elemento (in caso di cancellazione) o il sottoalbero vuoto al posto del quale va sostituita una foglia contenente il nuovo elemento (in caso di inserzione). INSERISCI è banale, poiché l'elemento viene sempre inserito in una nuova foglia che diviene figlio di un nodo che non aveva tale figlio, ma può far aumentare h di uno.

Per realizzare CANCELLA e CANCELLAMIN si procede in modo analogo all'inserzione, con una complicazione in più data dal fatto che l'elemento x da cancellare si trova in un nodo u che non necessariamente è una foglia. Possono sorgere tre casi.

a. Se u non ha figli, allora u è una foglia che viene semplicemente rimossa.
b. Se u ha un solo figlio f, allora u è rimosso rendendo f figlio del padre di u.
c. Se u ha due figli, ci si può ridurre al caso (a) o (b) sostituendo l'elemento da cancellare x con il più piccolo elemento y tale che $y > x$, che si trova nel nodo v raggiungibile scendendo sempre a sinistra nel sottoalbero radicato nel figlio destro di u. L'elemento y prende il posto di x nel nodo u mentre viene rimosso il nodo v che, per definizione, non può avere un figlio sinistro.

Esempio 8.3 (Cancellazione). I casi (a), (b) e (c) per la cancellazione di un elemento x sono illustrati nella Fig. 8.2. □

Usando un albero binario di ricerca, la complessità delle operazioni APPARTIENE, INSERISCI, CANCELLA, MIN e CANCELLAMIN (e anche MAX e CANCELLAMAX) è $O(h)$. Purtroppo, h è $O(n)$ nel caso pessimo, perché l'albero può "allungarsi" inserendo e cancellando nodi. Per fortuna, è possibile modificare la struttura introducendo opportuni metodi di "bilanciamento" che permettono di contenere h entro $O(\log n)$. Una di queste strutture modificate è l'"albero 2-3", introdotto da Hopcroft nel 1970 come diretta generalizzazione dell'albero binario di ricerca.

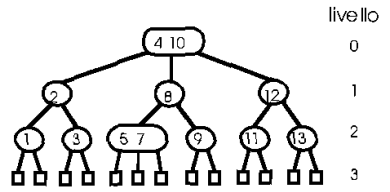


Figura 8.3: Un albero 2-3.

8.2 Alberi 2-3

Gli elementi dell'insieme A possono essere contenuti nei nodi di un albero ordinato D tale che:

1. ogni nodo interno (cioè non foglia) ha k figli, con k uguale a 2 o 3, e contiene $k-1$ elementi di A, ordinati in senso crescente;
2. le foglie non contengono alcun elemento di A;
3. tutte le foglie sono allo stesso livello;
4. se un nodo u ha k figli, allora l'elemento i-esimo di u è maggiore di tutti gli elementi contenuti nel sottoalbero la cui radice è l'i-esimo figlio di u, ed è minore di tutti gli elementi contenuti nel sottoalbero la cui radice è l'$(i+1)$-esimo figlio di u, per $1 \leq i \leq k-1$.

Esempio 8.4 (Albero 2-3). Un albero 2-3 che ospita l'insieme $\{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13\}$ è mostrato nella Fig. 8.3. □

È immediato convincersi che, per le proprietà (1) e (3), in un albero 2-3 con n elementi il livello L delle foglie verifica la seguente relazione:

$$\lceil \log_3(n+1) \rceil \leq L \leq \lceil \log_2(n+1) \rceil.$$

Come nel caso degli alberi binari di ricerca, le proprietà (1) e (4) permettono di effettuare agevolmente la ricerca di un elemento. È infatti sufficiente confrontare l'elemento da ricercare con gli elementi contenuti nella radice e, se l'elemento non è presente ma è maggiore dell'i-esimo elemento della radice e minore dell'$(i+1)$-esimo, riapplicare il procedimento all'$(i+1)$-esimo sottoalbero. In questo modo, APPARTIENE, MIN e MAX possono essere realizzate in modo da richiedere $O(\log n)$ tempo per essere eseguite. Per esempio, per la MIN è sufficiente scendere sempre per il primo figlio, come negli alberi binari di ricerca.

Per inserire un elemento non presente, occorre effettuare una ricerca (come nella APPARTIENE) per individuare il nodo in cui si dovrebbe trovare l'elemento. Se tale nodo, che si trova sempre al livello $L-1$, contiene un solo elemento, si effettua l'inserzione, aggiungendovi inoltre una nuova foglia come nuovo figlio. Se invece il nodo contiene già 2 elementi, lo si sdoppia in due nodi fratelli contenenti un elemento ciascuno. Per rispettare le proprietà (1) e (4), il primo (secondo) fratello prende l'elemento di valore minore (maggiore), mentre l'elemento di valore intermedio si inserisce nel padre. Nel caso pessimo, lo sdoppiamento si ripercuote anche nel padre, poi nel nonno, e così via fino a sdoppiare anche la radice. Poiché ogni sdoppiamento richiede tempo costante e si hanno al più L sdoppiamenti, l'operazione INSERISCI così realizzata ha complessità $O(\log n)$.

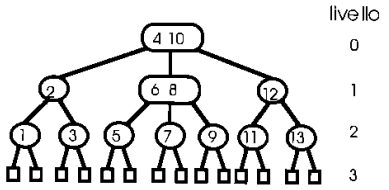


Figura 8.4: Inserzione dell'elemento 6 nell'albero 2-3 di Fig. 8.3.

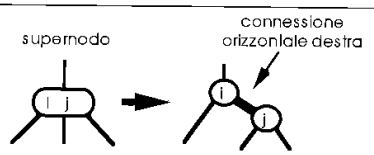


Figura 8.5: Rappresentazione di un supernodo con tre figli dell'albero 2-3 tramite due nodi binari con connessione orizzontale destra.

Esempio 8.5 (Inserzione). Inserendo 6 nell'albero di Fig. 8.3 si provoca lo sdoppiamento del nodo che contiene gli elementi 5 e 7, con l'elemento intermedio (6) che sale nel padre, come illustrato nella Fig. 8.4. □

Anche la cancellazione di un elemento x presente nell'albero 2-3 può essere effettuata in tempo $O(\log n)$, ma sia la comprensione del procedimento seguito che la realizzazione delle rispettive procedure Pascal non è molto semplice perché si possono presentare vari casi a loro volta distinguibili in ulteriori sottocasi. Infatti la cancellazione di un elemento x può provocare l'eliminazione del nodo u che lo contiene con conseguente ridistribuzione degli elementi fra i fratelli e il padre di u e successiva propagazione del procedimento di livello in livello su fino alla radice.

Per semplificare l'operazione di cancellazione, sono stati introdotti da Bayer (1971) i "B-alberi binari" che permettono di rappresentare un albero 2-3 con un albero binario di ricerca in cui il livello delle foglie resta $\Theta(\log n)$. Con i B-alberi binari, tutte le operazioni di dizionari e code con priorità, inclusa soprattutto CANCELLA, sono ragionevolmente semplici da capire e da programmare.

8.3 B-alberi binari

Un B-albero binario è un albero binario T di ricerca tale che ciascun nodo di un albero 2-3 D, detto supernodo, è rappresentato con uno o due nodi binari di T. Una connessione padre-figlio tra due nodi binari è detta orizzontale se è interna allo stesso supernodo, mentre è detta verticale se è tra due supernodi. Soltanto una connessione tra il padre ed il suo figlio destro (detta connessione destra) può essere orizzontale. In altri termini, un supernodo di D con due figli è rappresentato in T con un usuale nodo binario, avente cioè un figlio destro e un figlio sinistro. Invece, un supernodo di D con tre figli è rappresentato in T tramite due nodi binari: il primo nodo binario è detto radice del supernodo, ed ha per figlio destro (tramite una connessione orizzontale destra) il secondo nodo binario, come mostrato nella Fig. 8.5.

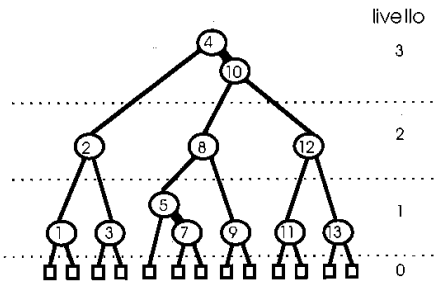


Figura 8.6: Il B-albero binario che rappresenta l'albero 2-3 di Fig. 8.3.

Ad ogni nodo u del B-albero binario è associato un intero, che è il livello del supernodo dell'albero 2-3 che u rappresenta. Anziché partendo dalla radice, è più conveniente numerare i livelli dei supernodi a partire dalle foglie. Così tutte le foglie (che sono vuote) stanno al livello 0, i loro padri sono al livello 1, e così via fino al supernodo radice che si trova al livello L. Poiché un supernodo con tre figli è rappresentato con due nodi binari aventi una connessione (destra) orizzontale, tali nodi binari hanno lo stesso livello (e questa proprietà può essere utilizzata per riconoscere le connessioni orizzontali). Essendo ogni supernodo rappresentato con uno o due nodi binari, un qualsiasi percorso dalla radice ad una foglia nel B-albero binario comprende al più $2L$ nodi binari (esclusa la foglia). Poiché il livello L della radice è $\Theta(\log n)$, segue che qualsiasi percorso radice-foglia nel B-albero binario è lungo $\Theta(\log n)$.

Esempio 8.6 (B-albero binario). La Fig. 8.6 mostra il B-albero binario che rappresenta l'albero 2-3 della Fig. 8.3. Per chiarezza, i livelli sono separati da linee orizzontali tratteggiate. □

Quando si inseriscono o cancellano elementi, e quindi nodi, occorre evitare di alterare la struttura del B-albero binario. In particolare, oltre all'ordinamento delle chiavi tipico degli alberi binari di ricerca, devono essere mantenute le seguenti proprietà:

a. una connessione orizzontale può essere solo destra;
b. un supernodo non può essere formato da più di due nodi binari con connessioni orizzontali;
c. una connessione verticale può avvenire solo se il livello del padre è uguale al livello del figlio più uno.

Per mantenere le proprietà precedenti, si utilizzano ripetutamente due semplici operazioni di bilanciamento INCLINA e FRAZIONA, dovute ad Andersson (1993). Se un nodo binario u ha una connessione orizzontale sinistra, INCLINA(u) modifica il sottoalbero radicato in u effettuando una rotazione destra, come mostrato nella Fig. 8.7. La connessione orizzontale sinistra è sostituita con una destra ed è mantenuto l'ordinamento tra gli elementi memorizzati nei nodi secondo le proprietà degli alberi binari di ricerca. In genere, l'operazione INCLINA può essere propagata seguendo sempre il percorso più a destra a partire da u ed effettuando una rotazione destra ogniqualvolta è incontrato un collegamento orizzontale sinistro.

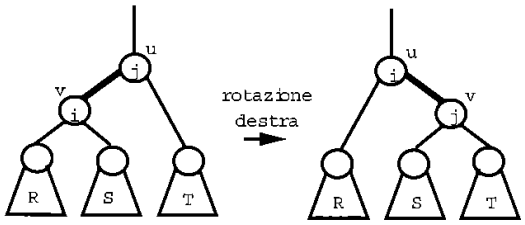


Figura 8.7: Rotazione destra effettuata da INCLINA(u).

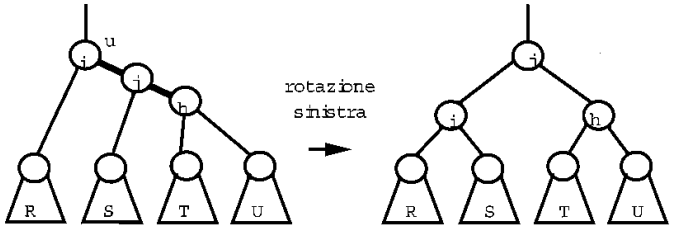


Figura 8.8: Rotazione sinistra effettuata da FRAZIONA(u).

Se sia un nodo binario u, sia suo figlio destro hanno una connessione orizzontale destra, cioè il supernodo radicato in u ha più di due nodi binari, FRAZIONA(u) modifica il sottoalbero radicato in u effettuando una "rotazione sinistra", come mostrato nella Fig. 8.8. È fatto salire di un livello il secondo nodo binario del supernodo, dividendo così il supernodo in tre supernodi, ma mantenendo sempre l'ordinamento tra gli elementi. In genere, l'operazione FRAZIONA può essere propagata seguendo sempre il percorso più a destra a partire da u ed effettuando una rotazione sinistra ogniqualvolta sono incontrati due collegamenti orizzontali destri consecutivi.

Grazie alle operazioni INCLINA e FRAZIONA, è molto semplice descrivere ad alto livello le operazioni INSERISCI e CANCELLA. La regola fondamentale da seguire consiste nell'accertarsi che solo le connessioni destre siano orizzontali prima di controllare la dimensione (numero di nodi binari) di un supernodo. Lo schema per l'inserzione di un nuovo elemento è il seguente:

INSERISCI:
1. Aggiungi un nuovo nodo col nuovo elemento al livello 1;
2. Risali il percorso dal nuovo nodo verso la radice e per ogni nodo binario u incontrato esegui INCLINA(u) seguita da FRAZIONA(u).

Esempio 8.7 (Inserzione). La Fig. 8.9 mostra l'effetto dell'inserzione dell'elemento 6 nel B-albero binario di Fig. 8.6.

Assumendo (per ora) che l'elemento da cancellare si trovi in un nodo al livello 1, l'algoritmo per la cancellazione è il seguente.

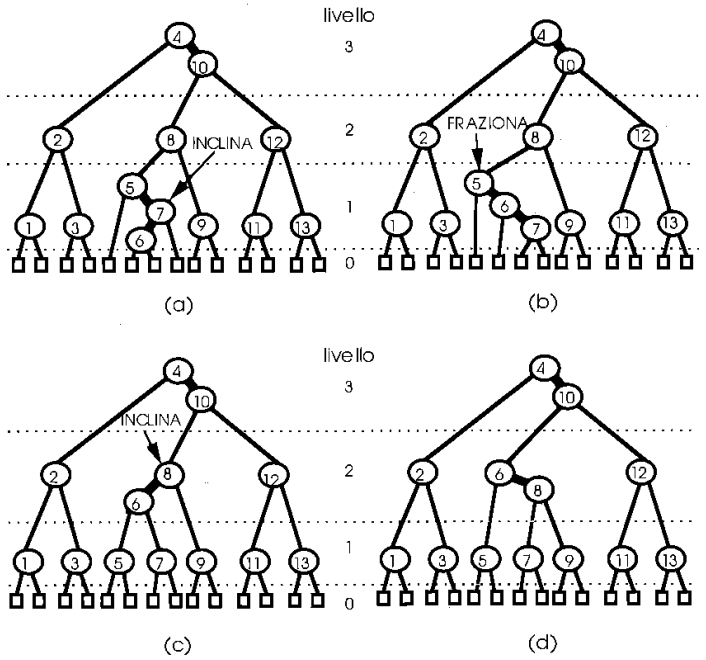


Figura 8.9: Inserzione dell'elemento 6 nel B-albero binario di Fig. 8.6.

CANCELLA:
1. Togli dal livello 1 il nodo con l'elemento da cancellare;
2. Risali il percorso dal nodo rimosso verso la radice e per ogni nodo binario u incontrato:

   2.1. Se manca un supernodo sotto ad u, cioè se il livello di uno dei due figli di u è uguale a quello di u meno due, decrementa di uno il livello di u. Se il figlio destro d di u appartiene allo stesso supernodo di u, decrementa di uno anche il livello di d.

   2.2. Esegui INCLINA(u) seguita da FRAZIONA(u).

Esempio 8.8 (Cancellazione). La Fig. 8.10 mostra l'effetto della cancellazione dell'elemento 1 dal B-albero binario di Fig. 8.6. □

Nel realizzare un B-albero binario con puntatori, non si mantengono esplicitamente le foglie (vuote), ma si usa una sentinella comune in fondo all'albero. Come si è già visto per le liste, l'utilizzo di una sentinella semplifica la scrittura dei programmi in quanto non è necessario effettuare un apposito confronto in più per accorgersi quando è finita la scansione di una sequenza. Grazie alla sentinella, che in questo caso è un nodo fittizio con livello 0, non bisogna verificare l'esistenza di un nodo prima di esaminarne il livello. Infatti ogni nodo senza un figlio anziché avere il valore **nil** nell'apposito campo punterà alla sentinella. In questo modo, ogni volta che durante

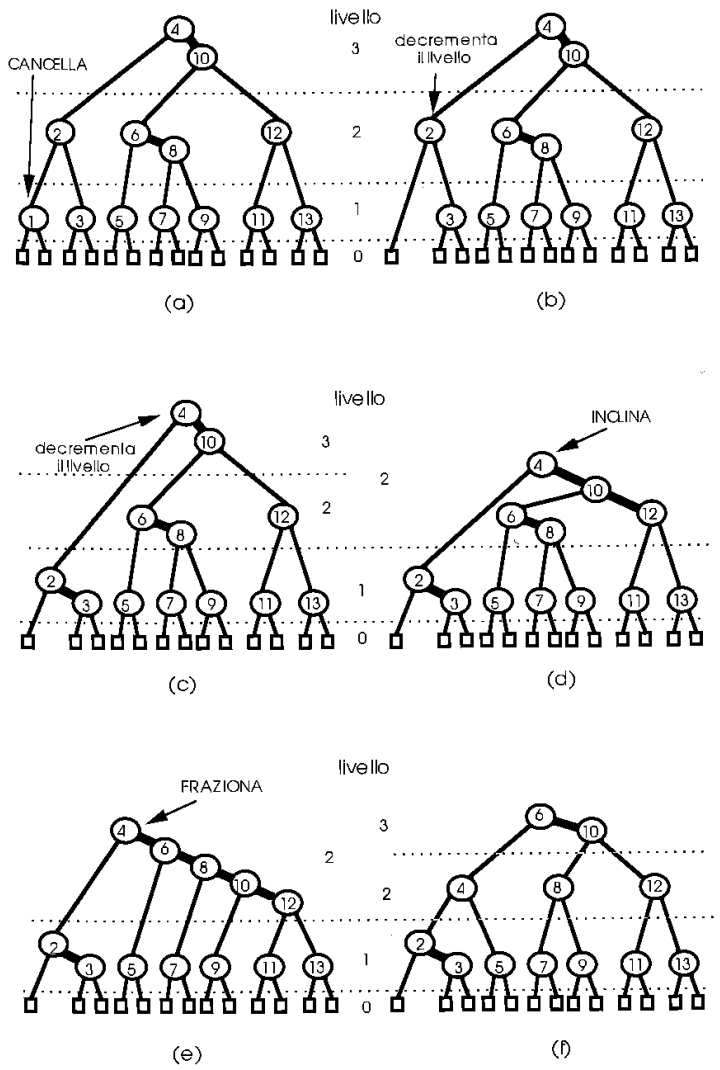


Figura 8.10: Cancellazione dell'elemento 1.

una discesa si cerca di uscire dall'albero per leggere il livello di un nodo ci si imbatte nella sentinella. È bene notare che più alberi possono condividere la stessa sentinella. Per non appesantire la presentazione delle procedure, è usata per sentinella una variabile globale *fondo*, che deve essere opportunamente inizializzata. Le dichiarazioni dei tipi di dato e la procedura Pascal per l'inizializzazione sono le seguenti.

```
type insieme = binalbero;
    binalbero = ↑ nodo
    nodo = record
        sinistro, destro: binalbero;
        livello: integer;
        elemento: tipoelem
    end;

var fondo, cancellato, ultimo: binalbero;

procedure INIZIALIZZAGLOBALI;
    begin
        new(fondo);
        fondo↑.livello := 0;
        fondo↑.sinistro := fondo; fondo↑.destro := fondo; cancellato := fondo
    end;
```

Sono impiegate anche altre due variabili globali *cancellato* e *ultimo*, per cancellare un generico nodo (non necessariamente al livello 1). Queste variabili sono usate durante una discesa in un percorso dalla radice verso una foglia nel modo seguente. In ogni nodo binario è effettuato un confronto tra l'elemento da cancellare e quello contenuto nel nodo. Se la chiave da cancellare è più piccola, allora si scende a sinistra, altrimenti si scende sempre a destra (cioè anche se l'elemento da cancellare è presente nel nodo). La variabile *ultimo* punta via via all'ultimo nodo interno (cioè non foglia) del percorso, mentre *cancellato* punta via via all'ultimo nodo dal quale si è scesi a destra. Quando è raggiunta la sentinella *fondo*, *cancellato* punta al nodo contenente l'elemento da cancellare (se esiste), mentre *ultimo* punta al nodo da eliminare (ovviamente, può accadere che i due nodi coincidano, qualora l'elemento da cancellare si trovi proprio al livello 1). A questo punto, se l'elemento da cancellare è presente nel nodo *cancellato*, si sposta l'elemento dal nodo *ultimo* al nodo *cancellato* e si elimina il nodo *ultimo*, riducendosi al caso di cancellazione di un nodo al livello 1 visto precedentemente.

Si noti che le seguenti procedure ricorsive Pascal INSERISCI e CANCELLA non utilizzano mai i puntatori ai padri (che infatti non sono stati neanche dichiarati). Infatti, aggiustano il bilanciamento dell'albero, con le operazioni INCLINA e FRAZIONA sistemate prima dell'**end** finale, durante la risalita dalla foglia verso la radice che avviene via via che rientrano le chiamate ricorsive sospese.

```
procedure INCLINA(var u: binalbero);
    var temp: binalbero;
    begin
        if u↑.sinistro↑.livello = u↑.livello then begin
            temp := u; u := u↑.sinistro;
            temp↑.sinistro := u↑.destro; u↑.destro := temp
        end
    end;
```

```
procedure FRAZIONA(var u: binalbero);
    var temp: binalbero;
    begin
        if u↑.destro↑.destro↑.livello = u↑.livello then begin
            temp := u;
            u := u↑.destro;
            temp↑.destro := u↑.sinistro;
            u↑.sinistro := temp;
            u↑.livello := u↑.livello + 1;
        end
    end;

procedure INSERISCI(x: tipoelem; var u: insieme);
    begin
        if u = fondo then begin
            new(u);
            u↑.elemento := x;
            u↑.sinistro := fondo;
            u↑.destro := fondo;
            u↑.livello := 1
        end else begin
            if x < u↑.elemento then
                INSERISCI(x, u↑.sinistro)
            else if x> u↑.elemento then
                INSERISCI(x, u↑.destro);
            INCLINA(u);
            FRAZIONA(u)
        end
    end;

procedure CANCELLA(x: tipoelem; var u: insieme);
    begin
        if u↑ fondo then begin
            {Fase 1: discendi nell'albero impostando ultimo e cancellato}
            ultimo := u;
            if x< u↑.elemento then
                CANCELLA(x, u↑.sinistro)
            else begin
                cancellato := u;
                CANCELLA(x, u↑.destro)
            end;
            {Fase 2: giunto in fondo all'albero togli l'elemento, se c'è}
            if (u = ultimo) and (cancellato ≠ fondo) and (x = cancellato↑.elemento)
                then begin
                    cancellato↑.elemento := u↑.elemento;
                    cancellato := fondo;
                    u := u↑.destro;
                    dispose(ultimo)
                end
```



```
            {Fase 3: ribilancia durante la risalita}
            else if (u↑.sinistro↑.livello < u↑.livello − 1) or
                    (u↑.destro↑.livello < u↑.livello − 1)
                then begin
                    u↑.livello := u↑.livello − 1;
                    if u↑.destro↑.livello > u↑.livello then
                        u↑.destro↑.livello := u↑.livello;
                    INCLINA(u);
                    INCLINA(u↑.destro);
                    INCLINA(u↑.destro↑.destro);
                    FRAZIONA(u);
                    FRAZIONA(u↑.destro)
                end;
        end;
    end;
```

La procedura CANCELLA è quella un po' più complicata. Essa consta di tre fasi. Nella prima fase si discende dalla radice verso una foglia, effettuando i confronti con gli elementi contenuti nei nodi e impostando le variabili globali *cancellato* e *ultimo* come descritto precedentemente. Nella seconda fase viene cancellato l'elemento, se presente, nel qual caso è rimosso il nodo puntato da *ultimo*. La terza fase avviene durante la risalita e provvede al ribilanciamento del B-albero binario con le operazioni INCLINA e FRAZIONA richiamate per ogni nodo del percorso di risalita.

Si noti che nella procedura INSERISCI avviene una sola chiamata di INCLINA e una sola di FRAZIONA, mentre nella procedura CANCELLA sono necessarie tre chiamate di INCLINA e due di FRAZIONA. Si può dimostrare che queste chiamate sono sufficienti a garantire la correttezza delle procedure INSERISCI e CANCELLA.

Esempio 8.9 (Procedure INCLINA e FRAZIONA). Con riferimento alla Fig. 8.10d), INCLINA(u) è effettuata sul nodo che contiene 4, INCLINA(u↑.*destro*) sul nodo contenente 10, e INCLINA(u↑.*destro*↑.*destro*) su quello con 8. Infatti quest'ultimo nodo è divenuto figlio destro di quello con 6, che a sua volta è diventato figlio destro di quello con 4, dopo la rotazione destra causata dalla INCLINA precedente. Successivamente, con riferimento alla Fig. 8.10e), FRAZIONA(u) è ancora eseguita sullo stesso nodo contenente 4, ma FRAZIONA(u↑.*destro*) su quello con 8 perché, per effetto della rotazione sinistra dovuta alla FRAZIONA precedente, il nuovo nodo puntato da u diventa quello che contiene 6. □

Le procedure INCLINA e FRAZIONA richiedono tempo $O(1)$. Poiché le procedure INSERISCI e CANCELLA discendono seguendo un percorso radice-foglia lungo $O(\log n)$ e richiamano un numero costante di INCLINA e FRAZIONA durante la risalita, la loro complessità è $O(\log n)$. Anche le altre operazioni APPARTIENE, MIN, MAX, CANCELLAMIN, CANCELLAMAX possono ovviamente essere descritte con procedure Pascal che richiedono la stessa complessità e sono lasciate per esercizio.

8.4 B-alberi

Esistono generalizzazioni degli alberi 2-3 (e quindi dei B-alberi binari): i "B-alberi" veri e propri (non binari). Il loro nome deriva forse dall'iniziale di Bayer che li ha inventati insieme a McCreight negli anni '70 o piuttosto dall'iniziale del termine "bilanciato". In un B-albero, fissato un intero $m \geq 2$, ogni nodo interno, esclusa la radice, può avere k figli, con $m \leq k \leq 2m$, mentre la radice ne ha $2 \leq k \leq 2m$. Tutte

le foglie sono vuote e stanno allo stesso livello. In tal caso, è facile verificare che il livello L delle foglie verifica la relazione $\lceil \log_{2m}(n+1) \rceil \leq L \leq \lceil \log_m(n+1) \rceil$, dove n è al solito il numero di elementi memorizzati. Per esaminare gli elementi di un singolo nodo risulta conveniente effettuare una ricerca binaria, che costa tempo $O(\log m)$.

Per inserire un elemento, si procede come per gli alberi 2-3. Se il nodo al livello $L-1$ dove inserire contiene meno di $2m-1$ elementi, allora vi si effettua direttamente l'inserzione, aggiungendo inoltre una nuova foglia. Se invece il nodo contiene già $2m-1$ elementi, allora lo si sdoppia in due nodi fratelli contenenti rispettivamente gli $m-1$ elementi più piccoli e gli m elementi più grandi, e inserendo nel padre l'elemento rimanente. Nel caso pessimo, lo sdoppiamento si propaga fino alla radice, incrementando così L di uno.

Per eliminare un elemento x presente nel B-albero, si individua il nodo u che lo contiene. Si possono presentare 3 casi, (a), (b) e (c), il secondo dei quali si può distinguere in quattro ulteriori sottocasi (b1), (b2), (b3) e (b4):

a. se u è al livello $L-1$, cioè i suoi figli sono foglie, e contiene almeno m elementi, allora si cancella x e un figlio di u;
b. se u è al livello $L-1$, è l'h-esimo figlio di suo padre, e contiene $m-1$ elementi, allora:
   - b1. se il fratello precedente di u ha almeno m elementi si ridistribiscono gli elementi tra i due fratelli e il padre: x è eliminato, l'elemento maggiore del fratello precedente passa nel padre e l'elemento $(h-1)$-esimo del padre passa nel nodo u;
   - b2. se il fratello precedente di u non ha almeno m elementi ma il fratello successivo ne ha almeno m, allora l'elemento minore del fratello successivo passa nel padre e l'elemento h-esimo del padre scende in u;
   - b3. se né il fratello precedente di u né il successivo ha almeno m elementi, ma il padre ce n'ha almeno m, allora si elimina uno dei fratelli ed u include, oltre agli elementi che aveva già (escluso ovviamente x), gli $m-1$ elementi del fratello più l'elemento di valore intermedio che scende dal padre;
   - b4. se né il fratello precedente, né il successivo, né il padre, ha almeno m elementi, occorre propagare la ridistribuzione degli elementi agli "antenati"; può accadere che la radice abbia un solo elemento che scende nell'unico figlio che risulta dalla fusione dei due vecchi figli della radice, aventi entrambi $m-1$ elementi; in tal caso, il livello L delle foglie diminuisce di uno e il nodo fuso diventa la nuova radice, che contiene $2m-2$ elementi ($m-2$ suoi, perché uno è stato cancellato, $m-1$ del vecchio fratello, e 1 sceso dalla vecchia radice);
c. se u è a livello minore di $L-1$ ci si riduce al caso (a) o al caso (b) sostituendo l'elemento cancellato x con il più piccolo elemento y tale che $y > x$ (se x è l'h-esimo elemento di u, y si trova nel nodo "più a sinistra" di livello $L-1$ che si raggiunge scendendo per il figlio $(h+1)$-esimo di u).

Esempio 8.10 (Cancellazione B-alberi). I casi (b2), (b3), (b4) e (c) per la cancellazione di un elemento in un B-albero con $m = 2$ sono illustrati nella Fig. 8.11. □

I B-alberi sono adatti alla memorizzazione di dizionari in memoria secondaria, il cui tempo di accesso è molto elevato rispetto a quello della memoria principale. Infatti, un'opportuna scelta di m permette di memorizzare un nodo in una pagina, che può essere trasferita interamente dalla memoria secondaria a quella principale con un solo accesso, e di effettuare nella memoria principale la ricerca binaria all'interno

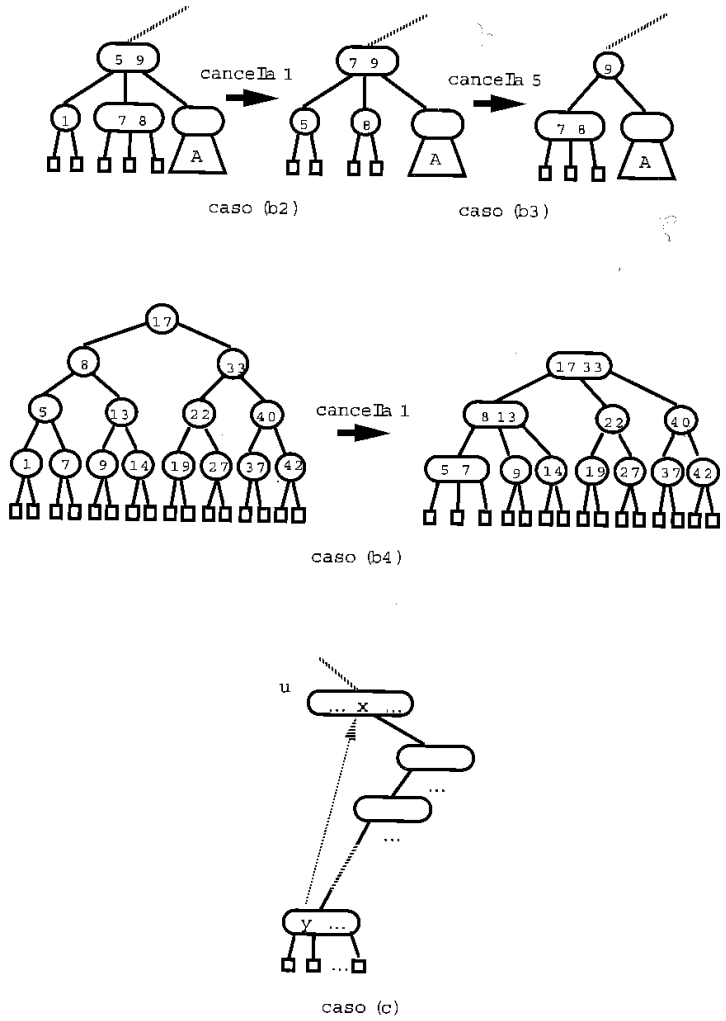


Figura 8.11: Cancellazione di un elemento da un B-albero.

del nodo. In questo modo, se per esempio il numero di elementi del dizionario è 10000 ed il numero di elementi per nodo è 100, bastano $\log_{100} 10000 = 2$ accessi alla memoria secondaria per ricercare un elemento.

8.5 Esercizi

Esercizio 8.1. Si scriva una funzione Pascal che verifichi se un albero binario è un albero di ricerca usando gli operatori degli alberi binari.

Esercizio 8.2. Si scrivano le procedure Pascal per gli operatori INSERISCI e CANCELLA usando un albero binario di ricerca realizzato con puntatori.

Esercizio 8.3 (Stampa intervallo). Dato un albero binario di ricerca i cui nodi contengono interi positivi, si scriva una procedura efficiente che usi gli operatori degli alberi binari e stampi, in ordine crescente, tutte le chiavi che appartengono ad un dato intervallo $[a, b]$.

L'algoritmo si basa su una visita simmetrica in cui vengono scartati i sottoalberi con chiavi non comprese nell'intervallo $[a, b]$. La sua complessità è $O(h + k)$, dove h è la profondità dell'albero e k è il numero di chiavi che appartengono ad $[a, b]$. Infatti, occorre tempo $O(h)$ per individuare la chiave più piccola e quella più grande in $[a, b]$ e tempo $O(k)$ per visitare tutte le chiavi in $[a, b]$.

```
procedure STAMPANODI(u: nodo; var T: binalbero; a, b: integer);
    begin
        if ((not(SINISTROVUOTO(u, T))) and (a < LEGGINODO(u, T)))
            then STAMPANODI(FIGLIOSINISTRO(u, T), T, a, b);
        if ((a ≤ LEGGINODO(u, T)) and (LEGGINODO(u, T) ≤ b))
            then write(LEGGINODO(u, T), ' ');
        if ((not(DESTROVUOTO(u, T))) and (LEGGINODO(u, T) < b)) then
            STAMPANODI(FIGLIODESTRO(u, T), T, a , b)
    end;
```

Esercizio 8.4 (Successore). Dato un albero binario di ricerca, realizzato con puntatori e i cui nodi contengono interi, si scrivano le funzioni Pascal MINIMO(u), che restituisce il nodo con chiave minima nel sottoalbero radicato in u, e SUCCESSORE(u), che restituisce il nodo con la più piccola chiave maggiore della chiave contenuta in u.

Trattandosi di un albero binario di ricerca, si determinano i nodi richiesti senza confronti fra chiavi. La funzione Pascal MINIMO assume che l'albero radicato in u non sia vuoto e trova il risultato seguendo sempre i figli sinistri fino a che è trovato un puntatore sinistro a **nil**. Poiché l'albero è realizzato con puntatori, si usa il solo parametro u per denotare sia un nodo che il sottoalbero radicato in tal nodo.

```
function MINIMO(u: nodo): nodo;
    begin
        if u↑.sinistro ≠ nil then
            MINIMO := MINIMO(u↑.sinistro)
        else
            MINIMO := u
    end;
```

SUCCESSORE(u) restituisce il nodo che contiene il minimo valore nel sottoalbero destro di u, se tale sottoalbero non è vuoto. Altrimenti restituisce il primo antenato

(salendo verso la radice) di u il cui figlio sinistro è pure un antenato di u. Se tale antenato non esiste, perché u contiene la chiave massima, allora è restituito **nil**.

```
function SUCCESSORE(u: nodo): nodo;
    var a: nodo;
    begin
        if u↑.destro ≠ nil then
            SUCCESSORE := MINIMO(u↑.destro)
        else begin
            a := u↑.genitore;
            while (a ≠ nil) and (u = a↑.destro) do
                begin
                    u := a;
                    a := a↑.genitore
                end;
            SUCCESSORE := a
        end
    end;
```

Entrambe le funzioni hanno complessità $O(h)$, dove h è l'altezza dell'albero.

Esercizio 8.5 (Invisita iterativa). Dato un albero binario di ricerca realizzato con puntatori e i cui nodi contengono interi, si scriva una versione Pascal iterativa della visita simmetrica senza usare una pila.

Si usano le funzioni MINIMO e SUCCESSORE dell'esercizio precedente, poiché una visita simmetrica di un albero binario di ricerca T visita le chiavi in ordine crescente. Una versione iterativa si ottiene immediatamente visitando per primo il nodo restituito da MINIMO(T) e poi richiamando iterativamente la SUCCESSORE finché non si giunge al nodo che contiene la chiave massima (e che ha **nil** come successore).

```
procedure INVISITA(T: binalbero);
    var u: nodo;
    begin
        u := MINIMO(T);
        while (u ≠ nil) do begin
            {esamina il nodo u}
            u := SUCCESSORE(u);
        end;
    end;
```

Nonostante INVISITA chiami per ogni nodo una procedura di complessità $O(h)$, la complessità della visita è $\Theta(n)$ e non $O(nh)$. Infatti la funzione SUCCESSORE attraversa il minimo numero di nodi necessari a raggiungere il successore di u. Pertanto, la sequenza di n chiamate nella INVISITA attraversa ciascuna coppia padre-figlio esattamente una volta scendendo dal padre al figlio e ciascuna coppia figlio-padre esattamente una seconda volta risalendo dal figlio al padre.

Esercizio 8.6 (Numero INCLINA e FRAZIONA). Si dimostri che le procedure INSERISCI e CANCELLA per i B-alberi binari sono corrette, mostrando che una INCLINA e una FRAZIONA su ciascun nodo del percorso di risalita sono sufficienti per la INSERISCI, mentre tre INCLINA e due FRAZIONA sono sufficienti per la CANCELLA.

Esercizio 8.7 (Livello nodi B-alberi binari). Si dimostri che non è necessario mantenere il livello nei nodi di un B-albero binario, ma che un singolo bit per nodo è sufficiente (purtroppo in tal caso le procedure di inserzione e cancellazione si complicano notevolmente).

Capitolo 9

Grafi

Un grafo orientato (o grafo diretto, o digrafo) è una coppia $G = (N, A)$, con N insieme finito di elementi, detti nodi (o vertici, o punti), ed A insieme finito di coppie ordinate di nodi, detti archi (o spigoli, o linee). I grafi orientati possono essere rappresentati graficamente disegnando ogni nodo con un cerchietto e ogni arco (i, j) con una freccia che esce dal nodo i ed entra nel nodo j. Il numero di nodi e il numero di archi sono di solito indicati, rispettivamente, con n ed m. Poiché possono esserci al più $n - 1$ archi uscenti da ciascun nodo, risulta $0 \leq m \leq n(n-1)$.

Esempio 9.1 (Grafo orientato). $G = (N, A)$ con $N = \{1, 2, 3, 4\}$ ed $A = \{(1, 2), (1, 3), (2, 4), (3, 2), (4, 2), (4, 3)\}$ è un grafo orientato, ove $n = |N| = 4$ ed $m = |A| = 6$. G può essere rappresentato come mostrato nella Fig. 9.1. □

In generale, i nodi sono usati per rappresentare oggetti e gli archi per rappresentare relazioni tra coppie di oggetti.

Esempio 9.2 (Linee aeree). I nodi possono rappresentare aeroporti e gli archi collegamenti aerei tra aeroporti. □

In un grafo orientato G, un cammino è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tali che $(u_i, u_{i+1}) \in A$, per $i = 0, 1, 2, \ldots, k-1$. Il cammino parte dal nodo u_0, attraversa i nodi $u_1, \ldots, u_k - 1$, arriva al nodo u_k, ed ha lunghezza uguale al numero k di archi. Se non ci sono nodi ripetuti, cioè $u_i \neq u_j$ per $0 \leq i < j \leq k$, allora il cammino è semplice, mentre se $u_0 = u_k$, allora è chiuso. Un cammino sia semplice che chiuso, dove l'unica ripetizione è $u_0 = u_k$, è detto ciclo.

Esempio 9.3 (Cammini). Nel grafo G di Fig. 9.1, la sequenza 1, 2, 4, 3, 2 è un cammino, 1, 2, 4 è un cammino semplice, e 2, 4, 3, 2 è un ciclo. □

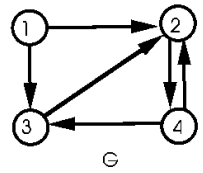


Figura 9.1: Un grafo orientato.

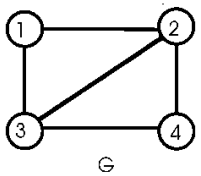


Figura 9.2: Un grafo non orientato.

Un grafo orientato è fortemente connesso se per ogni coppia di nodi u e v esiste almeno un cammino da u a v ed almeno un cammino da v ad u.

Esempio 9.4 (Forte connessione). Il grafo G di Fig. 9.1 non è fortemente connesso. Infatti esiste un cammino da 1 a 4, ma non da 4 a 1. □

Talvolta, gli archi di un grafo G sono formati da coppie non ordinate $[i, j]$. In tal caso, G è un grafo non orientato. Mentre in un grafo orientato (i, j) e (j, i) indicano due archi distinti, in uno non orientato $[i, j]$ e $[j, i]$ indicano lo stesso arco, che è detto incidere sui due nodi, i quali sono adiacenti tra loro. Grafi non orientati sono usati per rappresentare relazioni simmetriche tra oggetti.

Esempio 9.5 (Grafo non orientato). Il grafo non orientato $G = (N, A)$ con $N = \{1, 2, 3, 4\}$ e $A = \{[1, 2], [1, 3], [2, 4], [3, 2], [4, 3]\}$ è mostrato nella Fig. 9.2. I nodi 1 e 3 sono adiacenti, ma 1 e 4 non lo sono. □

Esempio 9.6 (Cabala). Il biblico "Legno (o Albero) della Vita", con i suoi 10 Sefiroth e i 22 Sentieri, è rappresentato nella tradizione cabalistica con il grafo non orientato mostrato all'inizio di questo testo (p.2). □

Anche nei grafi non orientati ci sono nozioni analoghe a quelle di cammino e ciclo. In un grafo non orientato G, una catena è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tali che $[u_i, u_{i+1}] \in A$, per $i = 0, 1, 2, \ldots, k-1$. La catena collega u_0 con u_k, o viceversa, ed ha lunghezza uguale a k. Se non ci sono nodi ripetuti, allora la catena è semplice, mentre se $u_0 = u_k$, allora è chiusa. Una catena sia semplice che chiusa i cui archi sono tutti distinti è un circuito. Si noti che nei grafi orientati la mancanza di ripetizioni di nodi implica la mancanza di ripetizione degli archi, ma questo non è vero per i grafi non orientati.

Esempio 9.7 (Catene e circuiti). Nel grafo G di Fig. 9.2, la sequenza 1, 2, 4, 2, 3, 1 è una catena chiusa e 2, 4, 3, 2 è un circuito. La sequenza 1, 2, 1 è una catena chiusa ma non un circuito, benché non ci siano nodi ripetuti, perché è ripetuto l'arco [1, 2]. □

Un grafo non orientato è connesso se per ogni coppia di nodi distinti u e v esiste una catena tra u e v.

Esempio 9.8 (Connessione). Il grafo G di Fig. 9.2 è connesso. □

Un grafo non orientato è un albero libero se è connesso e per ogni coppia di nodi esiste una e una sola catena semplice. È facile verificare che una definizione equivalente di albero libero richiede, oltre alla connessione, che il numero di archi sia minimo (cioè uguale al numero di nodi meno uno) oppure che non esista alcun circuito. Un albero radicato è ottenuto da uno libero stabilendo un ordinamento "verticale" tra i nodi: un nodo r è designato arbitrariamente come radice e i nodi rimanenti sono

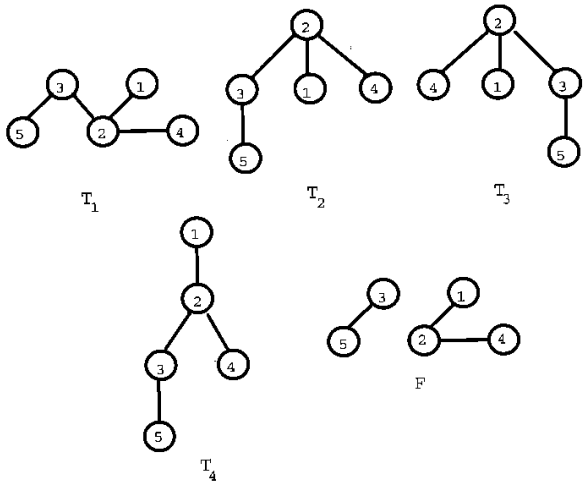


Figura 9.3: Alberi liberi, radicati e foreste.

ordinati per livelli, cioè in base alla loro distanza dalla radice. Dato un albero libero di n nodi, da questo si possono ricavare pertanto n alberi radicati tra loro distinti. Un albero ordinato è ottenuto da uno radicato stabilendo un ordinamento "orizzontale" tra i nodi che stanno allo stesso livello, come discusso nel § 4.

Esempio 9.9 (Alberi liberi e radicati). Il grafo G di Fig. 9.2 non è un albero libero perché ha 4 nodi e più di 3 archi. Invece, la Fig. 9.3 mostra quattro alberi $T_1, \ldots, T_4$. Visti come alberi liberi, essi sono tutti identici tra loro. T_1 non è radicato, mentre gli altri lo sono. Visti come alberi radicati, T_2 e T_3, avendo entrambi il nodo 2 come radice, sono identici tra loro, ma sono diversi da T_4 che è radicato in 1. La Fig. 9.3 mostra anche un grafo F che non è un albero perché ha 5 nodi, ma meno di 4 archi e non è neanche connesso. Essendo formato da un insieme di alberi, F è una foresta. □

In molti problemi pratici, ci si imbatte in grafi che sono caratterizzati da proprietà particolari, che risultano utili per risolvere il problema stesso.

Esempio 9.10 (Chi ha ucciso il duca McPollock?). Sherlock Holmes è stato incaricato di indagare sulla misteriosa morte del vecchio duca McPollock, ucciso da un'esplosione nel suo maniero scozzese. Il Dr. Watson si presenta a rapporto da Holmes in Baker Street:

Watson: "Ci sono novità, Holmes: pare che il testamento, andato distrutto nell'esplosione, fosse stato favorevole ad una delle sette 'amiche' del duca."

Holmes: "Ciò che è più strano, è che la bomba sia stata fabbricata appositamente per essere nascosta nell'armatura della camera da letto, il che fa supporre che *l'assassino abbia necessariamente fatto più di una visita al castello.*"

Watson: "Ho interrogato personalmente le sette donne, ma ciascuna ha giurato di essere stata nel castello *una sola volta nella sua vita.* Dagli interrogatori risulta che:

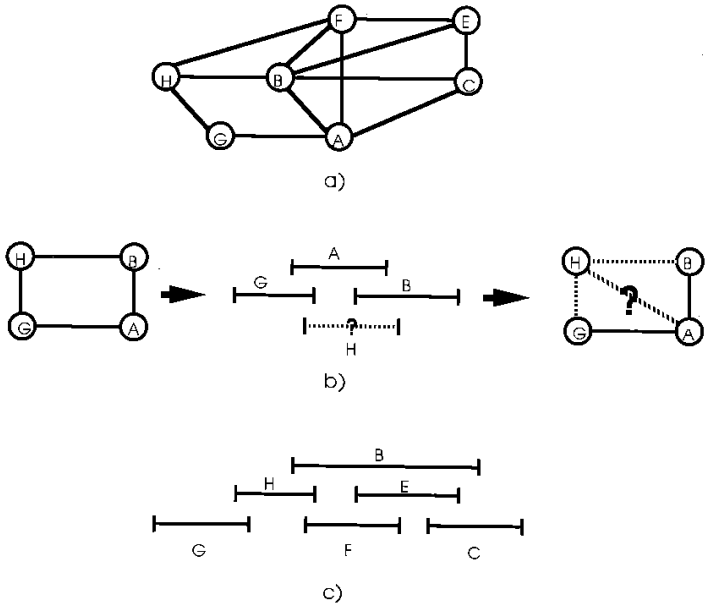


Figura 9.4: a) Il grafo degli incontri. b) Rappresentazione di circuiti di lunghezza 4 come intersezione di intervalli della retta. c) Possibili periodi di soggiorno al castello cancellando il nodo A.

Ann ha incontrato Betty, Charlotte, Felicia e Georgia;
Betty ha incontrato Ann, Charlotte, Edith, Felicia e Helen;
Charlotte ha incontrato Ann, Betty e Edith;
Edith ha incontrato Betty, Charlotte, Felicia;
Felicia ha incontrato Ann, Betty, Edith, Helen;
Georgia ha incontrato Ann e Helen;
Helen ha incontrato Betty, Felicia e Georgia.

Vedete, Holmes, che le testimonianze concordano. Ma chi sarà l'assassino?"

Holmes: "Elementare, mio caro Watson: ciò che mi avete detto individua inequivocabilmente l'assassino!"

Soluzione di Holmes: Il grafo degli incontri tra le donne, indicate con le loro iniziali, è disegnato nella Fig. 9.4a). Se ogni donna avesse detto la verità, questo grafo dovrebbe rappresentare l'intersezione di 7 intervalli della retta reale. In tal caso, però, ogni circuito di lunghezza maggiore o uguale a 4 dovrebbe possedere un arco che collegasse due nodi non consecutivi nel circuito, come esemplificato nella Fig. 9.4b). Il grafo degli incontri ha ben 3 circuiti che violano questa proprietà: (1) A, G, H, F, A, (2) A, G, H, B, A e (3) A, F, E, C, A. L'unico nodo che compare in tutti e tre i circuiti è il nodo A: l'assassino è dunque Ann! Infatti, cancellando A dal grafo, i

periodi di soggiorno nel castello delle altre 6 donne potrebbero essere quelli illustrati nella Fig. 9.4c). □

9.1 Specifica

Poiché ogni grafo non orientato G può essere visto come un grafo orientato D, ottenuto da G sostituendo ogni arco $[i, j]$ con i due archi (i, j) e (j, i), sarà fornita una specifica solo per grafi orientati, che saranno chiamati semplicemente grafi. Le operazioni definite sui grafi prevedono la possibilità di inserire o cancellare nodi o archi, e di spostarsi da un nodo ad un suo adiacente.

CREAGRAFO:	() → grafo
GRAFOVUOTO:	(grafo) → booleano
INSNODO:	(nodo, grafo) → grafo
INSARCO:	(nodo, nodo, grafo) → grafo
CANCNODO:	(nodo, grafo) → grafo
CANCARCO:	(nodo, nodo, grafo) → grafo
ADIACENTI:	(nodo, grafo) → insieme

L'operazione CREAGRAFO restituisce un grafo "vuoto", cioè senza nodi e senza archi, mentre GRAFOVUOTO verifica se il grafo è vuoto oppure no. L'operazione INSNODO aggiunge al grafo un nuovo nodo "isolato" (cioè che non è collegato a nessun altro nodo del grafo), mentre INSARCO aggiunge un nuovo arco tra una coppia di nodi del grafo. CANCARCO permette di cancellare un arco esistente tra una coppia di nodi, mentre CANCNODO cancella un nodo "isolato". Se il nodo da cancellare non è isolato, occorre preventivamente cancellare tutti gli archi che lo collagano ad altri nodi con ripetute applicazioni di CANCARCO. Infine, l'operazione ADIACENTI restituisce l'insieme di tutti i nodi adiacenti ad un certo nodo, cioè tutti i nodi raggiungibili con un arco che esce da tale nodo.

CREAGRAFO = G
  Post: $G = (N, A)$, con $N = \emptyset$ ed $A = \emptyset$
GRAFOVUOTO(G) = b
  Post: b = vero, se $N = \emptyset$ e $A = \emptyset$, b = falso, altrimenti
INSNODO(u, G) = G'
  Pre: $G = (N, A)$, $u \notin N$
  Post: $G' = (N', A)$, $N' = N \cup \{u\}$
INSARCO(u, v, G) = G'
  Pre: $G = (N, A)$, $u \in N$, $v \in N$, e $(u, v) \notin A$
  Post: $G' = (N, A')$, $A' = A \cup \{(u, v)\}$
CANCNODO(u, G) = G'
  Pre: $G = (N, A)$, $u \in N$, non esiste alcun nodo v tale che $(u, v) \in A$ o $(v, u) \in A$
  Post: $G' = (N', A)$, $N' = N - \{u\}$
CANCARCO(u, v, G) = G'
  Pre: $G = (N, A)$, $u \in N$, $v \in N$, e $(u, v) \in A$
  Post: $G' = (N, A')$, $A' = A - \{(u, v)\}$
ADIACENTI(u, G) = $A(u)$
  Pre: $G = (N, A)$, $u \in N$,
  Post: $A(u) = \{v\colon (u, v) \in A\}$, detto insieme di adiacenza di u

In molte applicazioni, ai nodi e/o agli archi sono associate delle informazioni di un certo tipo (dette etichette, o pesi). In tal caso, si parla di grafi etichettati (o pesati), sui quali è utile introdurre ovvi operatori, che qui non stiamo a definire, per reperire e/o modificare le informazioni associate a nodi e/o archi.

Nel seguito di questo capitolo, eviteremo di fornire realizzazioni dettagliate delle operazioni appena definite sul tipo di dato grafo. Il motivo di questa scelta è duplice. Innanzitutto, un grafo è univocamente definibile dall'insieme N dei nodi e dagli n insiemi di adiacenza $A(u)$, per ciascun $u \in N$. Realizzazioni efficienti di insiemi tramite vettori e liste sono state trattate ampiamente nei capitoli precedenti ed il lettore può cercare di combinarle per esercizio in modo da fornire realizzazioni efficienti per le operazioni di inserzione e cancellazione di nodi e/o archi. Inoltre, nelle applicazioni considerate in questo testo, i grafi sono usati per rappresentare relazioni tra oggetti che non mutano nel tempo. In altri termini, non si effettuano mai nè cancellazioni nè inserzioni, né di nodi né di archi, ma si esplorano sistematicamente grafi forniti in ingresso, al fine di determinarne sottoinsiemi di nodi e/o di archi che soddisfino certe proprietà. Nelle realizzazioni che saranno proposte si terrà conto di questo aspetto semplificativo e si supporrà senza perdere in generalità che gli n nodi siano individuati dai numeri interi compresi tra 1 ed n.

9.2 Realizzazioni con matrici

Le più semplici rappresentazioni di grafi utilizzano matrici. Tra queste le più usate sono la matrice di incidenza nodi-archi e la matrice di adiacenza nodi-nodi.

La matrice di incidenza nodi-archi è una matrice rettangolare $B = [b_{ik}]$ di dimensioni $n \times m$, dove ciascuna riga rappresenta un nodo e ciascuna colonna rappresenta un arco, tale che:

$$b_{ik} = \begin{cases} -1, & \text{se l'arco } k\text{-esimo esce dal nodo } i, \\ +1, & \text{se l'arco } k\text{-esimo entra nel nodo } i, \\ 0, & \text{altrimenti.} \end{cases}$$

Se il grafo è non orientato, invece, si ha:

$$b_{ik} = \begin{cases} 1, & \text{se l'arco } k\text{-esimo è incidente nel nodo } i, \\ 0, & \text{altrimenti.} \end{cases}$$

Esempio 9.11 (Matrice di incidenza nodi-archi). La matrice di incidenza del grafo orientato di Fig. 9.1 è mostrata nella Fig. 9.5a) (si noti che ogni colonna ha esattamente un +1 e un −1), mentre quella del grafo non orientato di Fig. 9.2 è mostrata nella Fig. 9.5b) (si noti che ogni colonna ha esattamente due 1). □

La matrice di incidenza è molto utile per rappresentare un grafo in certi problemi di PROGRAMMAZIONE LINEARE, nei quali si cerca un vettore x di m incognite (reali o intere) associate agli archi che soddisfi un sistema di equazioni del tipo $Bx = c$, dove c è un vettore di n termini noti interi. Purtroppo, con questa rappresentazione, lo spazio di memoria occupato è sempre $\Theta(nm)$. Inoltre, dato un nodo, non è agevole ricavare l'insieme di adiacenza. Infatti per calcolare un qualsiasi $A(u)$ occorre scandire la riga u di B alla ricerca delle colonne k con elementi $b_{uk} = -1$ e per ciascuna colonna k scandirla per individuare l'indice di riga v tale che $b_{vk} = +1$, per un totale di $\Theta(nm)$ tempo.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} (1,2) & (1,3) & (2,4) & (3,2) & (4,2) & (4,3) \end{array} \\ \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 0 \\ +1 & 0 & -1 & +1 & +1 & 0 \\ 0 & +1 & 0 & -1 & 0 & +1 \\ 0 & 0 & +1 & 0 & -1 & -1 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} [1,2] & [1,3] & [2,4] & [3,2] & [4,2] & [4,3] \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

(a) (b)

Figura 9.5: Matrici di incidenza nodi-archi: a) per il grafo orientato di Fig. 9.1; b) per il grafo non orientato di Fig. 9.2.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Figura 9.6: Matrice di adiacenza nodi-nodi per il grafo orientato di Fig. 9.1.

La matrice di adiacenza nodi-nodi, invece, è una matrice quadrata $E = [e_{ij}]$ di dimensioni $n \times n$, tale che:

$$e_{ij} = \begin{cases} 1, & \text{se } (i,j) \in A, \\ 0, & \text{se } (i,j) \notin A. \end{cases}$$

Se il grafo è non orientato, allora tale matrice è simmetrica, ovvero $e_{ij} = e_{ji}$ per ogni i e j.

Esempio 9.12 (Matrice di adiacenza nodi-nodi). La matrice di adiacenza del grafo di Fig. 9.2 è mostrata nella Fig. 9.6. □

Con questa realizzazione è banale verificare in tempo $O(1)$ se un dato arco è presente nel grafo oppure no. Purtroppo, il tempo per ricavare l'insieme di adiacenza $A(u)$ di un qualsiasi nodo u è sempre $\Theta(n)$, indipendentemente dalla cardinalità di $A(u)$, perché occorre scandire un'intera riga della matrice alla ricerca degli elementi $b_{uv} = 1$. Inoltre, lo spazio di memoria occupato è $\Theta(n^2)$ e il tempo per esaminare tutti gli archi è $\Theta(n^2)$, anche se il grafo è sparso, cioè contiene un numero m di archi che è $O(n)$. Se il grafo è pesato, allora nella matrice E si utilizzano i pesi degli archi al posto degli elementi binari. Se p_{ij} è il peso dell'arco (i, j), allora la matrice E diventa:

$$e_{ij} = \begin{cases} p_{ij}, & \text{se } (i,j) \in A, \\ +\infty \text{ (oppure } -\infty) & \text{se } (i,j) \notin A. \end{cases}$$

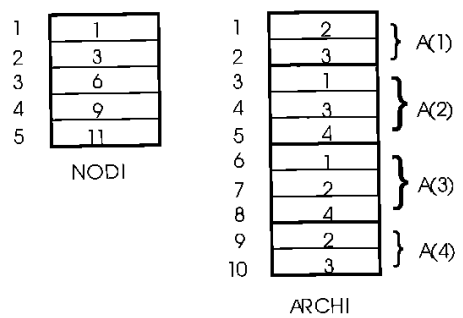


Figura 9.7: Vettori di adiacenza per il grafo non orientato di Fig. 9.2.

9.3 Realizzazioni con insiemi di adiacenza

È spesso conveniente mantenere esplicitamente gli insiemi di adiacenza per ogni nodo del grafo. Una possibilità consiste nell'utilizzare le cosiddette liste di adiacenza. Questa realizzazione è analoga a quella con liste di trabocco viste per i dizionari. Si usa un vettore di dimensione n dove l'i-esima posizione del vettore contiene l'identificativo di una lista, di solito realizzata con puntatori, che memorizza gli elementi di $A(i)$. Lo spazio di memoria è $\Theta(n)$ per il vettore e $\Theta(|A(i)|)$ per ciascuna lista, per un totale di $\Theta(n + \sum_{i \in N} |A(i)|) = \Theta(n + m)$, che è ottimo. Se il grafo è sparso, cioè m è $O(n)$, lo spazio risulta quindi essere $\Theta(n)$ anziché $\Theta(n^2)$. Inoltre, la scansione di ciascun insieme di adiacenza $A(i)$ richiede tempo ottimo $\Theta(|A(i)|)$ mentre la scansione di tutti gli archi del grafo richiede tempo ottimo $\Theta(n + m)$. Purtroppo, anche per verificare se $(i,j) \in A$ occorre una scansione di $A(i)$ che richiede $O(|A(i)|)$ tempo.

Nell'ipotesi semplificativa che non vengano effettuate né inserzioni né cancellazioni di nodi e archi, è possibile rappresentare gli insiemi di adiacenza senza utilizzare liste. Si usano invece due vettori di adiacenza, che chiameremo NODI e ARCHI. Il vettore NODI è formato da n+1 elementi, corrispondenti agli n nodi $1, 2, \ldots, n$ del grafo più la sentinella $n + 1$. NODI[i] contiene un cursore alla posizione del vettore ARCHI a partire dalla quale è memorizzata $A(i)$, di solito in ordine crescente, mentre NODI[$n + 1$] $= m + 1$. Il vettore ARCHI è formato da m elementi, e memorizza, nell'ordine, $A(1), A(2), \ldots, A(n)$. Se $A(i) = \emptyset$, allora NODI[i] = NODI[$i + 1$]. Si noti, in particolare, che se $A(n) = \emptyset$, allora risulta NODI[n] $= m + 1$.

Esempio 9.13 (Vettori di adiacenza). I vettori NODI e ARCHI che rappresentano il grafo di Fig. 9.2 sono mostrati nella Fig. 9.7. Si noti che, essendo il grafo non orientato, ogni arco $[i, j]$ è rappresentato due volte, poiché $i \in A(j)$ e $j \in A(i)$. □

Lo spazio di memoria occupato dai vettori di adiacenza è $\Theta(n + m)$, che è ottimo in ordine di grandezza come nel caso delle liste di adiacenza. Rispetto alle liste di adiacenza, però, c'è un ulteriore risparmio dovuto all'assenza dei campi per i puntatori nelle celle delle liste. Se il grafo è pesato, i pesi sui nodi e/o sugli archi possono essere memorizzati in altri due vettori, p.e. PESINODI e PESIARCHI, rispettivamente

di n ed m elementi. Così, se $(i,j) \in A$ e j = ARCHI[NODI[i] + k], allora il peso dell'arco (i, j) è contenuto in PESIARCHI [NODI[i] + k].

Per semplicità, la scansione di tutti gli elementi di $A(u)$ verrà d'ora in poi descritta informalmente col seguente costrutto:

for each $v \in A(u)$ **do** { elaborazione sul nodo v };

che è un modo più compatto di scrivere il brano Pascal:

```
type grafo = record
        NODI: array[1..n+1] of integer;
        ARCHI: array[1..m] of integer
    end;
    nodo = integer;
var G: grafo;
    ⋮
for i := G.NODI[u] to G.NODI[u + 1] − 1 do begin
        v := G.ARCHI[i];
        {elaborazione sul nodo v};
    end
```

Si noti che per costruzione G.NODI[u] = G.NODI[$u + 1$] se $A(u)$ è vuoto. In tal caso il ciclo **for** non è eseguito essendo G.NODI[u] > G.NODI[$u + 1$] − 1. Inoltre, poiché G.NODI[$n + 1$] − 1 è proprio l'ultima posizione del vettore ARCHI, l'uso della sentinella $n + 1$ evita di introdurre una verifica apposita quando $u = n$.

9.4 Esplorazione di un grafo

Vediamo due metodi sistematici per "visitare" almeno una volta ogni nodo ed ogni arco di un grafo orientato fortemente connesso (o non orientato e connesso). Entrambi i metodi effettuano la visita in tempo ottimo $\Theta(n + m)$ a partire da un generico nodo r:

1. *Depth-First-Search* (DFS), ovvero visita in profondità (detta anche a scandaglio);
2. *Breadth-First-Search* (BFS), ovvero visita in ampiezza (detta anche a ventaglio).

Il metodo di visita DFS è una diretta estensione della visita in ordine anticipato di un albero ordinato. Quando viene visitato un nuovo nodo u, ci si allontana da esso il più possibile lungo un cammino (o catena), visitando nodi (e archi) nel cammino fino a giungere in un "vicolo cieco", ovvero in un nodo v i cui nodi adiacenti sono già stati tutti visitati. Quando si è raggiunto v, si torna indietro lungo l'ultimo arco visitato, e si riprende la visita allontanandosi lungo un altro cammino di nodi (e archi) non ancora visitati.

Nella visita BFS, invece, i nodi sono visitati in ordine di distanza crescente dal nodo di partenza r, dove la distanza da r ad un generico nodo u è il minimo numero di archi in un cammino (o catena) da r ad u. Tale metodo di visita è una estensione di una visita per livelli di un albero radicato, in cui i figli di un nodo sono visitati dopo aver visitato tutti gli altri nodi che stanno allo stesso livello del padre.

In entrambi i metodi, occorre mantenere una lista dei nodi che sono già stati visitati, ma i cui nodi adiacenti non sono ancora stati visitati. Poiché la DFS torna indietro dal più recente vicolo cieco, tale lista è di fatto una pila. D'altro lato, poiché la BFS visita i nodi più vicini ad r prima di quelli più lontani, tale lista è una coda.

Nel seguito, la DFS è descritta con una procedura Pascal ricorsiva, perchè più concisa ed elegante di una iterativa che utilizzi una pila, mentre la BFS è descritta utilizzando gli operatori delle code [cfr. § 3]. Le chiamate delle procedure sono: DFS(G, r) e BFS(G, r).

```
procedure DFS(var G: grafo; u: nodo);
    var v: nodo;
    begin
        {esamina il nodo u e marcalo "visitato"};
        for each v ∈ A(u) do begin
            {esamina l'arco (u, v)};
            if v non è marcato "visitato" then DFS(G, v);
        end;
    end;

procedure BFS(var G: grafo; u: nodo);
    var v: nodo, Q: coda;
    begin
        CREACODA(Q); INCODA(u, Q);
        while not CODAVUOTA(Q) do begin
            u := LEGGICODA(Q); FUORICODA(Q);
            {esamina il nodo u e marcalo "visitato"};
            for each v ∈ A(u) do begin
                {esamina l'arco (u, v)};
                if v non è marcato "visitato" and v ∉ Q then INCODA(v, Q);
            end;
        end;
    end;
```

La verifica di non appartenenza del nodo v alla coda può essere facilmente effettuata in tempo $O(1)$ introducendo nella procedura BFS un opportuno vettore booleano in cui il bit v è impostato a vero quando v è inserito in Q ed è rimesso a falso quando v è estratto da Q. Questa verifica è necessaria per evitare che la coda contenga più di una copia di v, qualora v sia stato precedentemente inserito in coda durante l'esame di un arco (w, v) uscente da un nodo w già marcato visitato. Anche la "marcatura" di ciascun nodo può essere effettuata con un secondo vettore booleano, con ciascun bit u inizializzato a "falso", che poi viene impostato a "vero" quando viene visitato il nodo u.

Poiché ogni nodo del grafo è marcato una sola volta e, per ogni nodo v, viene scandito l'intero insieme $A(v)$ una sola volta, la complessità di entrambe le procedure è $\Theta(n + m)$.

Esempio 9.14 (DFS e BFS). Gli ordini di visita dei nodi e degli archi del grafo mostrato nella Fig. 9.8, assumendo di partire dal nodo $r = 1$ e che gli insiemi $A(i)$ siano memorizzati in ordine crescente, sono i seguenti, dove per chiarezza i nodi sono evidenziati in grassetto:

DFS = **1**, (1,2), **2**, (2,1), (2,4), **4**, (4,1), (4,2), (4,5), **5**, (5,2), (5,4), (2,5), (2,7), **7**, (7,2), (7,8), **8**, (8,1), (8,7), (7,9), **9**, (9,2), (9,7), (2,9), (1,3), **3**, (3,1), (3,6), **6**, (6,1), (6,3), (1,4), (1,6), (1,8);

BFS = **1**, (1,2), (1,3), (1,4), (1,6), (1,8), **2**, (2,1), (2,4), (2,5), (2,7), (2,9), **3**, (3,1), (3,6), **4**, (4,1), (4,2), (4,5), **6**, (6,1), (6,3), **8**, (8,1), (8,7), **5**, (5,2), (5,4), **7**, (7,2), (7,8), (7,9), **9**, (9,2), (9,7). □

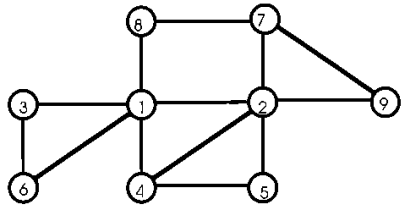


Figura 9.8: Un grafo non orientato

9.4.1 Componenti connesse

Le procedure DFS e BFS appena introdotte sono metodi di visita sistematica che, opportunamente modificati, possono essere usati per risolvere in tempo $\Theta(n + m)$ molti problemi. Tra questi consideriamo soltanto i seguenti:

1. Verificare se un grafo non orientato è connesso oppure no (lo è se e solo se, al termine della visita, tutti i nodi sono stati marcati visitati);
2. Verificare se un grafo orientato è fortemente connesso oppure no (lo è se e solo se, al termine della visita, tutti i nodi sono stati marcati visitati e, rovesciando successivamente il verso di tutti gli archi (i, j) in (j, i) e riapplicando la visita a partire dal medesimo nodo r, tutti i nodi vengono nuovamente marcati visitati);
3. Trovare le componenti connesse di un grafo non orientato G, cioè tutti i sottografi connessi di cui G è composto, tali che nessun sottografo è contenuto in uno più grosso (è sufficiente richiamare iterativamente una DFS modificata in modo da marcare ciascun nodo con il numero della componente connessa cui il nodo appartiene).

Vediamo l'algoritmo per individuare tutte le componenti connesse. Si utilizzano una variabile *numcomp* inizializzata a zero, che conta le componenti connesse ed un vettore COMP tale che COMP[i] è inizializzato a zero per ogni i, $1 \leq i \leq n$, e nel quale sarà registrato il numero della componente connessa alla quale appartiene il nodo i. La visita DFS vera e propria sul nodo i è richiamata in un ciclo ripetuto per $i = 1, 2, \ldots, n$. Se COMP[i] $= 0$, allora i non è stato ancora raggiunto dalla visita e quindi appartiene ad una nuova componente connessa. Viene pertanto incrementata di uno *numcomp* e richiamata la DFS sul nodo i, in modo da assegnare *numcomp* sia ad i che a tutti i nodi raggiungibili da i.

```
procedure COMPONENTICONNESSE(var G: grafo; n: integer);
    var COMP: array[1..maxlung] of integer; numcomp, i: integer;
    begin
        numcomp := 0;
        for i := 1 to n do COMP[i] := 0;
        for i := 1 to n do
        if COMP[i] = 0 then begin
            numcomp := numcomp + 1;
            DFS(G, i)
        end
    end;
```

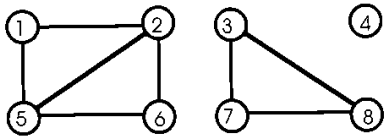


Figura 9.9: Un grafo non orientato con tre componenti connesse.

```
procedure DFS(var G: grafo; u: nodo);
    var v: nodo;
    begin
        COMP[u] := numcomp;
        for each v ∈ A(u) do
            if COMP[v] = 0 then DFS(G, v);
    end;
```

Esempio 9.15 (Componenti connesse). Il grafo di Fig. 9.9 è formato da tre componenti connesse. Eseguendo la procedura COMPONENTICONNESSE l'ordine di visita dei nodi è 1, 2, 5, 6, 3, 7, 8, 4 ed alla fine il vettore COMP contiene: COMP[1] = 1, COMP[2] = 1, COMP[3] = 2, COMP[4] = 3, COMP[5] = 1, COMP[6] = 1, COMP[7] = 2, COMP[8] = 2. □

9.4.2 Alberi di copertura DFS

Gli archi che, mentre sono esaminati in una visita DFS, connettono un nodo marcato ad uno non marcato formano un albero radicato T, detto albero di copertura DFS. Qualora non tutti i nodi siano raggiungibili dalla radice, cioè dal nodo di partenza r, perché il grafo non è fortemente connesso (o non è connesso), allora una visita completa del grafo si ottiene richiamando iterativamente la DFS, come mostrato nella procedura COMPONENTI CONNESSE. In tal caso, la visita forma una foresta di copertura DFS. Se il grafo non è orientato, la visita provoca un orientamento arbitrario a ciascun arco $[i, j]$, dovuto alla direzione in cui è esaminato, mentre se il grafo è orientato viene rispettato l'orientamento.

Un arco che è esaminato passando da un nodo di T ad un altro nodo che è suo antenato in T, è detto arco all'indietro. Se il grafo è non orientato, ciascun arco $[i, j]$ sarà o un arco di T o un arco all'indietro. Un arco che è esaminato passando da un nodo di T ad un suo discendente (che non sia figlio) in T è detto arco in avanti. Se il grafo è orientato, la DFS partiziona i suoi archi in 4 classi: quelli in T, quelli all'indietro, quelli in avanti, e quelli di attraversamento (cioè quelli tra due nodi nessuno dei quali è discendente dell'altro).

Esempio 9.16 (Albero di copertura DFS). La Fig. 9.10 mostra un grafo orientato G e la corrispondente partizione degli archi ottenuta eseguendo una DFS a partire dal nodo 1. L'ordine di visita dei nodi e degli archi è: 1, (1,2), 2, (2,3), 3, (3,1), (1,3), (1,4), 4, (4,2), (4,3). Gli archi (1,2), (2,3) e (1,4) sono in T, l'arco (1,3) è in avanti, l'arco (3,1) è all'indietro, e gli archi (4,2) e (4,3) sono di attraversamento. □

La distinzione tra le 4 classi di archi per i grafi orientati (2 per quelli non orientati) è importante perché permette di definire il seguente schema generale di visita DFS, analogo a quello fornito nel § 4 per gli alberi binari, che può essere poi particolarizzato per risolvere problemi specifici. Nello schema si fa uso di due vettori INIZIO e

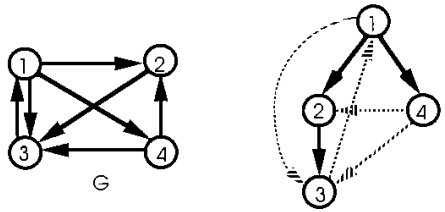


Figura 9.10: Un grafo orientato e la partizione degli archi indotta da una DFS.

FINE e di una variabile globale con le funzioni di orologio discreto. Questa variabile è inizializzata a 0 ed è incrementata di uno sia all'inizio che alla fine della procedura DFS. Tutte le posizioni dei vettori INIZIO e FINE sono inizializzate a "*maxint*" (il più grande intero rappresentabile in Pascal), mentre nelle posizioni INIZIO[u] e FINE[u] sono registrati i valori dell'orologio, rispettivamente, all'inizio e alla fine della procedura di visita sul nodo u.

```
var
    INIZIO: array[1..n] of integer;     {inizializzati tutti a maxint}
    FINE: array[1..n] of integer;       {inizializzati tutti a maxint}
    orologio: integer;                  {inizializzato a 0}

procedure DFS (var G: grafo; u: nodo);
    var v: nodo;
    begin
        {esamina il nodo u per la prima volta (come nella previsita)};
        INIZIO[u] := orologio;
        orologio := orologio + 1;
        for each v ∈ A(u) do begin
            {esamina l'arco (u, v) di qualsiasi tipo};
            if INIZIO[v] = maxint then begin
                {esamina l'arco (u, v) in T};
                DFS(G, v);
                {esamina il nodo u dopo il nodo v (come nella invisita)};
            end else
                if (INIZIO[u] > INIZIO[v]) and (FINE[v] = maxint)
                    then {esamina l'arco (u, v) all'indietro}
                else if (INIZIO[u]<INIZIO[v]) and (FINE[v]<maxint)
                    then {esamina l'arco (u, v) in avanti}
                    else {esamina l'arco (u, v) di attraversamento}
        end;
        {esamina il nodo u prima del ritorno (come nella postvisita)};
        FINE[u] := orologio;
        orologio := orologio + 1;
    end;
```

Confrontando i valori di INIZIO[u], INIZIO[v] e FINE[v] è facile stabilire il tipo dell'arco (u, v). Se INIZIO[v] = *maxint*, allora la visita del nodo v non è ancora iniziata, e quindi (u, v) è in T, mentre se INIZIO[v] < *maxint*, allora (u, v) non è in

T. In quest'ultimo caso, quando INIZIO[u] > INIZIO[v] e FINE[v] = *maxint*, la visita di u è iniziata dopo quella di v che però non è ancora terminata, e pertanto (u, v) è all'indietro; quando invece INIZIO[u] < INIZIO[v] e FINE[v] < *maxint*, la visita di v è sia iniziata che finita dopo quella di u, che però non è ancora terminata, e quindi (u, v) è in avanti; nei rimanenti casi (u, v) è di attraversamento.

In alcuni problemi può essere necessario individuare quali nodi w si trovano nel cammino (o catena) dal nodo r all'attuale nodo u. Tali nodi sono esattamente quelli la cui visita è iniziata ma non ancora terminata, cioè con INIZIO[w] < *maxint* e FINE[w] = *maxint*. Talvolta, può essere utile conoscere i padri dei nodi in T. Per mantenere quest'informazione si può utilizzare un vettore PADRE dove 0 è assegnato a PADRE[r] prima della chiamata nel programma principale, mentre u è assegnato a PADRE[v] subito prima della chiamata ricorsiva interna DFS(G, v). Infine, può risultare utile conoscere il numero ordinale di ogni nodo rispetto alla visita, che si può ottenere con un vettore ORDINALE ed un contatore globale, inizializzato a 0, che viene incrementato di uno ed assegnato ad ORDINALE[u] all'inizio della procedura.

Per concludere, accenniamo alle risoluzioni dei seguenti problemi, ottenibili per esercizio modificando opportunamente lo schema proposto:

1. Verificare se un grafo non orientato è un albero libero oppure no (lo è se e solo se, al termine della visita, tutti i nodi sono stati marcati visitati e, durante la visita, gli unici archi esaminati che non sono in T sono quelli tra un nodo e suo padre in T);
2. Verificare se un grafo orientato è aciclico oppure no (è privo di cicli se e solo se, durante la visita, non è esaminato alcun arco all'indietro);
3. Verificare se un grafo non orientato è bipartito oppure no, cioè se l'insieme N dei nodi può essere partizionato o no in due sottoinsiemi disgiunti tali che nessun arco $[i, j]$ in A abbia entrambi gli estremi i e j nello stesso sottoinsieme della partizione (è bipartito se e solo se i nodi possono essere marcati con "0" o "1" in modo che ogni arco colleghi sempre due nodi marcati con valori diversi).

9.5 Esercizi

Esercizio 9.1 (BFS). Una versione alternativa della visita BFS prevede di esaminare il nodo u e marcarlo visitato immediatamente prima dell'istruzione **while** e di esaminare il nodo v e marcarlo visitato prima di inserirlo nella coda. Si dimostri che in questo caso è possibile inserire v nella coda facendo a meno di verificare che non sia già in coda. Si esegua la nuova procedura di visita BFS sul grafo dell'Esempio 9.14: l'ordine di visita dei nodi e degli archi rimane lo stesso?

Esercizio 9.2 (DFS iterativa). Scrivere una versione iterativa della DFS dove l'ordine di visita dei nodi e degli archi sia lo stesso di quello della versione ricorsiva descritta nel § 9.4 ed eseguirla sul grafo dell'Esempio 9.14.

Esercizio 9.3 (Grafo aciclico). Scrivere una funzione Pascal di complessità ottima per verificare se un grafo orientato è aciclico oppure no.

Un grafo è privo di cicli se e solo se, durante la visita, non è esaminato alcun arco all'indietro. Per riconoscere un arco all'indietro, invece dei due vettori INIZIO e FINE, si possono usare due vettori ORDINALE e CAMMINO rispettivamente di interi e di booleani. ORDINALE[v] memorizza il numero d'ordine di inizio visita del nodo v, mentre CAMMINO[v] è "vero" se v è nel cammino da r ad u, ovvero se la visita di v è iniziata ma non ancora finita. Pertanto un arco (u, v) è all'indietro se e solo se

CAMMINO[v] = "vero" ed ORDINALE[v] < ORDINALE[u]. La complessità è $\Theta(n + m)$.

```
function ACICLICO(var G: grafo): boolean;
    var
        CAMMINO: array [1..n] of boolean;
        ORDINALE: array[1..n] of integer;
        id: integer;
        esisteciclo: boolean
    begin
        id := 0;
        for i := 1 to n do begin
            CAMMINO[i] := false;
            ORDINALE[i] := 0
        end;
        esisteciclo := false;
        for i := 1 to n do
            if ORDINALE[i] = 0 then DFS(G, i);
        ACICLICO := not esisteciclo;
    end;

procedure DFS(var G: grafo; u: nodo);
    var v: nodo;
    begin
        CAMMINO[u] := true;
        id := id + 1;
        ORDINALE[u] := id;
        for each v ∈ A(u) do begin
            if ORDINALE[v] = 0 then
                DFS(G, v)
            else if CAMMINO[v] and ORDINALE[v] < ORDINALE[u]
                then esisteciclo := true
        end;
        CAMMINO[u] := false;
    end;
```

Esercizio 9.4. Un grafo non orientato G è bipartito se l'insieme dei nodi può essere partizionato in due parti tali che nessun arco connette due nodi appartenenti alla stessa parte. G è 2-colorabile se ogni nodo può essere colorato di bianco o di rosso in modo che nodi connessi da archi siano colorati con colori distinti. Si dimostri che G è bipartito: (1) se e solo se è 2-colorabile; (2) se e solo se non contiene circuiti di lunghezza dispari.

Esercizio 9.5 (Grafo bipartito). Scrivere una procedura Pascal basata su una visita BFS per verificare se un grafo non orientato è bipartito oppure no.

Si cerca di colorare i nodi del grafo con due colori, utilizzando un vettore COLORE. I nodi non colorati adiacenti al nodo u vengono colorati col colore diverso da quello di u, mentre si verifica che i nodi già colorati adiacenti al nodo u abbiano colore diverso da quello di u. La seguente visita BFS modificata restituisce il risultato nella variabile booleana *bipartito*.

```
procedure BFS(G: grafo; r: nodo; var bipartito: boolean);
    type
        colori = (bianco, rosso, noncolorato);
    var u: nodo;
        v: nodo;
        COLORE: array[1..n] of colori;
        Q: coda; i: integer;
    begin
        CREACODA(Q);
        for i := 1 to n do
            COLORE[i] := noncolorato;
        COLORE[r] := bianco;
        INCODA(r, Q);
        bipartito := true;
        while not CODAVUOTA(Q) do begin
            u := LEGGICODA(Q);
            FUORICODA(Q);
            for each v ∈ A(u) do
                if COLORE[v] = noncolorato then begin
                    if COLORE[u] = bianco
                        then COLORE[v] := rosso
                        else COLORE[v] := bianco;
                    INCODA(v, Q);
                end else
                    if COLORE[u] = COLORE[v]
                        then bipartito := false;
        end;
    end;
```

Esercizio 9.6 (Ilona). La celebre spogliarellista Ilona deve organizzare il suo spettacolo. Ella deve togliere certi indumenti prima di altri (per esempio le scarpe prima delle calze), mentre altri indumenti possono essere tolti in un ordine qualsiasi (per esempio scarpe e camicetta). Come può trovare un ordine giusto per spogliarsi?

Esercizio 9.7 (Ordine topologico). Si definisce ordine topologico (o linearizzazione) di un grafo orientato aciclico (non necessariamente connesso) un ordinamento dei nodi tale che se (u, v) è un arco del grafo, allora u precede v nell'ordinamento. Scrivere una procedura Pascal che restituisce nel vettore TOPOLOGICO i nodi ordinati topologicamente.

L'ordine topologico si può ottenere eseguendo una visita DFS che calcola l'istante FINE$[u]$ in cui finisce la visita di ciascun nodo u e ordinando i nodi per istanti di fine decrescenti. Poiché il grafo è aciclico, non esistono archi all'indietro, ed è pertanto agevole dimostrare che se (u, v) è un arco, allora FINE$[v]$ < FINE$[u]$. Anziché calcolare esplicitamente FINE$[u]$, è sufficiente memorizzare il nodo u nel vettore TOPOLOGICO prima di completare la chiamata ricorsiva su u. Il nodo la cui visita DFS termina per i-esima è memorizzato in TOPOLOGICO$[n - i + 1]$, cioè il vettore è riempito a ritroso.

```
var TOPOLOGICO: array[1..n] of integer;
    VISITATO: array[1..n] of boolean;
```



```
procedure TOPOLOGICSORT(var G: grafo; n: integer);
    var i, u: integer;
    begin
        for u := 1 to n do VISITATO[u] := false;
        i := 1;
        for u := 1 to n do
            if not VISITATO[u] then DFS(G, u);
    end;

procedure DFS (var G: grafo; u: integer);
    var v: integer;
    begin
        VISITATO[u] := true;
        for each v ∈ A(u) do
            if not VISITATO[v] then DFS(G, v);
        TOPOLOGICO[n − i + 1] := u;
        i := i + 1;
    end;
```

Esercizio 9.8 (Scheduling). Siano dati n programmi $P_1, P_2, \ldots, P_n$ che richiedono rispettivamente tempo $t_1, t_2, \ldots, t_n$ per essere eseguiti su un calcolatore. Tra i programmi è definita una relazione $<$ di precedenza aciclica tale che $P_i < P_j$ se l'esecuzione di P_i deve essere completata prima dell'inizio dell'esecuzione di P_j. Si fornisca una procedura Pascal per determinare in tempo polinomiale l'istante in cui iniziare l'esecuzione di ciascun programma in modo da rispettare i vincoli di precedenza e terminare l'esecuzione di tutti i programmi il più presto possibile.

Si possono rappresentare i programmi come nodi di un grafo orientato aciclico dove gli archi rappresentano le relazioni di precedenza, cioè se $P_i < P_j$ allora (i, j) è un arco del grafo. Gli istanti di partenza si calcolano scandendo la sequenza dei nodi ordinati topologicamente (cfr. Esercizio 9.7, e assegnando all'istante di partenza di ogni nodo un valore che è determinato dalla somma dei tempi di esecuzione dei nodi precedenti. Oltre ai vettori TOPOLOGICO e VISITATO dell'Esercizio 9.7, si impiegano i vettori TEMPI, SOMME e PARTENZA che contengono, rispettivamente, i tempi di esecuzione di ogni nodo, la somma dei tempi di esecuzione dei nodi che precedono ciscun nodo secondo l'ordine topologico, e gli istanti di partenza di ciascun nodo. La complessità dell'algoritmo è $\Theta(n + m)$.

```
var TEMPI, PARTENZA, SOMME: array[1..n] of integer;

procedure SCHEDULING(var G: grafo);
    var j: integer;
    begin
        TOPOLOGICSORT(G, n);
        SOMME[1] := 0;
        for j := 2 to n do
            SOMME[j] := SOMME[j − 1] + TEMPI[TOPOLOGICO[j − 1]];
        for j := 1 to n do
            PARTENZA[TOPOLOGICO[j]] := SOMME[j];
    end;
```

Esercizio 9.9 (Componenti connesse). Si scriva una procedura Pascal per trovare le *componenti connesse* di un grafo non orientato usando un mfset con la compressione dei percorsi. Si analizzi la complessità della procedura.

Esercizio 9.10 (Larghezza albero radicato). Si modifichi una visita BFS per calcolare la larghezza di un albero radicato (cfr Esercizio 4.3).

Esercizio 9.11 (Distanze minime). Sia dato un grafo orientato fortemente connesso. Fissato un nodo r, la distanza minima di un qualsiasi nodo u da r è il numero minimo di archi tra tutti i cammini da r ad u. Si modifichi una visita BFS per calcolare la distanza minima di ciscun nodo.

Parte seconda

Progetto ed analisi di algoritmi

Capitolo 10

Strutture di dati e progetto di algoritmi

Dato un problema, non ci sono "ricette generali" per progettare algoritmi efficienti che lo risolvano. Al contrario, accade molto spesso di ripartire praticamente da zero ogniqualvolta si debba progettare un nuovo algoritmo!

Nel progettare algoritmi, comunque, possono essere evidenziate quattro "fasi" comuni attraverso le quali occorre usualmente passare:

1. *Classificazione del problema.* Innanzitutto, si cerca di stabilire se il problema da risolvere fa parte di una classe più ampia di problemi aventi caratteristiche comuni. Di solito, si usa distinguere tre classi di problemi:

   Problemi decisionali la cui risposta è sì o no (o vero o falso) a seconda che il dato d'ingresso soddisfi o no una certa proprietà (ad esempio: verificare se un grafo non orientato è connesso, o decidere se esiste una foglia di un albero radicato il cui livello sia maggiore di una certa costante k);

   Problemi di ricerca dove, tra tutte le possibili soluzioni, si vuole trovarne una, detta soluzione ammissibile, che soddisfi una certa condizione (ad esempio: ordinare una sequenza di numeri in senso crescente, o trovare una foglia di un albero radicato il cui livello sia maggiore di una certa costante k);

   Problemi di ottimizzazione nei quali alle soluzioni ammissibili è associata una misura (o costo, o obiettivo) e si vuole trovare una soluzione ottima, cioè una soluzione ammissibile la cui misura sia minima o massima (ad esempio: trovare una foglia di un albero radicato il cui livello sia massimo).

2. *Caratterizzazione della soluzione.* In secondo luogo, si prova a caratterizzare matematicamente la soluzione del problema. Nella maggior parte dei casi, la matematica usata è "elementare", ma ciò non significa che i risultati siano banali da dimostrare. Tale caratterizzazione di solito suggerisce un algoritmo di risoluzione semplice ma non molto efficiente (ad esempio: in una sequenza ordinata di numeri, il primo è minore degli altri $n-1$, il secondo è minore dei rimanenti $n-2$, e così via fino all'ultimo che è il massimo; questa caratterizzazione suggerisce un algoritmo di ordinamento basato sulla ricerca iterata del minimo, come illustrato nell'Esempio I.10);
3. *Tecnica di progetto.* Successivamente, si cerca di applicare certe tecniche di progetto di algoritmi per rendere gli algoritmi più veloci. Le cinque tecniche principali sono: divide et impera, backtrack, greedy, programmazione dinamica e ricerca locale. La prima di queste tecniche è già stata implicitamente introdotta, e consiste nel partizionare il problema in sottoproblemi più piccoli, risolubili indipendentemente, le cui soluzioni possono essere ricombinate per ottenere la

soluzione del problema di partenza (ad esempio: la procedura di "ricerca binaria" vista nel § 5);

4. *Strutture di dati*. Infine, si vede se c'è un modo astuto di strutturare i dati elaborati dall'algoritmo che permetta di eseguire velocemente le operazioni utilizzate nell'algoritmo (ad esempio: l'algoritmo di ordinamento basato sulla iterata ricerca del minimo è reso più veloce utilizzando uno heap).

Di solito, si passa per le fasi (2)-(4) nell'ordine citato, ma questo non sempre è vero. In alcuni casi, se ne possono considerare solo alcune e in un ordine qualsiasi, mentre in altri casi possono essere riconsiderate più di una volta per raffinare e rendere più efficienti diverse versioni successive dell'algoritmo.

Nel seguito di questo capitolo sono esemplificate le fasi di caratterizzazione matematica della soluzione e di utilizzo di strutture di dati nel progetto di algoritmi efficienti. Per far questo, si considera un noto problema di ottimizzazione su grafi orientati pesati, quello dei cammini minimi, e si mostra come la caratterizzazione della soluzione permetta di definire un algoritmo "prototipo", a partire dal quale si possono derivare diversi algoritmi, più o meno veloci, a seconda della struttura di dati impiegata. Le cinque tecniche di progetto, invece, saranno l'argomento dei prossimi capitoli.

10.1 Cammini minimi

Si consideri il seguente problema di ottimizzazione su grafi:

CAMMINI MINIMI: "Dato un grafo orientato $G = (N, A)$, con pesi (o lunghezze) interi relativi sugli archi in A, e dato un nodo $r \in N$, trovare un cammino da r ad u, per ogni nodo $u \in N$, tale che la somma delle lunghezze degli archi nel cammino sia la più piccola possibile".

Esempio 10.1 (Cammini minimi). Un proprietario di un TIR è libero di trasportare containers da una città ad un'altra. Un viaggio dalla città i alla città j porta un profitto di p_{ij} euro se c'è un container da trasportare da i a j, ma provoca una perdita di p_{ij} euro se il TIR viaggia scarico. Assumendo profitti $p_{ij} > 0$ e perdite $p_{ij} < 0$, il percorso più vantaggioso tra due città corrisponde ad un cammino minimo, in cui le lunghezze degli archi sono $c_{ij} = -p_{ij}$. □

Una soluzione ammissibile di questo problema è data da un insieme di $n - 1$ cammini, ognuno dei quali parte dal medesimo nodo r, ma giunge ad uno e uno solo dei rimanenti nodi. In generale, non è detto che un cammino da r ad un qualsiasi altro nodo u esista, poiché u potrebbe non essere raggiungibile da r. Perché esista almeno una soluzione ammissibile, pertanto, occorre che ogni nodo sia raggiungibile da r con un cammino.

Il verificarsi di questa condizione, da sola, non garantisce l'esistenza di una soluzione ottima. Infatti, poiché sono ammesse lunghezze negative, potrebbe esistere nel grafo un ciclo per il quale la somma delle lunghezze degli archi nel ciclo dia un valore negativo. In tal caso, la lunghezza minima di un qualsiasi cammino che includa un nodo di tale ciclo non sarebbe inferiormente limitata. Occorre quindi che non esistano cicli di lunghezza negativa.

Si osservi che due cammini distinti possono avere un tratto iniziale comune, da r fino ad un certo nodo s, e poi divergere verso le rispettive destinazioni u e v, ma viceversa non possono convergere in un nodo comune s dopo aver percorso due tratti iniziali distinti, perché in tal caso ci sarebbero due cammini distinti da r ad s, mentre

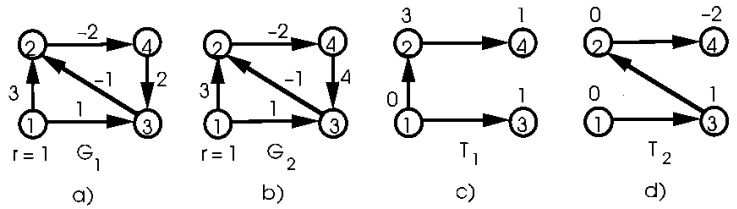


Figura 10.1: a) Un grafo G_1 con un ciclo negativo; b) un grafo G_2 privo di cicli negativi; c) una soluzione ammissibile per G_2; d) una soluzione ottima per G_2.

il problema ne richiede uno solo. Pertanto, una soluzione ammissibile altro non è che un albero di copertura, radicato in r, che include un cammino da r ad ogni altro nodo.

Esempio 10.2 (Albero di cammini). La Fig. 10.1 mostra un grafo G_1 che contiene un ciclo negativo ed un grafo G_2 privo di cicli negativi. Gli alberi T_1 e T_2 sono, rispettivamente, una soluzione ammissibile ed una soluzione ottima per G_2 (assumendo $r = 1$). □

Sia T una soluzione ammissibile. Ogni nodo i del grafo è caratterizzato da un intero d_i, che indica la distanza di i da r in T, e che è uguale alla somma delle lunghezze degli archi che si trovano nell'unico cammino tra r ed i in T. Quali caratteristiche devono avere le distanze affinché T sia una soluzione ottima? Consideriamo un generico arco $(i,j) \in A$ e sia c_{ij} la sua lunghezza. Ovviamente, se $(i,j) \in T$, allora $d_i + c_{ij} = d_j$. Se invece $(i,j) \notin T$, allora, affinché T sia ottimo, deve risultare $d_i + c_{ij} \geq d_j$, poiché altrimenti esisterebbe nel grafo G un cammino tra r e j più corto di quello in T, e precisamente il cammino tra r ed i in T seguito dell'arco (i,j). Viceversa, sia $d_i + c_{ij} = d_j$ per ogni $(i,j) \in T$, e $d_i + c_{ij} \geq d_j$, per ogni arco $(i,j) \notin T$. Se per assurdo il cammino da r a u in T non fosse ottimo, allora esisterebbe un altro cammino in G da r a u tale che la distanza d'_u di u da r sarebbe minore di d_u. Sia d'_j la distanza da r ad un generico nodo j che appare in tale cammino. Poiché $d'_r = d_r = 0$, ma $d'_u < d_u$, esisterebbe un arco (h,k) in questo cammino per cui $d'_h \geq d_h$ e $d'_k < d_k$. Per costruzione $d'_h + c_{hk} = d'_k$, mentre per ipotesi $d_h + c_{hk} \geq d_k$. Combinando queste due relazioni, si ottiene: $d'_k = d'_h + c_{hk} \geq d_h + c_{hk} \geq d_k$, e quindi $d'_k \geq d_k$, che contraddice l'ipotesi. Si è pertanto dimostrato il seguente risultato, dovuto a Bellman (1958):

Teorema di Bellman. *Una soluzione ammissibile T è ottima se e solo se valgono le seguenti condizioni: $d_i + c_{ij} = d_j$ per ogni arco $(i,j) \in T$, e $d_i + c_{ij} \geq d_j$ per ogni arco $(i,j) \in A$.* □

Questa caratterizzazione della soluzione permette di formulare un algoritmo che si basa sulla verifica, arco per arco, delle condizioni di Bellman, con conseguente sostituzione degli archi che violano tali condizioni:

```
procedure CAMMINIMINIMI(var G: grafo; r: nodo);
  begin
    {inizializza T ad un albero di copertura qualsiasi di radice r
      contenente un cammino da r ad ogni altro nodo u di G};
    while esiste un arco (i,j) tale che d_i + c_ij < d_j do begin
      d_j := d_i + c_ij;
      {sostituisci in T l'attuale arco entrante in j con l'arco (i,j)}
    end
  end;
```

Il passo cruciale dell'algoritmo consiste nel selezionare un arco (i,j) che violi le condizioni di Bellman. Ciò può essere effettuato mantenendo un insieme S di nodi, contenente inizialmente solo r. Selezionando un nodo i da S è possibile verificare le condizioni di Bellman su tutti gli archi (i,j) tali che j appartiene all'insieme di adiacenza $A(i)$. Se viene modificata la distanza d_j, il nodo j viene inserito in S. Rappresentando l'albero radicato T con un vettore dei padri, inizializzato ad un albero "fittizio" in cui tutti i nodi sono figli della radice r ed hanno una distanza molto grande (p.e. *maxint*), si ottiene la seguente procedura "prototipo":

```
procedure CAMMINIMINIMI(var G: grafo; r: nodo);
  var T, d: vettore; S: insieme; k: integer; i, j: nodo;
  begin
    CREAINSIEME(S);
    T[r] := 0; d[r] := 0;
    for k := 1 to n do
      if k ≠ r then begin
        T[k] := r; d[k] := maxint
      end;
    INSERISCI(r, S);
    while not INSIEMEVUOTO(S) do begin
      i := LEGGI(S); CANCELLA(i, S);
      for each j ∈ A(i) do
        if d[i] + c_ij < d[j] then begin
          T[j] := i;
          d[j] := d[i] + c_ij;
          if not APPARTIENE(j, S) then
            INSERISCI(j, S)
        end
    end
  end;
```

Nella procedura CAMMINIMINIMI sono utilizzati gli usuali operatori degli insiemi [cfr. § 5], più il nuovo operatore LEGGI che restituisce un elemento qualsiasi di S.

Tale procedura è corretta (in assenza di cicli negativi) ma non molto efficiente. La sua complessità dipende infatti dal numero massimo di inserzioni del medesimo nodo in S e dal numero massimo di miglioramenti delle distanze. In particolare, quest'ultima quantità può non essere polinomialmente correlata alla dimensione n del problema. Nel caso pessimo, le lunghezze c_{ij} possono avere valori che crescono come $\Omega(2^n)$, mentre le distanze possono essere migliorate solo di una unità alla volta, provocando così un tempo complessivo di esecuzione superpolinomiale in n.

Per ottenere una procedura efficiente, non basta che la struttura di dati S utilizzata sia un generico "insieme" con un'altrettanto generica operazione LEGGI, ma occorre specializzarla ulteriormente! Scelte ragionevoli per la struttura S possono sembrare la coda con priorità, la coda e la pila. Infatti, la prima è un caso particolare di insieme, e le altre due sono alla base di tutte le procedure di visita sistematica di un grafo. Inoltre, tutte e tre adottano criteri diversi ma sistematici di "lettura", al contrario della troppo vaga operazione LEGGI.

10.1.1 Algoritmo di Dijkstra

Se la struttura S è una coda con priorità, realizzata con una lista non ordinata, si ottiene un algoritmo noto fin dal 1959 e attribuito a Dijkstra. In tal caso, le operazioni LEGGI e CANCELLA si riducono rispettivamente a MIN e CANCELLAMIN [cfr. § 6]. Gli elementi della coda con priorità sono nodi del grafo, e le loro priorità sono le loro distanze da r. Ad ogni iterazione, è estratto da S il nodo i avente distanza $d[i]$ minima.

Benché la complessità dell'algoritmo resti superpolinomiale qualora il grafo contenga archi di lunghezza negativa, tale complessità si riduce drasticamente se tutte le lunghezze sono non negative. Sotto questa ipotesi, infatti, ogni nodo del grafo viene estratto da S una e una sola volta e, al momento dell'estrazione, la sua distanza da r è minima. Per convincersi di ciò, si proceda per induzione. Non essendoci lunghezze negative, l'asserto è chiaramente vero per il primo nodo estratto, cioè r, per il quale $d[r] = 0$. Quando viene estratto il k-esimo nodo, p.e. i, la sua distanza $d[i]$ dipende soltanto dai $k-1$ nodi già estratti, ognuno dei quali ha distanza minima per ipotesi d'induzione, ma non può dipendere dai nodi che si trovano ancora in S, che hanno distanza maggiore o uguale a $d[i]$. Non essendoci lunghezze negative, ed essendo $d[i]$ la distanza più piccola tra tutte quelle dei nodi in S, $d[i]$ è minima ed il nodo i non sarà più reinserito in S.

Esempio 10.3 (Algoritmo di Dijkstra). La Fig. 10.2 mostra un grafo G con lunghezze non negative degli archi e l'esecuzione dell'algoritmo di Dijkstra su tale grafo. L'ordine di estrazione da S è: 1, 2, 3, 5, 4, 6. □

Poiché ogni nodo è estratto da S una e una sola volta, il ciclo **while** è eseguito esattamente n volte. All'interno del ciclo, le operazioni più costose sono MIN e CANCELLAMIN che, utilizzando una lista non ordinata, richiedono tempo $O(n)$. L'operazione APPARTIENE può essere facilmente realizzata utilizzando un vettore booleano di n elementi [cfr. § 5] e richiede tempo $O(1)$, mentre la INSERISCI è anch'essa $O(1)$ utilizzando una lista non ordinata. Pertanto l'algoritmo di Dijkstra ha complessità $O(n^2)$.

10.1.2 Algoritmo di Johnson

Qualora la struttura S sia una coda con priorità realizzata con uno heap, si ottiene un algoritmo proposto da D.B. Johnson (1977). Poiché gli elementi della coda con priorità non coincidono con le priorità stesse (gli elementi sono nodi del grafo e le priorità distanze dei nodi da r) la struttura semplificata dello heap proposta nel § 7, dove gli elementi coincidevano con le priorità, va leggermente modificata. In particolare, poiché le priorità degli elementi sono decrementate con l'assegnamento "$d[j] := d[i] + c_{ij}$", lo heap va mantenuto ordinato secondo le priorità utilizzando una nuova operazione:

DECREMENTA: (elemento, priorità, prioricoda) → prioricoda
DECREMENTA(i, p, S) = S'
Pre: $i \in S$ e p è minore della priorità di i
Post: p è la nuova priorità di i

Per decrementare la priorità dell'elemento i, è necessario conoscerne la posizione nello heap. Ciò può essere ottenuto utilizzando un vettore di n elementi in cui l'elemento di indice i contiene un cursore all'elemento dello heap contenente i (se i non è nello heap, il cursore vale zero). La rappresentazione dello heap è la seguente (cfr. § 7):

```
type
    prioricoda = record
        elementi: array[1..maxlung] of integer;
        priorità: array[1..maxlung] of integer;
        posizione: array[1..maxlung] of integer;
        ultimo: 0..maxlung
    end;
```

Le operazioni APPARTIENE e DECREMENTA possono essere realizzate in Pascal come segue (la modifica delle operazioni MIN, CANCELLAMIN e INSERISCI è lasciata per esercizio):

```
function APPARTIENE(u: nodo; var S: prioricoda): boolean;
    begin
        APPARTIENE := (S.posizione[u] > 0)
    end;

procedure DECREMENTA(u: nodo; p: integer; var S: prioricoda);
    var temp1, temp2, i: integer;
    begin
        i := S.posizione[u];
        S.priorità[i] := p;
        while (i > 1) and (S.priorità[i] < S.priorità[i div 2]) do begin
            temp1 := S.elementi[i];
            temp2 := S.priorità[i];
            S.elementi[i] := S.elementi[i div 2];
            S.priorità[i] := S.priorità[i div 2];
            S.elementi[i div 2] := temp1;
            S.priorità[i div 2] := temp2;
            S.posizione[u] := i div 2;
            S.posizione[S.elementi[i]] := i;
            i := i div 2
        end
    end;
```

L'algoritmo di Johnson è analogo a quello di Dijkstra, e si ottiene dalla procedura prototipo CAMMINIMINIMI sostituendo, come prima, le operazioni degli insiemi con quelle delle code con priorità, e inserendo, dopo l'assegnamento "$d[j] := d[i] + c_{ij}$", la seguente istruzione:

"**if** APPARTIENE(j, S) **then** DECREMENTA(j, $d[i] + c_{ij}$, S);"

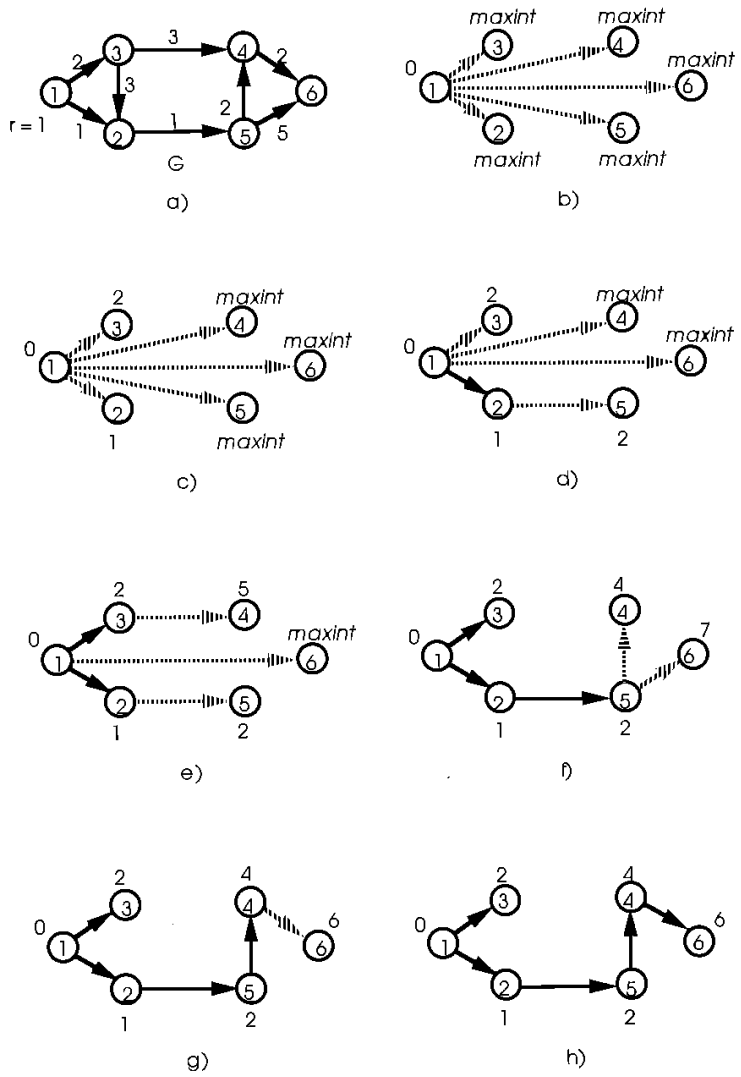


Figura 10.2: Algoritmo di Dijkstra. a) Un grafo G con lunghezze non negative; b) inizializzazione ad un albero fittizio per $r = 1$; c) - h) aggiornamento dell'albero e delle distanze, estraendo i nodi nell'ordine: 1, 2, 3, 5, 4, 6.

Qualora le lunghezze degli archi siano non negative, ogni nodo è estratto dallo heap S una e una sola volta e il ciclo **while** è eseguito esattamente n volte, come nell'algoritmo di Dijkstra. All'interno del ciclo, però, le operazioni CANCELLAMIN e DECREMENTA richiedono tempo $O(\log n)$. Poiché, nel caso pessimo, DECREMENTA è richiamata ad ogni iterazione su tutti i nodi di $A(i)$, la complessità totale è

$$O\left(n \log n + \left(\sum_{1 \le i \le n} |A(i)|\right) \log n\right).$$

Pertanto l'algoritmo di Johnson ha complessità $O(m \log n)$.

Si noti che per grafi sparsi, nei quali il numero m di archi è $O(n)$, la complessità diviene $O(n \log n)$ e l'algoritmo di Johnson risulta più veloce di quello di Dijkstra. Per grafi densi, dove m è $\Omega(n^2)$, la complessità risulta $O(n^2 \log n)$ e l'algoritmo di Johnson risulta più lento di quello di Dijkstra.

10.1.3 Algoritmo di Bellman-Ford-Moore

Se la struttura S è una coda si ottiene un algoritmo, noto sin dalla fine degli anni cinquanta e scoperto indipendentemente da molti ricercatori, che ha complessità polinomiale anche se ci sono archi di lunghezza negativa!

L'algoritmo di Bellman-Ford-Moore si ottiene dal prototipo sostituendo, rispettivamente, CREAINSIEME, INSERISCI, INSIEMEVUOTO, LEGGI e CANCELLA con CREACODA, INCODA, CODAVUOTA, LEGGICODA e FUORICODA, e realizzando APPARTIENE, al solito, usando un vettore booleano. La struttura dell'algoritmo è in pratica quella di una visita BFS in cui la marcatura di un nodo consiste nel diminuirne la distanza ed il medesimo nodo può essere visitato, cioè estratto dalla coda S, più di una volta, ma non più di $n-1$ volte!

La dimostrazione di questa proprietà è un po' macchinosa e si rifà alla nozione di "passata", che è definita induttivamente come segue:

a. per $k = 0$, la zeresima passata consiste nell'estrazione del nodo r dalla coda S;
b. per $k > 0$, la k-esima passata consiste nell'estrazione di tutti i nodi presenti in S al termine della passata $(k-1)$-esima.

Si indichino con $d_{v,k}$ il valore della distanza d_v all'inizio della passata k-esima, con p_v l'ultima passata in cui il nodo v si trova in S, con S_k l'insieme dei nodi in S nella passata k-esima, con $B(v)$ l'insieme $\{u\colon (u,v) \in A\}$, e con T^* l'albero ottimo dei cammini minimi, con distanze d_i^* per ogni nodo i.

La proprietà cruciale dell'algoritmo è che, per ogni arco (i,j) che sta nella soluzione ottima T^*, p_j è uguale a p_i oppure a $p_i + 1$. Innanzitutto, è facile verificare che, per ogni v e k, $d_{v,k} = \min\{d_{v,k-1} + c_{uv}\colon u \in B(v) \cap S_{k-1}\}$, poiché S è una coda. Da questa proprietà discende che se $j \in S_{p_j}$, allora deve esistere un nodo $v \in S_{p_j - 1}$ che ha provocato l'inserimento di j in S per l'ultima volta. Ma allora la distanza ottima d_j^* è stata determinata esplorando o l'arco (v,j), e quindi $p_j = p_v + 1$, oppure un arco (i,j) con $i \in S_{p_j}$, e allora $p_j = p_v$. In entrambi i casi, un eventuale reinserimento di i o v causerebbe un'ulteriore diminuzione di p_j e un reinserimento di j in S, il che è impossibile. Poiché il cammino semplice più lungo contiene al più $n-1$ archi, allora $p_j < n$ per ogni nodo i, ed il massimo numero di passate è n. Siccome ad ogni passata

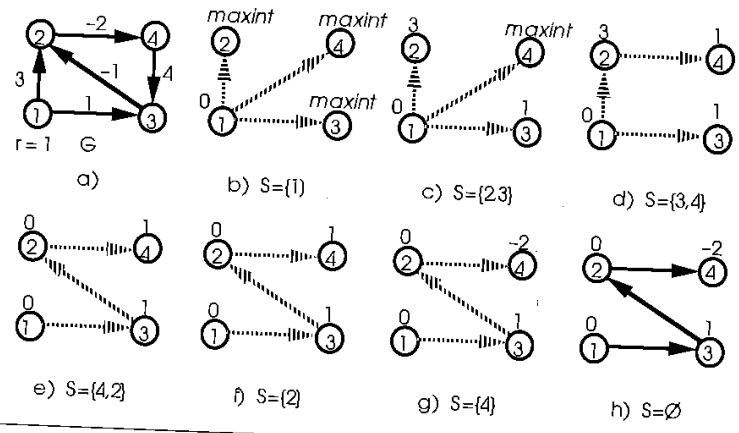


Figura 10.3: Algoritmo di Bellman-Ford-Moore. a) Un grafo G ($r = 1$); b)-g) estrazione dalla coda dei nodi nell'ordine: 1, 2, 3, 4, 2, 4; h) la soluzione ottima.

la coda S contiene al più tutti i nodi, ad ogni passata si esaminano al più tutti gli archi. Essendo il numero di passate limitato da n, l'algoritmo di Bellman-Ford-Moore ha quindi complessità $O(nm)$, qualsiasi sia il segno delle lunghezze degli archi.

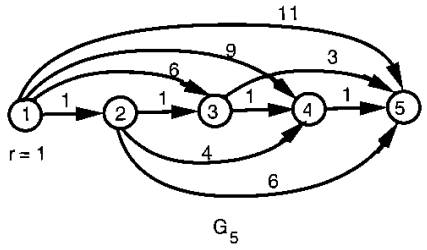
Esempio 10.4 (Algoritmo di Bellman-Ford-Moore). La Fig. 10.3 mostra l'esecuzione dell'algoritmo di Bellman-Ford-Moore su un grafo G per $r = 1$. L'ordine di estrazione dei nodi da S è: 1 (passata 0); 2, 3 (passata 1); 4, 2 (passata 2) e 4 (passata 3). □

10.1.4 Utilizzo di una pila

Benché venga spontaneo pensare che un algoritmo che impieghi una pila anziché una coda debba avere la stessa complessità dell'algoritmo di Bellman-Ford-Moore, l'utilizzo di una pila per la struttura S rende la procedura CAMMINIMINIMI di complessità superpolinomiale.

Si consideri il grafo $G_n = (N_n, A_n)$, con $n \ge 2$, così definito: $N_n = \{1, 2, \ldots, n\}$, $A_n = \{(i,j)\colon i < j\}$, con lunghezze $c_{ij} = 2^{n-i-1} - 2^{n-j} + j - i$. Si noti che G_n è un grafo *aciclico* e che le lunghezze sono non negative. Nonostante questa apparente semplicità della struttura di G_n, è possibile dimostrare che, utilizzando una pila, l'estrazione del nodo i da S provoca ogni volta l'inserimento di tutti i nodi k con $k > i$. Pertanto ogni nodo può essere estratto da S per $O(2^n)$ volte. Il fatto più curioso è che l'algoritmo di Bellman-Ford-Moore richiede soltanto un tempo $O(m)$ qualora sia eseguito su G_n!

Esempio 10.5 (Utilizzo di una pila). La Fig. 10.4 mostra il grafo G_n per $n = 5$, le configurazioni successive della pila S (sotto forma di vettore booleano), i nodi estratti da S, e i valori delle distanze ad ogni iterazione dell'algoritmo CAMMINIMINIMI eseguito su G_5 con $r = 1$. Si noti che le configurazioni della pila corrispondono a tutti i numeri in base 2 compresi tra $16 = 2^{n-1}$ e 0, e che il nodo $n = 5$ viene estratto da S per $8 = 2^{n-2}$ volte. □



pila S 1 2 3 4 5	nodo estratto	distanze d_1 d_2 d_3 d_4 d_5
1 0 0 0 0	1	0 ∞ ∞ ∞ ∞
0 1 1 1 1	5	0 1 6 9 11
0 1 1 1 0	4	0 1 6 9 11
0 1 1 0 1	5	0 1 6 9 10
0 1 1 0 0	3	0 1 6 9 10
0 1 0 1 1	5	0 1 6 7 9
0 1 0 1 0	4	0 1 6 7 9
0 1 0 0 1	5	0 1 6 7 8
0 1 0 0 0	2	0 1 6 7 8
0 0 1 1 1	5	0 1 2 5 7
0 0 1 1 0	4	0 1 2 5 7
0 0 1 0 1	5	0 1 2 5 6
0 0 1 0 0	3	0 1 2 5 6
0 0 0 1 1	5	0 1 2 3 5
0 0 0 1 0	4	0 1 2 3 5
0 0 0 0 1	5	0 1 2 3 4
0 0 0 0 0		0 1 2 3 4

Figura 10.4: Esecuzione della procedura CAMMINIMINIMI sul grafo G_5 usando una pila.

10.1.5 Algoritmo di Pape-D'Esopo

In pratica, l'algoritmo che risulta più veloce nei problemi reali impiega per S una struttura detta dequeue (per "double ended queue"), cioè una sequenza modificabile ad entrambi gli estremi che combina le proprietà di una pila e di una coda (cfr. Esercizio 3.4).

L'algoritmo che usa la dequeue, proposto da Pape (1974), permette di estrarre solo dalla testa di S, ma ammette l'inserzione sia in testa che in fondo. Per la precisione, quando un nodo è inserito in S per la prima volta allora viene aggiunto in fondo ad S, mentre quando è inserito le volte successive, allora viene aggiunto in testa ad S. In altri termini, quando un nodo j va inserito in S, se $d[j] = maxint$ allora S è trattata come una coda e j diviene l'ultimo elemento della sequenza, mentre se $d[j] < maxint$ allora S è trattata come una pila e j diventa il primo elemento della sequenza. La giustificazione di questa modalità di inserimento è la seguente. Ogni volta che la distanza $d[j]$ di un nodo j è decrementata, allora conviene tentare di diminuire anche le distanze dei nodi che seguono j nei cammini che appaiono nell'albero T che rappresenta la soluzione parziale corrente. Questo è lo scopo dell'inserimento in testa quando $d[j] < maxint$. D'altra parte, per evitare casi patologici con effetto "ping-pong" come si è visto nell'Esempio 10.5, non è conveniente inserire in testa quando la distanza corrisponde ad un cammino nell'albero "fittizio" T dovuto all'inizializzazione. Questo è lo scopo dell'inserimento in fondo quando $d[j] = maxint$.

Esempio 10.6 (Algoritmo di Pape-D'Esopo). Eseguendo l'algoritmo di Pape-D'Esopo sul grafo G della Fig. 10.3, è facile verificare che l'ordine di estrazione dei nodi da S è: 1, 2, 3, 2, 4. Si noti che rispetto all'algoritmo di Bellman-Ford-Moore il nodo 4 è estratto una sola volta e che il numero complessivo di estrazioni è 5 anziché 6. □

Benché nel caso pessimo l'algoritmo di Pape-D'Esopo richieda tempo superpolinomiale come quello che usa una pila, in pratica esso risulta il più veloce di tutti specie per grafi che rappresentano vere reti di comunicazione stradale. Questi grafi hanno la proprietà di essere sparsi, cioè con m che cresce come $O(n)$, e planari, cioè possono essere disegnati sul piano in modo che le linee corrispondenti a due archi distinti non si sovrappongano mai (dato un grafo G, verificarne la planarità richiede tempo $O(n+m)$ con un algoritmo complicato ideato da Hopcroft e Tarjan (1974); inoltre, se G è planare, è possibile fornirne un disegno planare in tempo polinomiale).

Esempio 10.7 (Planarità). I grafi illustrati nelle Figure 10.2a) e 10.3a) sono planari. Il grafo G_5 della Figura 10.4 invece non è planare, in quanto non è possibile disegnarlo nel piano eliminando la sovrapposizione tra gli archi (1,4) e (3,5) senza introdurre ulteriori sovrapposizioni tra altre coppie di archi. □

10.2 Esercizi

Esercizio 10.1 (Algoritmo di Dijkstra). Cosa succede se l'algoritmo di Dijkstra è eseguito su un grafo in cui le lunghezze degli archi possono essere negative?

Esercizio 10.2 (Algoritmo di Bellman-Ford-Moore). Si utilizzi l'algoritmo di Bellman-Ford-Moore per verificare in tempo $O(mn)$ se un grafo possiede cicli negativi.

Esercizio 10.3 (Algoritmo di Pape-D'Esopo). Si realizzi l'algoritmo di Pape-D'Esopo con una procedura Pascal e lo si esegua sul grafo G_5 della Fig. 10.4.

Esercizio 10.4 (Insieme indipendente di intervalli). Dato un insieme di n intervalli della retta reale, si vuole trovarne un insieme indipendente massimo, cioè un sottoinsieme X di intervalli, avente massima cardinalità, tale che ogni coppia di intervalli in X abbia intersezione vuota. Si fornisca un algoritmo polinomiale, formulando il problema come problema di cammini minimi su un opportuno grafo.

Si assuma senza perdere in generalità che gli n intervalli $[a_1, b_1], \ldots, [a_n, b_n]$ abbiano tutti i $2n$ estremi distinti e si aggiungano due intervalli fittizi $[a_0, b_0]$ e $[a_{n+1}, b_{n+1}]$, tali che $b_0 < a_i$, e $b_i < a_{n+1}$, per $1 \leq i \leq n$. Si costruisce un grafo orientato aciclico G con $n+2$ nodi corrispondenti agli $n+2$ intervalli tale che esiste un arco (i,j) se e solo se $b_i < a_j$. La distanza dell'arco (i,j) è -1 se $i \neq 0$, mentre è 0 se $i = 0$. È facile verificare che un cammino minimo tra il nodo 0 ed il nodo $n+2$ corrisponde ad un insieme indipendente massimo.

Esercizio 10.5 (Insieme dominante di intervalli). Dato un insieme di n intervalli della retta reale, si vuole trovarne un insieme dominante minimo, cioè un sottoinsieme X di intervalli, avente minima cardinalità, tale che ogni intervallo che non sta in X abbia intersezione non vuota con almeno un intervallo che sta in X. Si fornisca un algoritmo polinomiale, formulando il problema come problema di cammini minimi su un opportuno grafo.

Esercizio 10.6 (Insieme dominante totale pesato). Si riconsideri il problema dell'esercizio precedente qualora ad ogni intervallo sia associato un peso intero relativo e si richieda di trovare un insieme dominante totale di peso minimo, dove ogni intervallo (anche se sta in X) deve avere intersezione non vuota con almeno un intervallo che sta in X.

Esercizio 10.7 (Numero dei cammini minimi). Dati un grafo orientato con lunghezze positive sugli archi e due nodi r ed s, possono esserci molti cammini minimi distinti da r ad s. Si modifichi l'algoritmo di Dijkstra per determinare il numero dei cammini minimi distinti da r ad s.

Esercizio 10.8 (Cammino con arco più corto). Dati un grafo orientato con lunghezze positive sugli archi e due nodi r ed s, si vuole individuare il cammino da r ad s che contiene l'arco più corto, cioè quello con lunghezza minima tra tutti gli archi che possono apparire nel cammino. Si fornisca un algoritmo per determinare il cammino da r ad s contenente l'arco più corto e se ne analizzi la complessità.

Capitolo 11

Divide et impera

La tecnica più importante e più usata per progettare algoritmi efficienti si basa su una massima antica dei tiranni, enunciata anche dal Machiavelli a proposito della politica dell'antico senato romano:

"Per regnare, occorre tenere disuniti i nemici e trarne vantaggio".

Applicata alla risoluzione di un problema computazionale, questa tecnica consiste nel partizionare il problema in sottoproblemi più piccoli dello stesso tipo, risolverli, e successivamente ricombinare, con poco sforzo, le soluzioni parziali per ottenere la soluzione del problema originale. Lo "schema" di una procedura ricorsiva DIVIDEETIMPERA per risolvere un problema P di dimensione n è il seguente, dove k è una costante intera prefissata:

```
procedure DIVIDEETIMPERA(P, n);
  begin
    if n ≤ k then
      {risolvi P direttamente}
    else begin
      {dividi P in h sottoproblemi P_1, ... , P_h di dimensione n_1, ... , n_h};
      for i := 1 to h do DIVIDEETIMPERA(P_i, n_i);
      {combina i risultati di P_1, ... , P_h per ottenere quello di P}
    end
  end;
```

La tecnica "divide et impera" permette di provare agevolmente la correttezza dell'algoritmo usando il principio di induzione e di impostarne facilmente le relazioni ricorrenti della funzione $T(n)$ di complessità:

$$T(n) = \begin{cases} c, & \text{per } n \leq k, \\ D(n) + C(n) + \sum_{i=1}^{h} T(n_i), & \text{per } n > k, \end{cases}$$

dove c è una costante, mentre $D(n)$ e $C(n)$ sono il numero di operazioni per dividere il problema e per combinare i risultati, rispettivamente. Se i dati sono partizionati in maniera bilanciata, cioè tutti gli n_i sono all'incirca uguali, allora l'algoritmo può risultare molto efficiente.

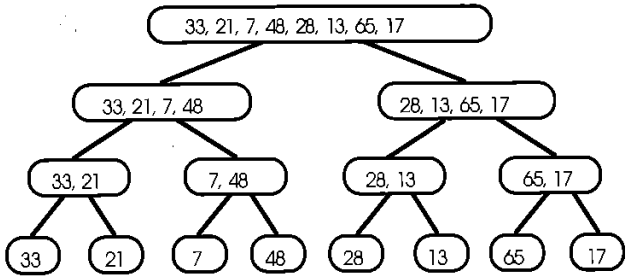


Figura 11.1: Partizionamento dei dati con la procedura MERGESORT.

Esempio 11.1 (Ricerca binaria). La procedura RICBIN vista nel § 6 è basata sulla tecnica "divide et impera", dove: $h = 2$, $n_2 \leq n_1 \leq n_2 + 1$, $C(n)$ e $D(n)$ sono costanti, e la chiamata ricorsiva avviene su uno solo dei due sottoproblemi. □

Il punto cruciale per ottenere algoritmi efficienti con la tecnica "divide et impera" è proprio il bilanciamento dei dati. Per chiarire questo punto, consideriamo il progetto di due diversi algoritmi di ordinamento, entrambi molto efficienti, ma totalmente differenti dall'HEAPSORT visto nel § 7, e di un algoritmo efficiente per calcolare la Trasformata Discreta di Fourier, che permette di effettuare molto velocemente il prodotto di due polinomi.

11.1 Mergesort

Un noto algoritmo per ordinare gli n elementi $A[1], \ldots, A[n]$ di un vettore consiste nel dividere il vettore in due sequenze di $n/2$ elementi ciascuno, ordinare ciascuna sequenza, e poi "fondere" le due metà ordinate in un'unica sequenza ordinata. La seguente procedura ricorsiva MERGESORT, da chiamare con l'istruzione MERGESORT(A, 1, n), realizza tale algoritmo:

```
procedure MERGESORT(var A: vettore; primo, ultimo: integer);
    var mezzo: integer;
    begin
        if primo < ultimo then begin
            mezzo := (primo + ultimo) div 2;
            MERGESORT(A, primo, mezzo);
            MERGESORT(A, mezzo + 1, ultimo);
            MERGE(A, primo, ultimo, mezzo)
        end
    end;
```

Esempio 11.2 (Partizione dei dati con Mergesort). Si supponga di eseguire la procedura MERGESORT per i dati d'ingresso: $A[1] = 33$, $A[2] = 21$, $A[3] = 7$, $A[4] = 48$, $A[5] = 28$, $A[6] = 13$, $A[7] = 65$, $A[8] = 17$. Il partizionamento dei dati corrispondente è visualizzato con l'albero mostrato nella Fig. 11.1. □

Per fondere le due metà ordinate, la MERGESORT chiama la procedura MERGE, che si avvale di un vettore di appoggio B, utilizzato come parametro globale, e che

è molto simile alla procedura utilizzata nel § 5 per l'operatore INTERSEZIONE nella realizzazione di insiemi mediante liste ordinate. La procedura MERGE usa tre indici i, j e k per scandire, rispettivamente, $A[primo..mezzo]$, $A[mezzo + 1..ultimo]$ e $B[primo..ultimo]$. Ad ogni passo, sono confrontati gli elementi $A[i]$ e $A[j]$, il minore dei due è copiato in $B[k]$, e sono incrementati k e l'indice dell'elemento minore. Il procedimento è iterato finché una delle due metà $A[primo..mezzo]$ e $A[mezzo + 1..ultimo]$ è esaurita. Se la prima metà è stata esaurita (cioè $i > mezzo$), gli eventuali elementi non scanditi $A[j..ultimo]$ della seconda metà si trovano già nelle posizioni che competono loro nell'ordinamento finale di $A[primo..ultimo]$. Se invece la prima metà non è stata esaurita (cioè $i \leq mezzo$), gli elementi non scanditi $A[i..mezzo]$ della prima metà vengono subito spostati nelle ultime posizioni $A[k..ultimo]$ che competono loro nell'ordinamento finale. Infine, la porzione $B[primo..k-1]$ è ricopiata in $A[primo..k-1]$, ottenendo così $A[primo..ultimo]$ ordinato.

```
procedure MERGE(var A: vettore; primo, ultimo, mezzo: integer);
    var i, j, k, h: integer;
    begin
        i := primo; j := mezzo + 1; k := primo;
        while (i ≤ mezzo) and (j ≤ ultimo) do begin
            if A[i] ≤ A[j]
                then begin B[k] := A[i]; i := i + 1 end
                else begin B[k] := A[j]; j := j + 1 end;
            k := k + 1;
        end;
        j := k;
        for h := i to mezzo do begin
            A[j] := A[h];
            j := j + 1
        end;
        for j := primo to k − 1 do A[j] := B[j];
    end;
```

Esempio 11.3 (Computazione Mergesort). L'ordine delle successive chiamate ricorsive della procedura MERGESORT può essere individuato visitando in ordine anticipato l'albero di Fig. 11.1. Ciascuna attivazione della procedura è eseguita sui dati di ingresso contenuti nel corrispondente nodo. Viceversa, l'ordine dei ritorni delle attivazioni ricorsive della procedura MERGESORT può essere individuato visitando in ordine differito l'albero di Fig. 11.2, che ha esattamente le stesse foglie dell'albero di Fig. 11.1. Ciascuna attivazione della procedura fornisce come dati di uscita quelli contenuti nel corrispondente nodo. L'asimmetria nei due casi è dovuta al fatto che la formulazione di un algoritmo ricorsivo va "dall'alto in basso" (la procedura è definita in termini della procedura stessa per dati più semplici), mentre la sua esecuzione procede "dal basso in alto" (si risolve prima il problema banale di ordinare un solo elemento e successivamente si fondono sequenze già ordinate ottenendo sequenze ordinate più lunghe). □

Poiché la procedura MERGE è $O(n)$, la complessità $T(n)$ di MERGESORT è $O(n \log n)$. Infatti, assumendo per semplicità $n = 2^m$ e indicando con c e d due

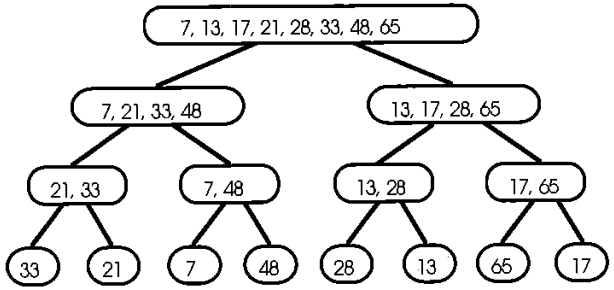


Figura 11.2: Fusione dei dati con la procedura MERGESORT.

costanti positive, si ottiene:

$$\begin{aligned} T(n) &= c, && \text{per } n = 1, \\ T(n) &\leq 2T(n/2) + dn, && \text{per } n \geq 2. \end{aligned}$$

Sostituendo successivamente, si ricava:

$$\begin{aligned} T(n) &\leq 2T(n/2) + dn \\ &\leq 2(2T(n/2^2) + dn/2) + dn \\ &\leq 2(2(2T(n/2^3) + dn/2^2) + dn/2) + dn \\ &\vdots \\ &\leq 2^m T(n/2^m) + 2^{m-1}dn/2^{m-1} + \ldots + 2^2 dn/2^2 + 2dn/2 + dn \\ &= 2^m c + dmn \end{aligned}$$

Essendo $m = \log n$, $T(n)$ risulta essere $O(n \log n)$. Poiché una limitazione inferiore del problema è $\Omega(n \log n)$, la procedura MERGESORT è ottima. Questa ottimalità, però, è solo in ordine di grandezza, perché ci sono altri algoritmi di ordinamento che in pratica risultano più veloci di MERGESORT.

Comunque, algoritmi di ordinamento basati sulla tecnica "mergesort" risultano vantaggiosi qualora gli elementi da ordinare risiedano in memoria secondaria (per esempio su disco). In tal caso, il tempo per trasferire avanti e indietro dati tra la memoria principale, che è veloce ma poco capiente, e la memoria secondaria, che viceversa è capiente ma lenta, supera di gran lunga il tempo richiesto per effettuare l'ordinamento nella memoria principale. Con le memorie secondarie, è quindi conveniente elaborare i dati in blocchi sequenziali, proprio come viene fatto dalla procedura MERGE. Pertanto, la maggior parte degli algoritmi di ordinamento su memoria secondaria si basano su una tecnica "mergesort polifase", distribuendo dapprima gli elementi da ordinare in molte sequenze ordinate (dette corse) memorizzate in memoria secondaria e poi eseguendo ripetutamente operazioni di fusioni tra corse finché non rimane un'unica corsa.

11.2 Quicksort

L'algoritmo praticamente più efficiente per ordinare gli elementi di un vettore è stato proposto da Hoare (1962) e, non a caso, è stato chiamato "quicksort" (quick significa "svelto"). Questo algoritmo è basato sulla tecnica "divide et impera", ma differisce dal "mergesort" nel modo in cui divide il problema e combina i risultati. La strategia consiste nel selezionare un elemento del vettore, detto perno, attorno al quale riarrangiare gli altri elementi. Tutti gli elementi più piccoli del perno vengono spostati in posizioni del vettore che precedono quella del perno, mentre gli elementi più grandi del perno sono spostati in posizioni che la seguono. Lo stesso algoritmo è poi riapplicato alle due porzioni di elementi minori e maggiori del perno.

Sia $A[primo..ultimo]$ la porzione di A che l'algoritmo, ad una generica chiamata, deve ordinare. La seguente procedura QUICKSORT richiama al suo interno una procedura PERNO che sceglie un elemento perno x e permuta gli elementi trovando un indice j tale che $A[i] \leq x$ per $primo \leq i < j$, $A[j] = x$, e $A[i] \geq x$ per $j+1 \leq i \leq ultimo$:

```
procedure QUICKSORT(var A: vettore; primo, ultimo: integer);
    var j: integer;
    begin
        if primo < ultimo then
            begin
                PERNO(A, primo, ultimo, j);
                QUICKSORT(A, primo, j − 1);
                QUICKSORT(A, j + 1, ultimo)
            end
    end;
```

La procedura PERNO può scegliere l'elemento x in vari modi. Supponiamo per semplicità che x sia uguale ad $A[primo]$. Gli altri elementi del vettore vengono via via suddivisi in tre sequenze di elementi: quelli minori di x, quelli maggiori o uguali ad x, e quelli non ancora esaminati. La procedura utilizza due cursori: il cursore i punta al primo elemento y non ancora esaminato e il cursore j dà la posizione finale in cui va spostato il perno. Ad ogni iterazione, se $y \geq x$ allora y non viene spostato, mentre se $y < x$ allora y è scambiato con il primo elemento tra quelli maggiori o uguali ad x. Al termine, x è scambiato con $A[j]$.

```
procedure PERNO(var A: vettore; primo, ultimo: integer; var j: integer);
    var x, temp: tipoelem;
    i: integer;
    begin
        x := A[primo]; j := primo;
        for i := primo 1 to ultimo do
            if A[i] < x then begin
                j := j + 1;
                temp := A[j];
                A[j] := A[i];
                A[i] := temp
            end;
        A[primo] := A[j];
        A[j] := x
    end;
```

1	2	3	4	5	6	7	8	9	10
9	12	8	18	6	13	11	3	5	10
j		i							
9	8	12	18	6	13	11	3	5	10
	j			i					
9	8	6	18	12	13	11	3	5	10
		j					i		
9	8	6	3	12	13	11	18	5	10
			j					i	
9	8	6	3	5	13	11	18	12	10
				j					
5	8	6	3	9	13	11	18	12	10

Figura 11.3: Esecuzione della procedura PERNO.

Esempio 11.4 (PERNO). La Fig. 11.3 mostra l'esecuzione di PERNO(A, 1, 10, j) per i dati d'ingresso: $A[1] = 9$, $A[2] = 12$, $A[3] = 8$, $A[4] = 18$, $A[5] = 6$, $A[6] = 13$, $A[7] = 11$, $A[8] = 3$, $A[9] = 5$ e $A[10] = 10$. Al termine dell'esecuzione, $j = 5$. □

Esempio 11.5 (QUICKSORT). La Fig. 11.4 mostra la partizione dei dati indotta dall'esecuzione della procedura QUICKSORT sul vettore A dell'Esempio 11.4. □

Per valutare la complessità della procedura QUICKSORT, impostiamone le relazioni ricorrenti. Siano $T(n)$ e $P(n)$ il numero di confronti tra elementi di A eseguiti, rispettivamente, dalle procedure QUICKSORT e PERNO. Poiché $P(n) = n - 1$, e le due porzioni di A sulle quali avviene la chiamata ricorsiva hanno $j-1$ e $n-j$ elementi, si ottiene:

$$T(n) = 0, \qquad \text{per } n = 0, 1,$$
$$T(n) = n - 1 + T(j-1) + T(n-j), \qquad \text{per } n \geq 2.$$

Nel caso pessimo, $A[primo]$ è sempre l'elemento più piccolo, e quindi la porzione di A con elementi minori di x è vuota, mentre quella con elementi maggiori o uguali ne contiene $n - 1$. Pertanto, $T(n) = n - 1 + T(n-1)$ e, sostituendo successivamente, si ha: $T(n) = \sum_{2 \leq k \leq n}(k-1)$, che è $O(n^2)$. Nel caso pessimo, la procedura QUICKSORT è lenta come l'algoritmo banale visto nell'Esempio I.10! Paradossalmente, ciò avviene proprio quando il vettore A è già ordinato e non sarebbe necessaria alcuna elaborazione, tranne una verifica del corretto posizionamento degli elementi che richiede solo $O(n)$ tempo. Questo cattivo comportamento è dovuto ad un forte sbilanciamento dei due insiemi sui quali avvengono le chiamate ricorsive!

Esempio 11.6 (Caso pessimo QUICKSORT). La Fig. 11.5 mostra la partizione dei dati fortemente sbilanciata indotta eseguendo QUICKSORT su un vettore già ordinato. □

Il termine "quicksort", comunque, non è immeritato per questo algoritmo, poiché la sua velocità è elevata non nel caso pessimo, ma in quello medio. Per analizzarne la complessità nel caso medio, si supponga per semplicità che tutti gli elementi di A

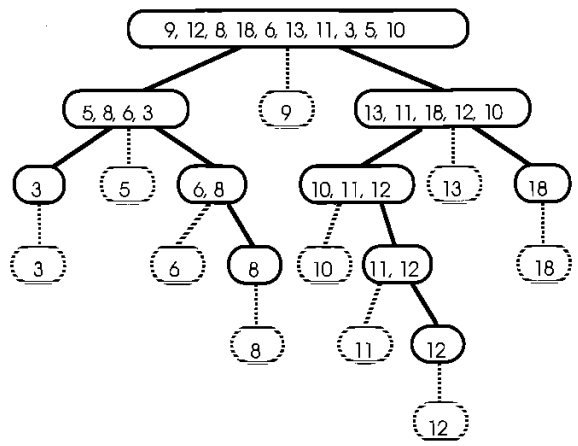


Figura 11.4: Ordinamento con la procedura QUICKSORT.

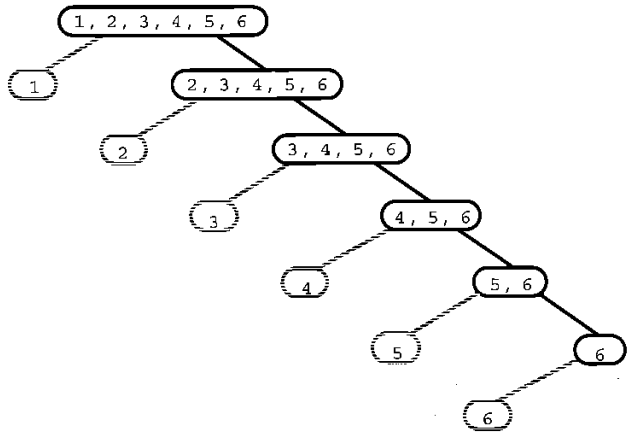


Figura 11.5: Caso pessimo della procedura QUICKSORT.

siano distinti, e che tutte le permutazioni degli elementi $A[1], \ldots, A[n]$ siano equiprobabili. Allora, il valore j prodotto dalla procedura PERNO può essere con la stessa probabilità $1/n$ in ciascuna posizione compresa tra 1 ed n, e quindi:

$$\begin{aligned} T(n) &= 0, && \text{per } n = 0, 1, \\ T(n) &= n - 1 + \frac{1}{n}\sum_{j=1}^{n}(T(j-1) + T(n-j)) \\ &= n - 1 + \frac{2}{n}\sum_{j=1}^{n-1} T(j), && \text{per } n \geq 2. \end{aligned}$$

Per risolvere la relazione, la si riscriva dapprima per $n-1$, e successivamente si calcoli $T(n) - (n-1)T(n-1)/n$, ottenendo:

$$\begin{aligned} T(n-1) &= n - 2 + \frac{2}{n-1}\sum_{j=1}^{n-2} T(j), \\ \frac{(n-1)T(n-1)}{n} &= \frac{(n-2)(n-1)}{n} + \frac{2}{n}\sum_{j=1}^{n-2} T(j), \\ T(n) - \frac{(n-1)T(n-1)}{n} &= n - 1 - \frac{(n-2)(n-1)}{n} + \frac{2T(n-1)}{n}. \end{aligned}$$

Semplificando, dall'ultima equazione si ricava:

$$\begin{aligned} T(n) &= \frac{2(n-1)}{n} + \frac{(n+1)T(n-1)}{n} \leq 2 + \frac{(n+1)T(n-1)}{n} \\ &\leq 2 + \left[2 + \frac{nT(n-2)}{n-1}\right]\frac{n+1}{n} \\ &= 2 + \frac{2(n+1)}{n} + \frac{(n+1)T(n-2)}{n-1} \\ &\leq 2 + \frac{2(n+1)}{n} + \left[2 + \frac{(n-1)T(n-3)}{n-2}\right]\frac{n+1}{n-1} \\ &= 2 + \frac{2(n+1)}{n} + \frac{2(n+1)}{n-1} + \frac{(n+1)T(n-3)}{n-2} \\ &\vdots \\ &\leq 2 + 2(n+1)\left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right] + \frac{(n+1)T(1)}{2} \\ &= \frac{2(n+1)}{n+1} + 2(n+1)\left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right] \\ &= 2(n+1)\sum_{k=3}^{n+1}\frac{1}{k}. \end{aligned}$$

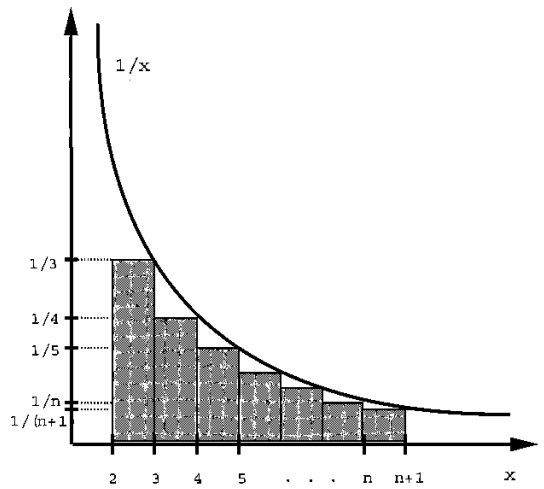


Figura 11.6: Maggiorazione di $1/3 + 1/4 + \cdots + 1/n + 1/(n+1)$ (area dei rettangoli grigi) con $\int_2^{n+1} \frac{1}{x}\mathrm{d}x$ (area sottesa dalla curva $1/x$ per $2 \leq x \leq n+1$).

Poiché, come illustrato nella Figura 11.6, si ha che

$$\sum_{k=3}^{n+1}\frac{1}{k} < \int_2^{n+1}\frac{1}{x}\,\mathrm{d}x = \ln(n+1) - \ln 2,$$

$T(n)$ risulta essere $O(n \log n)$ e la procedura QUICKSORT è quindi ottima nel caso medio!

Intuitivamente, il perno avrà in media un valore centrale tra $A[primo], A[primo + 1], \ldots, A[ultimo]$ e la procedura sarà richiamata ricorsivamente su due sottoinsiemi all'incirca della stessa dimensione. In pratica, se si adottano alcuni "accorgimenti", la procedura QUICKSORT non solo è veloce, ma è addirittura la più veloce tra tutti gli algoritmi di ordinamento noti finora per n abbastanza "grande" (per un fattore costante, ovviamente, non in ordine di grandezza, ché MERGESORT e HEAPSORT sono anch'esse asintoticamente ottime)! I principali accorgimenti, che possono essere combinati tutti insieme per esercizio ottenendo una nuova versione della procedura, sono i seguenti:

1. *Scelta del perno.* Un buon criterio di scelta del perno x consiste nel considerare i tre elementi $A[primo]$, $A[(primo + ultimo)$ **div** $2]$ e $A[ultimo]$, e sceglierne il mediano, cioè quello che non ha né il valore massimo né il valore minimo dei tre (dopo averlo scambiato con $A[primo]$, naturalmente);
2. *Procedura* PERNO. Un metodo più veloce, ma meno intuitivo, di realizzare la procedura PERNO consiste nell'utilizzare due cursori i e j per scorrere gli elementi di A, rispettivamente, da sinistra verso destra e da destra verso sinistra: i e j sono spostati finché non valgono entrambe $x > A[i]$ e $x < A[j]$, e a questo

punto $A[i]$ e $A[j]$ sono scambiati tra loro; inoltre, il tempo perso per la chiamata della procedura può essere risparmiato scrivendo le istruzioni della PERNO direttamente nella procedura QUICKSORT;

3. *Condizione di terminazione.* Poiché il tempo perso nella gestione della ricorsione è elevato quando il numero di elementi da ordinare è piccolo, è opportuno scegliere una dimensione minima di A, per esempio 10, al di sotto della quale l'ordinamento è effettuato con una semplice procedura non ricorsiva (per esempio con SHELLSORT, che sarà illustrata nel § 15);
4. *Gestione esplicita della ricorsione.* Per abbassare il tempo necessario alla gestione della ricorsione, è opportuno gestire la ricorsione esplicitamente con una pila, come illustrato nel § 3, avendo cura di salvare nella pila la minima informazione necessaria; per far questo, si possono salvare nella pila solo gli "estremi" di una porzione di A, ordinando direttamente l'altra porzione;
5. *Occupazione di memoria.* Poiché nel caso pessimo la partizione di A può essere sbilanciata, la porzione di A i cui estremi sono salvati nella pila può essere sempre la più piccola e la pila richiedere così $O(n)$ spazio; per ridurre la dimensione della pila, basta salvare nella pila gli estremi della porzione più grande, e ordinare direttamente quella più piccola; in questo modo, l'occupazione di memoria della pila diviene $O(\log n)$.

11.3 Trasformata veloce di Fourier

Dato un vettore di n componenti reali, la trasformata discreta di Fourier (DFT) lo trasforma in un vettore di n componenti complesse.

Sia $a = (a_0, a_1, \ldots, a_{n-1})$ il vettore da trasformare e si assuma n potenza di 2 (se non lo fosse, il vettore può essere espanso aggiungendo componenti nulle finché il numero di componenti n' non raggiunge la più piccola potenza di 2 maggiore di n, e le considerazioni sulla complessità non cambiano, perché $n' \leq 2n$). Sia ω la radice primitiva (complessa) n-esima dell'unità e sia $F = [f_{hk}]$ la matrice complessa $n\ times n$ tale che $f_{hk} = \omega^{hk}$, $0 \leq h, k \leq n-1$, dove ω^{hk} è la potenza hk-esima di ω. La trasformata discreta di Fourier del vettore a è il vettore complesso b ottenuto moltiplicando la matrice F per il vettore (colonna) a da trasformare, ovvero: $b = Fa$. Le singole componenti di b possono essere scritte per esteso come segue:

$$
\begin{aligned}
b_0 &= w^0 a_0 + w^0 a_1 + \ldots + w^0 a_{n-2} + w^0 a_{n-1} \\
b_1 &= w^0 a_0 + w a_1 + \ldots + w^{n-2} a_{n-2} + w^{n-1} a_{n-1} \\
&\vdots \\
b_h &= w^0 a_0 + w^h a_1 + \ldots + w^{h(n-2)} a_{n-2} + w^{h(n-1)} a_{n-1} \\
&\vdots \\
b_{n-1} &= w^0 a_0 + w^{n-1} a_1 + \ldots + w^{(n-1)(n-2)} a_{n-2} + w^{(n-1)(n-1)} a_{n-1}
\end{aligned}
$$

ovvero:

$$b_h = \sum_{k=0}^{n-1} a_k \left(w^h\right)^k \qquad \text{per } 0 \leq h \leq n-1.$$

In altri termini, indicato con

$$a(x) = \sum_{k=0}^{n-1} a_k x^k$$

il polinomio di grado $n-1$ avente come coefficienti le n componenti del vettore a, allora le componenti della trasformata di Fourier $b = Fa$ si ottengono valutando il polinomio $a(x)$ negli n punti w_0, w, w^2, ..., w^{n-1}, cioè in ciascuna delle radici n-esime dell'unità.

Un algoritmo banale per calcolare la trasformata di Fourier si ricava pertanto applicando per n volte la regola di Horner [cfr. § 6] per valutare in tempo $O(n)$ un polinomio in un punto, richiedendo quindi in totale $O(n^2)$ tempo. Sfruttando le proprietà delle radici complesse dell'unità, è possibile progettare un algoritmo basato sulla tecnica "divide et impera", detto FFT (per "Fast Fourier Transform", cioè "trasformata veloce di Fourier"), la cui complessità è $O(n \log n)$.

Prima di considerare l'algoritmo FFT, è opportuno richiamare brevemente le proprietà fondamentali delle radici complesse dell'unità[1].

11.3.1 Radici complesse dell'unità

Ricordiamo che $i = \sqrt{-1}$ è un numero immaginario e che un numero complesso z consiste di due parti, una reale ed una immaginaria, ed è di solito scritto come $z = r + is$, dove r ed s sono due numeri reali. Per sommare e moltiplicare due numeri complessi, si usano le usuali regole, sostituendo però ogni occorrenza di i^2 con -1.

I numeri complessi possono essere messi in corrispondenza biunivoca con i punti del piano, così che z corrisponde al punto di coordinate cartesiane (r, s) o, equivalentemente, di coordinate polari (m, θ), ove m è il modulo (distanza euclidea del punto dall'origine) e θ è l'angolo (formato tra l'asse reale positivo e il segmento che unisce il punto all'origine), espresso in radianti. È facile verificare che $m = \sqrt{r^2 + s^2}$ e $\theta = \tan^{-1}(r/s)$, mentre, viceversa, $r = m \cos\theta$ ed $s = m \operatorname{sen}\theta$. Dati due numeri complessi z_1 e z_2, allora il risultato della moltiplicazione $z_3 = z_1 z_2$ è tale che $m_3 = m_1 m_2$ e $\theta_3 = \theta_1 + \theta_2$.

Ogni polinomio di grado N con coefficienti complessi ha N radici complesse. In particolare, il polinomio $z^N - 1$ ha N radici complesse, date dalle N soluzioni dell'equazione $z^N = 1$. È immediato verificare che $(1, 0)$ è una radice di $z_N - 1$. Inoltre, se (m, θ) è una radice allora anche $(m^N, N\theta)$ lo è, per la proprietà della moltiplicazione usando le coordinate polari. Pertanto, le radici debbono essere del tipo $(1, N\theta)$, dove $N\theta = 0$. Poiché $2\pi j = 0$ per j intero, le N radici distinte sono $(1, 2\pi j/N)$, per $0 \leq j \leq N-1$. In particolare, la radice $\omega = (1, 2\pi/N) = \cos(2\pi/N) + i \operatorname{sen}(2\pi/N)$ è detta radice primitiva N-esima dell'unità, e tutte le N radici distinte dell'unità sono espresse come le j-esime potenze di ω, per $0 \leq j \leq N-1$. In altri termini, tali radici sono: $\omega^0, \omega^1, \omega^2, \ldots, \omega^{N-1}$, dove ovviamente $\omega^0 = 1$ e $\omega^1 = \omega$. Si noti che $\omega^N = 1$ e $\omega^{j+N} = \omega^j$ mentre, in generale, $\omega^j \omega^k = \omega^{j+k} = \omega^{(j+k) \bmod N}$. Analogamente, $\omega^{-1} = \omega^{N-1}$. Inoltre, se N è pari, allora $\omega^{N/2} = -1$, perché $(\omega^{N/2})^2 = \omega^N = 1$.

[1] Il lettore che abbia già dimestichezza con le proprietà delle radici complesse dell'unità può saltare direttamente al Paragrafo 11.3.2.

Esempio 11.7 (Radici quarte dell'unità). Se $N = 4$, allora gli angoli $2\pi j/N$ per $0 \leq j \leq 3$, sono uguali a: $0, \pi/2, \pi$, e $3\pi/2$. Le quattro radici quarte dell'unità sono: $\omega^0 = cos(0) + i\,\mathrm{sen}(0) = 1$, $\omega^1 = cos(\pi/2) + i\,\mathrm{sen}(\pi/2) = i$, $\omega^2 = cos(\pi) + i\,\mathrm{sen}(\pi) = -1$, e $\omega^3 = cos(3\pi/2) + i\,\mathrm{sen}(3\pi/2) = -i$. Si noti che $\omega^2 = \omega^{N/2} = -1$. □

Un lemma importante (detto di cancellazione), mette in relazione le radici dell'unità per valori diversi di N: per ogni $N \geq 0, j \geq 0$ e $d > 0$, la potenza dj-esima della radice primitiva dN-esima dell'unità uguaglia la potenza j-esima della radice primitiva N-esima dell'unità. Un secondo lemma (detto del dimezzamento), mostra invece che, se $N > 0$ è pari, allora i quadrati delle N radici complesse N-esime dell'unità sono le $N/2$ radici complesse $N/2$-esime dell'unità. Quest'ultima proprietà è cruciale per poter progettare un algoritmo "divide et impera", perché permette di dividere il problema in due sottoproblemi di dimensioni dimezzate.

11.3.2 Algoritmo FFT

Il polinomio $a(x)$ di n coefficienti (e grado $n-1$), del quale si vuole calcolare la trasformata di Fourier, può essere espresso per mezzo di due polinomi $a^P(x)$ e $a^D(x)$ di $n/2$ coefficienti ciascuno (e grado $n/2-1$) che includono, rispettivamente, gli elementi di posto pari e di posto dispari di a:

$$a^P(x) = \sum_{k=0}^{n/2-1} a_{2k}x^k, \qquad a^D(x) = \sum_{k=0}^{n/2-1} a_{2k+1}x^k,$$

così che

$$a(x) = a^P(x^2) + xa^D(x^2).$$

In questo modo, il problema di valutare $a(x)$ nelle n radici n-esime dell'unità $\omega^0, \omega, \omega^2, \ldots, \omega^{n-1}$ si riduce a valutare i due polinomi $a^P(x)$ e $a^D(x)$, di dimensioni dimezzate, nei punti $(\omega^0)^2, (\omega^1)^2, (\omega^2)^2, \ldots, (\omega^{n-1})^2$. Questi punti, per il lemma del dimezzamento, quando n è pari altro non sono che le $n/2$ radici $n/2$-esime dell'unità, con ciascuna radice che compare esattamente due volte. Infatti, essendo $\omega^{n/2} = -1$, risulta $\omega^{k+n/2} = -\omega^k$ e quindi $(\omega^{k+n/2})^2 = (\omega^k)^2$. Pertanto, il problema originale è stato partizionato in due sottoproblemi dello stesso tipo ciascuno di dimensione dimezzata. L'algoritmo FFT risultante, di tipo "divide et impera", può essere descritto con una procedura informale come segue:

procedure FFT(a: polinomio-reale; n: integer; **var** b: polinomio-complesso);
  var a^P, a^D: polinomio-reale; b^P, b^D: polinomio-complesso;
  begin
    if $n = 1$ **then** $b_0 := a_0$
    else begin
      dividi a in $a^P = (a_0, a_2, \ldots, a_{n-2})$ e $a^D = (a_1, a_3, \ldots, a_{n-1})$;
      FFT(a^P, n **div** 2, b^P); FFT(a^D, n **div** 2, b^D);
      ricava b ricombinando b^P e b^D
    end
  end;

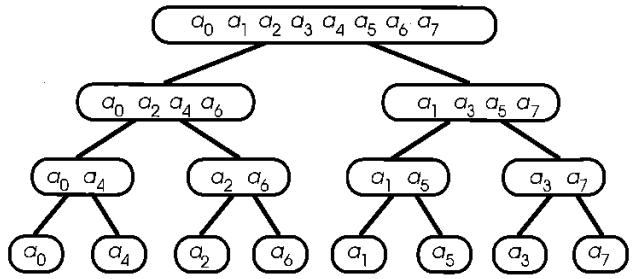


Figura 11.7: Partizionamento dei dati con l'algoritmo FFT.

Per ricostruire velocemente b da b^P e b^D nell'algoritmo FFT, si possono usare due variabili complesse ω e $\overline{\omega}$, inizializzate in modo che $\omega = \cos(2\pi/n) + i\,\mathrm{sen}(2\pi/n)$ e $\overline{\omega} = 1 = w^0$. Allora, la ricombinazione dei risultati dei due sottoproblemi può essere scritta informalmente col seguente ciclo:

for $h := 0$ **to** n **div** $2-1$ **do begin**
  $b_h := b_h^P + \overline{\omega} b_h^D$;
  $b_{h+n/2} := b_h^P - \overline{\omega} b_h^D$;
  $\overline{\omega} := \omega\overline{\omega}$
end

Infatti, poiché $\overline{\omega}$ assume all'interno del ciclo i valori $\omega^0, \omega, \omega^2, \ldots, \omega^{n/2-1}$, applicando le proprietà delle radici complesse dell'unità, si può dimostrare che, per $0 \leq h \leq n/2-1$, si ottengono:

$$\begin{aligned} b_h &= b_h^P + \omega^h b_h^D \\ &= a^P(\omega^{2h}) + \omega^h a^D(\omega^{2h}) \\ &= a(\omega^h), \end{aligned}$$

$$\begin{aligned} b_{h+n/2} &= b_h^P - \omega^h b_h^D \\ &= b_h^P + \omega^{h+n/2} b_h^D \\ &= a^P(\omega^{2h}) + \omega^{h+n/2} a^D(\omega^{2h}) \\ &= a^P(\omega^{2h+n}) + \omega^{h+n/2} a^D(\omega^{2h+n}) \\ &= a(\omega^{h+n/2}). \end{aligned}$$

Esempio 11.8 (Partizione dei dati con FFT). La Fig. 11.7 mostra il partizionamento del vettore a effettuato dall'algoritmo FFT quando $n = 8$. L'ordine delle chiamate corrisponde alla visita anticipata dell'albero di Fig. 11.7. □

Esempio 11.9 (Ricombinazione dei risultati con FFT). La Fig. 11.8 mostra la ricombinazione dei risultati per il calcolo del vettore trasformato b effettuato dall'algoritmo

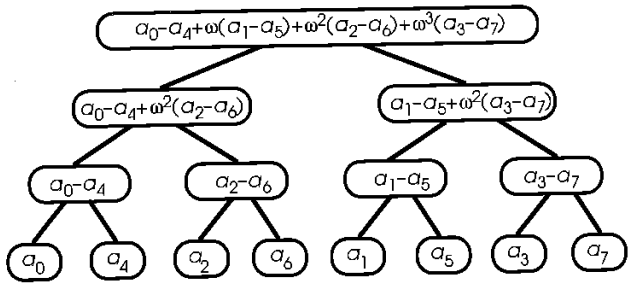


Figura 11.8: Ricombinazione dei risultati per il calcolo di b_1 con l'algoritmo FFT.

FFT per i dati dell'Esempio 11.8. Per semplicità, è mostrato il calcolo della sola componente $b_1 = a(\omega)$, che corrisponde alla visita differita dell'albero di Fig. 11.8, che ovviamente ha le stesse foglie dell'albero di Fig. 11.7. □

Nell'analizzare il numero di operazioni $T(n)$ dell'algoritmo FFT, è immediato verificare che sia la divisione in due sottoproblemi che la ricombinazione dei risultati dei due sottoproblemi richiede tempo $O(n)$. Ne segue che le risultanti relazioni di ricorrenza

$$\begin{aligned} T(n) &= c, && \text{per } n = 1, \\ T(n) &\leq 2T(n/2) + dn, && \text{per } n \geq 2, \end{aligned}$$

dove c e d sono costanti, sono identiche a quelle già viste per la procedura MERGESORT. Pertanto, la complessità dell'algoritmo FFT è $O(n \log n)$.

11.3.3 Moltiplicazione di polinomi

Tra le innumerevoli applicazioni della trasformata discreta di Fourier, ne consideriamo solo una, vedendo come sia possibile utilizzarne l'algoritmo FFT per calcolare velocemente il prodotto di due polinomi.

Siano dati i due polinomi $p(x) = p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \ldots + p_1x + p_0$ e $q(x) = q_{n-1}x^{n-1} + q_{n-2}x^{n-2} + \ldots + q_1x + q_0$, ciascuno avente n coefficienti (e quindi grado $n-1$). Tenendo conto che il polinomio prodotto $r(x)$ ha un numero doppio di coefficienti, un algoritmo banale di moltiplicazione può essere espresso con il seguente brano di istruzioni "quasi" Pascal:

for $h := 0$ **to** $2*(n-1)$ **do** $r_h := 0$;
for $h := 0$ **to** $n-1$ **do**
  for $k := 0$ **to** $n-1$ **do**
    $r_{h+k} := r_{h+k} + p_h * q_k$;

Questo banale algoritmo ha complessità $\Theta(n^2)$. Per abbassare la complessità, è opportuno passare attraverso una rappresentazione dei polinomi non per mezzo dei loro coefficienti, bensì dei valori che assumono quando sono valutati in n punti distinti qualsiasi. L'operazione di valutazione di un polinomio in un punto x_0, già vista precedentemente, consiste, dato il polinomio $a(x)$ rappresentato per mezzo dei suoi

coefficienti $(a_0, a_1, \ldots, a_{n-1})$, nel calcolare il valore di $a(x_0)$. In contrasto, l'operazione inversa alla valutazione è detta interpolazione e consiste, date n coppie di valori $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$, nel ricavare i coefficienti $(a_0, a_1, \ldots, a_{n-1})$ del polinomio $a(x)$ tale che $y_k = a(x_k)$, per $0 \leq k \leq n-1$. Un noto teorema assicura che il polinomio di interpolazione esiste ed è unico, qualsiasi siano gli n punti distinti $x_0, x_1, \ldots, x_{n-1}$.

Date le rappresentazioni per punti $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ e $(x_0, v_0), (x_1, v_1), \ldots, (x_{n-1}, v_{n-1})$ di due polinomi $p(x)$ e $q(x)$, allora $(x_0, y_0 + v_0), (x_1, y_1 + v_1), \ldots, (x_{n-1}, y_{n-1} + v_{n-1})$ è una rappresentazione per punti di $s(x) = p(x) + q(x)$. Allo stesso modo, ma usando $2n$ punti anziché n, poiché il polinomio prodotto ha grado $2n-1$, date le rappresentazioni per punti $(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})$ e $(x_0, v_0), (x_1, v_1), \ldots, (x_{2n-1}, v_{2n-1})$ di $p(x)$ e $q(x)$, allora $(x_0, y_0v_0), (x_1, y_1v_1), \ldots, (x_{2n-1}, y_{2n-1}v_{2n-1})$ è una rappresentazione per punti di $r(x) = p(x)q(x)$.

È immediato verificare che la rappresentazione per punti del polinomio prodotto $r(x)$ si può facilmente calcolare in $\Theta(n)$ tempo, date le rappresentazioni per punti di $p(x)$ e $q(x)$, contro il tempo $\Theta(n^2)$ richiesto per calcolare la rappresentazione con coefficienti di $r(x)$ date le rappresentazioni con coefficienti di $p(x)$ e $q(x)$! Questo suggerisce di passare dapprima dalle rappresentazioni con coefficienti di $p(x)$ e $q(x)$ alle loro rappresentazioni per punti, calcolare quindi velocemente la rappresentazione per punti di $r(x)$, e infine ricavare da questa la rappresentazione con coefficienti di $r(x)$. Occorrono pertanto $2n$ punti per valutare $p(x)$ e $q(x)$. In linea di principio, qualsiasi $2n$ punti distinti andrebbero bene, ma se questi punti sono le $2n$ radici complesse $2n$-esime dell'unità, allora la valutazione di $p(x)$ e $q(x)$ richiede soltanto $O(n \log n)$ tempo utilizzando l'algoritmo FFT!

Una volta trovata la rappresentazione per punti di $r(x)$, l'operazione di interpolazione che permette di ricavare la rappresentazione con coefficienti di $r(x)$ equivale ad una antitrasformata di Fourier, e può essere eseguita con un algoritmo quasi identico all'algoritmo FFT e sempre in tempo $O(n \log n)$. Infatti, mentre $b = Fa$ è la trasformata di Fourier di a, si ha che $a = F^{-1}b$ è l'antitrasformata di Fourier, dove $F^{-1} = [f_{ij}]$ è una matrice complessa $n \times n$ tale che $f_{hk} = \omega^{-kh}/n$, $0 \leq h, k \leq n-1$. La matrice F^{-1} è la matrice inversa di F, poiché $F^{-1}F = I$, dove I è la matrice identità $n \times n$ che ha elementi uguali ad 1 sulla diagonale principale, cioè quando $h = k$, ed elementi uguali a 0 altrove. Dati F^{-1} e b, le singole componenti di a sono:

$$a_k = \frac{1}{n} \sum_{h=0}^{n-1} b_h (\omega^{-k})^h \qquad \text{per } 0 \leq k \leq n-1.$$

Pertanto, l'algoritmo per calcolare velocemente l'antitrasformata di Fourier si ricava dall'algoritmo FFT semplicemente scambiando i ruoli di a e b, sostituendo ω con ω^{-1}, ed infine dividendo per n ciascuna componente del vettore risultante.

Usando trasformate ed antitrasformate di Fourier, è così possibile passare in tempo $O(n \log n)$ dalla rappresentazione con coefficienti di un polinomio a quella per punti, e viceversa.

Ricapitolando, per moltiplicare i due polinomi $p(x)$ e $q(x)$ date le loro rappresentazioni con coefficienti $(p_0, p_1, \ldots, p_{n-1})$ e $(q_0, q_1, \ldots, q_{n-1})$, dove n è una potenza di 2, innanzitutto si espandono tali rappresentazioni con zeri fino a raggiungere $2n$ componenti e poi si procede come segue per ricavare la rappresentazione con coefficienti $(r_0, r_1, \ldots, r_{2n-1})$ del polinomio prodotto:

1. si valutano $p(x)$ e $q(x)$ nelle $2n$ radici complesse $2n$-esime dell'unità, calcolando le trasformate di Fourier dei vettori espansi $(p_0, \ldots, p_{2n-1})$ e $(q_0, \ldots, q_{2n-1})$ in $O(n \log n)$ tempo;
2. si eseguono i prodotti $p(\omega^j)q(\omega^j)$, per $0 \leq j \leq 2n-1$, in $O(n)$ tempo;
3. si calcola l'antitrasformata di Fourier per interpolare e ricavare i coefficienti $(r_0, r_1, \ldots, r_{2n-1})$ del polinomio prodotto in $O(n \log n)$ tempo.

Esempio 11.10 (Moltiplicazione di polinomi). Siano $p(x) = 3x - 2$ e $q(x) = 4x + 1$. In tal caso, $n = 2$ e quindi $2n = 4$. Essendo le 4 radici quarte dell'unità $\omega^0 = 1$, $\omega^1 = i$, $\omega^2 = -1$, e $\omega^3 = -i$, la valutazione dei polinomi tramite la trasformata di Fourier è la seguente: $p(1) = 1$, $q(1) = 5$, $p(i) = -2 + 3i$, $q(i) = -1 + 4i$, $p(-1) = -5$, $q(-1) = -3$, $p(-i) = -2 - 3i$, $q(-i) = 1 - 4i$. Pertanto i prodotti dei valori dei polinomi nelle radici dell'unità sono: $p(1)q(1) = 5$, $p(i)q(i) = -14 - 5i$, $p(-1)q(-1) = 15$, e $p(-i)q(-i) = -14 + 5i$. Interpolando, si ricava tramite l'antitrasformata di Fourier che $r(x) = 12x^2 - 5x - 2$. □

11.4 Esercizi

Esercizio 11.1 (Mergesort su lista). Si riscriva la procedura Pascal MERGESORT nel caso in cui la sequenza da ordinare sia contenuta in una lista realizzata con puntatori, mantenendone la stessa complessità $O(n \log n)$.

Esercizio 11.2 (Quicksort su lista). Si riscriva la procedura Pascal QUICKSORT nel caso in cui la sequenza da ordinare sia contenuta in una lista realizzata con puntatori, mantenendone la stessa complessità media $O(n \log n)$.

Esercizio 11.3 (Counting sort). Dato un vettore A di n interi (non necessariamente distinti) appartenenti all'intervallo $1..k$, si scriva una procedura Pascal per ordinare il vettore A in tempo $O(\max\{n, k\})$.

La seguente procedura COUNTING-SORT inizialmente scandisce A e, avvalendosi di un vettore di appoggio *conta*[1..k], registra in *conta*[$A[j]$] il numero di occorrenze del valore $A[j]$, per $j = 1, 2, \ldots, n$. Successivamente, la procedura scandisce il vettore *conta*, per $i = 1, 2, \ldots, k$, e scrive il valore i nel vettore A per *conta*[i] volte.

```
procedure COUNTING-SORT(var A: vettore; n, k: integer);
    var i, j, h: integer; conta: array[1..maxlung] of integer;
    begin
        for i := 1 to k do conta[i] := 0;
        for j := 1 to n do conta[A[j]] : = conta[A[j]] + 1;
        for j := 1 to n do A[j] := 0;
        j := 1;
        for i := 1 to k do
            for h := 1 to conta[i] do begin
                A[j] := i;
                j := j + 1
            end
    end;
```


Poiché la procedura scandisce vettori di k ed n elementi, il tempo richiesto è $O(\max\{n, k\})$. Se k è $O(n)$, la procedura COUNTING-SORT ha complessità $O(n)$ e risulta più vantaggiosa di MERGESORT e QUICKSORT. È bene notare che la limitazione superiore $O(n)$ non è in contrasto con la limitazione inferiore $\Omega(n \log n)$ sul numero di confronti tra elementi da ordinare. Infatti, COUNTING-SORT non effettua confronti tra elementi di A. Ovviamente, se k è $\Omega(n \log n)$, allora COUNTING-SORT non è più conveniente degli algoritmi di ordinamento che hanno complessità $O(n \log n)$ ed anzi può essere molto svantaggiosa (per esempio, se k è $\Theta(2^n)$ allora la complessità è superpolinomiale).

Esercizio 11.4 (Inversioni). Data una sequenza di n interi memorizzata in un vettore A, si ha un'inversione tra gli elementi $A[i]$ e $A[j]$ se $i < j$ e $A[i] > A[j]$. Si modifichi la procedura MERGESORT per contare il numero totale di inversioni presenti in A.

Osservando la procedura MERGE, si nota che ogniqualvolta $A[i] > A[j]$ nel primo **if**, allora tutti gli elementi $A[i], A[i+1], \ldots, A[mezzo]$ provocano un'inversione con $A[j]$, per un totale di $mezzo - i + 1$ inversioni. Pertanto, si può scrivere una funzione CONTAMERGE, ottenuta dalla MERGE aggiungendo un nuovo parametro intero *conta*, inizializzato a 0 e restituito come risultato della funzione, e modificando il primo **if** come segue:

```
if A[i] ≤ A[j] then begin
    B[k] := A[i];
    i := i + 1
end else begin
    B[k] := A[j];
    j := j + 1;
    conta := conta + mezzo − i + 1
end;
```

La risultante funzione INVERSIONI, che conta il numero totale di inversioni di A, è ottenuta banalmente dalla procedura MERGESORT ed ha anch'essa complessità $O(n \log n)$:

```
function INVERSIONI(var A: vettore; primo, ultimo: integer): integer;
    var mezzo: integer;
    begin
        if primo ≥ ultimo then
            INVERSIONI := 0
        else begin
            mezzo := (primo + ultimo) div 2;
            INVERSIONI := INVERSIONI(A, primo, mezzo) +
                INVERSIONI(A, mezzo + 1, ultimo) +
                CONTAMERGE(A, primo, ultimo, mezzo);
        end
    end;
```

Esercizio 11.5 (Analisi Quicksort). Si analizzi la complessità di QUICKSORT nel caso in cui tutti gli elementi del vettore da ordinare abbiano lo stesso valore.

Esercizio 11.6 (Mergesort iterativo). Si fornisca una versione iterativa della procedura MERGESORT senza usare una pila e mantenendone la complessità di $O(n \log n)$.

Esercizio 11.7 (Ricerca in matrice ordinata). Sia data una matrice $n \times n$ di interi in cui gli elementi in ogni riga sono ordinati in modo crescente ed anche gli elementi in ogni colonna sono ordinati in modo crescente. Si progetti un algoritmo di tipo "divide et impera" per determinare se un dato intero k è contenuto nella matrice e se ne discuta la complessità.

Una limitazione inferiore di $\Omega(n)$ si può dimostrare applicando la tecnica dell'oracolo. Si supponga di avere una matrice $M[1..n, 1..n]$ con righe e colonne ordinate in modo crescente con elementi uguali a 0 oppure 1 sulla diagonale secondaria, con elementi negativi nel triangolo sopra la diagonale secondaria, ed elementi maggiori di 1 nel triangolo sotto la diagonale secondaria:

$$M[i,j] \begin{cases} < 0 & \text{per } i < n-j+1, \\ = 0 \text{ oppure } 1 & \text{per } i = n-j+1, \\ > 1 & \text{per } i > n-j+1. \end{cases}$$

Se l'elemento k da cercare è 1 e c'è un solo 1 nella matrice, tale elemento può occupare una posizione qualsiasi sulla diagonale secondaria, pertanto un qualsiasi algoritmo deve necessariamente scandirla tutta.

La seguente funzione APPARTIENE è una generalizzazione sulle matrici della ricerca binaria per i vettori ordinati. Si comincia la ricerca di k a partire dalla posizione in alto a destra $M[1,n]$. Si esclude dalla ricerca successiva la colonna n, qualora $M[1,n] > k$, oppure la riga 1, se $M[1,n] < k$. La ricerca viene poi riapplicata ricorsivamente alla sottomatrice rettangolare $n \times (n-1)$ (oppure $(n-1) \times n$) privata dell'ultima colonna (oppure della prima riga). La chiamata è APPARTIENE(M, 1, n, k, n).

```
function APPARTIENE(var M: matrice; i, j, k, n: integer): boolean;
  begin
    if M[i,j] = k then
      APPARTIENE := true
    else if M[i,j] < k then
      if i < n
        then APPARTIENE := APPARTIENE(M, i + 1, j, k, n)
        else APPARTIENE := false
      else if j > 1
        then APPARTIENE := APPARTIENE(M, i, j - 1, k, n)
        else APPARTIENE := false
  end;
```

Ad ogni chiamata si esclude o una riga o una colonna di M. Poiché M ha n righe ed n colonne, APPARTIENE ha complessità ottima $O(n)$.

Esercizio 11.8 (Minimo in una matrice). Data una matrice $n \times n$ di interi con n potenza di 2, si vuole determinarne l'elemento minimo. Si scriva una funzione Pascal utilizzando la tecnica "divide et impera".

La seguente funzione MINIMO divide la matrice in 4 sottomatrici di dimensioni dimezzate, richiama ricorsivamente se stessa sulle 4 sottomatrici, e poi trova il minimo tra i 4 minimi delle sottomatrici. Per semplicità, si assume di disporre di una funzione MIN4(a, b, c, d), che restituisce il minimo tra i quattro interi a, b, c, d.


```
function MINIMO(var M: matrice; imin, imax, jmin, jmax: integer): integer;
  var a, b, c, d, imed, jmed: integer;
  begin
    if imin > imax or jmin > jmax then
      MINIMO := maxint
    else begin
      if imin = imax and jmin = jmax then
        MINIMO := M[imin, jmin]
      else begin
        imed := (imin + imax) div 2;
        jmed := (jmin + jmax) div 2;
        a := MINIMO(imin, imed, jmin, jmed);
        b := MINIMO(imed + 1, imax, jmin, jmed);
        c := MINIMO(imin, imed, jmed + 1, jmax);
        d := MINIMO(imed + 1, imax, jmed + 1, jmax);
        MINIMO := MIN4(a, b, c, d)
      end
    end
  end;
```

Esercizio 11.9 (Torneo di tennis). Si progetti un algoritmo "divide et impera" per determinare il calendario di un torneo di tennis, cui partecipano $n = 2^k$ giocatori, in modo che ciascun giocatore incontri una ed una sola volta tutti gli altri, disputi un incontro al giorno, e la durata del torneo sia di $n-1$ giorni (Suggerimento: si costruisca una matrice $n \times n$ dove la prima colonna contiene i giocatori in ordine crescente e nelle rimanenti $n-1$ colonne si trovano tutti i giocatori che ognuno di essi dovrà incontrare).

Esercizio 11.10 (Prodotto di numeri complessi). Si individui un algoritmo per moltiplicare due numeri complessi usando solo tre operazioni aritmetiche di moltiplicazione anziché quattro.

Esercizio 11.11 (Esecuzione di FFT). Si calcoli la trasformata di Fourier del vettore $(0, 1, 2, 3)$ eseguendo l'algoritmo FFT.

Esercizio 11.12 (Realizzazione Pascal di FFT). Si riscriva l'algoritmo FFT con una procedura Pascal, utilizzando una realizzazione adeguata per i numeri complessi (cfr. Esercizi 1.1 e 1.2).

Esercizio 11.13 (Polinomi sparsi). Si supponga di dover moltiplicare tra loro due polinomi sparsi, cioè che hanno la maggior parte di coefficienti nulli. È conveniente utilizzare l'algoritmo FFT per effettuarne il prodotto? Come si possono rappresentare i polinomi sparsi?

Capitolo 12

Backtrack

Una tecnica non molto astuta, ma che talvolta può risultare utile, si basa sulla ovvia considerazione seguente:

"Prova a fare qualcosa e, se non funziona, disfala e riprova qualcosaltro".

Questa tecnica, detta backtrack, è alla base di molti algoritmi di visita di alberi e grafi, ed è usata per generare sistematicamente tutte le soluzioni ammissibili di un problema ("to backtrack" significa "tornare sui propri passi").

Esempio 12.1 (Visita di alberi e grafi). Le procedure PREVISITA, POSTVISITA e INVISITA [cfr. § 4] sono basate sul backtrack, dove l'elaborazione su un nodo è sospesa temporaneamente per permettere la visita di alcuni suoi sottoalberi. Anche la DFS [cfr. § 9], che è una diretta estensione della PREVISITA, è basata sul backtrack. □

Per meglio illustrare questa tecnica, consideriamo il progetto di due algoritmi per trovare il più piccolo poligono convesso che racchiude un dato insieme di punti del piano e per trovare un'occorrenza di una sequenza di simboli all'interno di un'altra sequenza di simboli.

12.1 Inviluppo convesso

Un poligono nel piano bidimensionale è convesso se ogni segmento di retta che congiunge due punti del poligono sta interamente nel poligono stesso, incluso il bordo. Dati n punti $p_1, \ldots, p_n$ nel piano, il loro inviluppo convesso è dato dal più piccolo poligono convesso che li contiene tutti. Intuitivamente, i punti possono essere pensati come chiodi conficcati a metà in un'asse di legno ed il bordo del loro inviluppo convesso come la sagoma formata da un elastico che li stringe tutti.

Esempio 12.2 (Inviluppo convesso). La Fig. 12.1 mostra un poligono non convesso, un poligono convesso, un insieme di punti, e il loro inviluppo convesso. □

Un algoritmo banale per trovare l'inviluppo convesso si basa sulla seguente caratterizzazione della soluzione. Un poligono può essere rappresentato per mezzo dei suoi spigoli, cioè dei segmenti di retta che giacciono sul suo bordo. Si consideri allora un generico segmento S_{ij} che unisce due punti distinti p_i e p_j, $1 \leq i < j \leq n$, e si consideri anche la retta che include S_{ij}. Tale retta divide il piano in due parti (dette semipiani chiusi, ognuno dei quali comprende anche la retta). Se tutti i rimanenti $n-2$ punti stanno dalla stessa parte rispetto alla retta che passa per S_{ij}, cioè stanno tutti nello stesso semipiano, inclusa la retta, allora S_{ij} è uno spigolo dell'inviluppo convesso

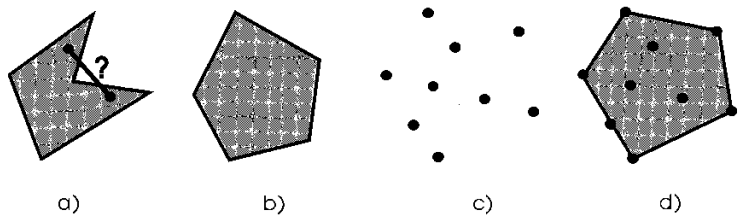


Figura 12.1: a) Un poligono non convesso; b) un poligono convesso; c) un insieme di punti; d) il loro inviluppo convesso.

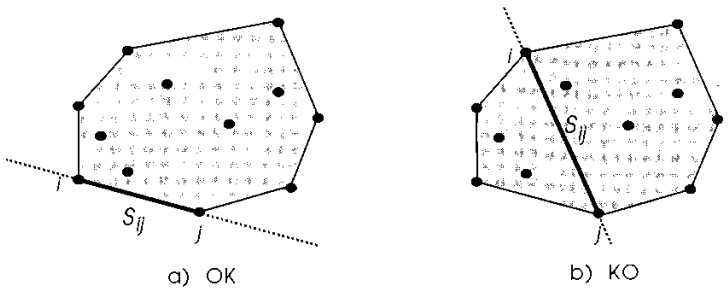


Figura 12.2: a) Il segmento S_{ij} è uno spigolo dell'inviluppo convesso; b) il segmento S_{ij} non è uno spigolo dell'inviluppo convesso.

e p_i e p_j ne sono due vertici (si veda la Fig. 12.2). Ripetendo per tutte le $n(n-1)/2$ coppie distinte di punti, si possono trovare tutti gli spigoli dell'inviluppo convesso. Individuati gli spigoli, il poligono può essere facilmente rappresentato tramite la sequenza dei suoi vertici, elencati secondo il verso antiorario (oppure orario).

Per realizzare tale algoritmo, vediamo innanzitutto come sia possibile verificare in tempo $O(1)$ se due punti stanno dalla stessa parte rispetto ad una retta oppure no. Ricordiamo che l'equazione di una retta è $y = ax + b$, dove $a = \Delta_y/\Delta_x$ ne è il coefficiente angolare. La retta divide il piano nei due "semipiani" dati dalle disequazioni

$$y \leq ax + b \qquad \text{e} \qquad y \geq ax + b.$$

Poiché la retta potrebbe essere verticale (cioè Δ_x potrebbe valere 0), si moltiplica per Δ_x ottenendo $y\Delta_x - x\Delta_y - b\Delta_x = 0$, dalla quale è possibile ricavare $b\Delta_x$ se sono date le coordinate di un punto $p = (p_x, p_y)$ che appartiene alla retta: $b\Delta_x = p_y\Delta_x - p_x\Delta_y$.

Consideriamo uno dei due semipiani, per esempio quello dato dalla disequazione

$$y\Delta_x - x\Delta_y - p_y\Delta_x + p_x\Delta_y \leq 0.$$

Se un punto $q = (q_x, q_y)$ appartiene a tale semipiano, allora deve valere

$$q_y\Delta_x - q_x\Delta_y - p_y\Delta_x + p_x\Delta_y \leq 0,$$

dalla quale segue che

$$(q_y - p_y)\Delta_x - (q_x - p_x)\Delta_y \leq 0.$$

Pertanto, per ogni punto q che appartiene a tale semipiano, la precedente quantità $(q_y - p_y)\Delta_x - (q_x - p_x)\Delta_y$ deve essere negativa o nulla, mentre per ogni punto q appartenente all'altro semipiano la stessa quantità deve essere positiva o nulla. Ne consegue che due punti q e q' stanno dalla stessa parte se le rispettive quantità hanno lo stesso segno, ovvero se il loro prodotto è positivo o nullo. Consideriamo adesso l'operazione STESSAPARTE, la cui specifica è:

STESSAPARTE: (retta, punto, punto) → booleano
STESSAPARTE: $(R, P, Q) = b$
Post: b = vero, se P e Q stanno dalla stessa parte rispetto ad R (incluso il caso che almeno uno dei due punti giaccia su R),
b = falso, se P e Q stanno da parti opposte rispetto ad R.

Per scrivere una funzione Pascal che realizzi STESSAPARTE, occorre innanzitutto scegliere un modo per rappresentare punti e rette. Per semplicità, assumiamo che i punti abbiano coordinate intere. Poiché un punto è individuato da una coppia di valori (le sue coordinate) mentre una retta è individuata da una coppia di punti distinti, una memorizzazione adeguata è la seguente:

```
type
    punto = record
        X, Y: integer
    end;
    retta = record
        P1, P2: punto
    end;
```

Dati una retta R, individuata dai due punti $R.P1$ ed $R.P2$, e due punti P e Q, la seguente funzione Pascal STESSAPARTE verifica in tempo $O(1)$ se le quantità discusse precedentemente hanno lo stesso segno, cioè se il loro prodotto è non negativo.

```
function STESSAPARTE(R: retta; P, Q: punto): boolean;
    var dx, dy, dx1, dx2, dy1, dy2: integer;
    begin
        dx := R.P2.X − R.P1.X;
        dy := R.P2.Y − R.P1.Y;
        dx1 := P.X − R.P1.X;
        dy1 := P.Y − R.P1.Y;
        dx2 := Q.X − R.P2.X;
        dy2 := Q.Y − R.P2.Y;
        STESSAPARTE := ((dx * dy1 − dy * dx1) * (dx * dy2 − dy * dx2) ≥ 0)
    end;
```


Utilizzando questa funzione, è un facile esercizio scrivere una procedura Pascal per trovare l'inviluppo convesso. Infatti, fissati due punti qualsiasi, cioè un segmento S_{ij}, per vedere se i rimanenti $n-2$ punti stanno dalla stessa parte rispetto alla retta R che passa per S_{ij} basta tener fisso uno di tali punti, p.e. P, ed eseguire STESSAPARTE(R, P, Q) per $n-3$ volte al variare di Q per ciascuno dei rimanenti $n-3$ punti. Pertanto verificare se un segmento S_{ij} è uno spigolo dell'inviluppo convesso richiede $O(n)$ tempo. Poiché ci sono in totale $n(n-1)/2$ segmenti S_{ij} distinti, si possono individuare tutti gli spigoli dell'inviluppo convesso con complessità $O(n^3)$.

12.1.1 Algoritmo di Graham

L'algoritmo appena illustrato non è molto efficiente, perché esamina gli n punti in modo troppo caotico. Vediamo come un esame sistematico dei punti, basato su un ordinamento ed un backtrack, permetta di progettare un algoritmo molto più efficiente, proposto da Graham (1972).

Innanzitutto, si osservi che in un insieme di punti, quello con ordinata minima è un vertice dell'inviluppo convesso. Inoltre, tracciando una retta orizzontale in tale punto e facendola ruotare intorno al punto stesso in senso antiorario si incontrano in sequenza tutti i rimanenti punti (uno per volta, se non ci sono due punti "allineati"). Come illustrato nella Fig. 12.3, i punti si possono rinumerare in base all'angolo α che la retta passante per ciascun punto p_i e quello di ordinata minima (il punto p_1) forma con l'asse orizzontale che passa per p_1. Se ci sono più punti con ordinata minima, allora quelli con ascissa minima e massima, sono vertici dell'inviluppo convesso, mentre gli altri giacciono sullo spigolo che unisce i due punti con ascissa minima e massima e possono essere eliminati, e può essere scelto come punto p_1 quello di ascissa massima. Allo stesso modo, se ci sono più punti allineati, cioè con lo stesso angolo α, allora possono essere eliminati tutti tranne il più distante da p_1, cioè quello di ordinata massima, perché tali punti non possono essere vertici dell'inviluppo convesso. Una volta che i punti sono ordinati per angolo α crescente, l'eliminazione dei punti allineati può essere eseguita in tempo $O(n)$ con una scansione della sequenza ordinata dei punti. Pertanto, senza perdere in generalità, si può assumere che non ci siano punti allineati, tranne eventualmente l'ultimo punto, che può avere la stessa ordinata del primo, ma ascissa minore.

Si osservi che, una volta ordinati, i primi due punti sono sempre vertici dell'inviluppo convesso (se $n \geq 2$) così come l'ultimo punto (se $n \geq 3$). Allora si possono considerare i punti $p_1, \ldots, p_n$ per indici (cioè angoli α) crescenti e costruire in modo incrementale l'inviluppo convesso "corrente" dei primi i punti $p_1, \ldots, p_i$, per $i = 1, 2, \ldots, n$. Ovviamente, se $n \geq 3$, allora i punti p_1, p_2 e p_3 sono i vertici dell'inviluppo convesso di p_1, p_2 e p_3. Si assuma di aver costruito l'inviluppo convesso dei primi $i-1$ punti, dato dalla sequenza dei suoi vertici in senso antiorario, e si provi ad aggiungere il nuovo punto p_i. Chiaramente, sia p_i che p_1 sono vertici dell'inviluppo convesso di $p_1, \ldots, p_i$. Se i punti p_i e p_1 non stanno dalla stessa parte rispetto alla retta R che passa per gli ultimi due vertici p_{j-1} e $p_j = p_{i-1}$ dell'inviluppo di $p_1, \ldots, p_{i-1}$, allora il punto p_{i-1} non può essere un vertice dell'inviluppo convesso di $p_1, \ldots, p_i$ e va eliminato. Il procedimento è ripetuto a ritroso aggiornando j e considerando la retta R che passa per gli ultimi due vertici p_{j-1} e p_j (escluso ovviamente quello appena eliminato) dell'inviluppo di $p_1, \ldots, p_{i-1}$, continuando ad eliminare l'ultimo vertice p_j se i punti p_i e p_1 non stanno dalla stessa parte rispetto alla retta R. Tale procedimento di eliminazione ha sicuramente termine, al più, quando si arriva a considerare la retta R che passa per p_1 e p_2, poiché tali punti sono vertici dell'invi-

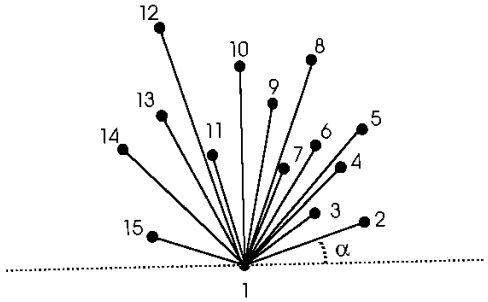


Figura 12.3: Ordinamento dei punti in base all'angolo α formato con l'asse orizzontale quando sono connessi al punto p_1.

luppo convesso di $p_1, \ldots, p_i$ per ogni i, $2 \leq i \leq n$. Queste osservazioni suggeriscono il seguente algoritmo informale:

function GRAHAM(insieme di punti $p_1, \ldots, p_n$): inviluppo convesso;
  var i, j: integer;
  begin
    trova il punto "più basso a destra" e ridenominalo p_1;
    riordina i rimanenti punti in base all'angolo formato rispetto all'asse orizzontale quando sono connessi con p_1 e ridenominali $p_2, \ldots, p_n$;
    elimina gli eventuali punti "allineati" tranne i più lontani da p_1, aggiornando se necessario n;
    inserisci p_1 nell'inviluppo "corrente" e, se $n \geq 2$, inserisci anche p_2;
    for i := 3 **to** n **do begin**
      scandisci "a ritroso" i punti nell'inviluppo "corrente" ed elimina p_j se STESSAPARTE(R, p_1, p_i) = **false**, dove R è la retta che passa per p_j e p_{j-1}, e termina tale scansione se p_j non deve essere eliminato;
      aggiungi p_i all'inviluppo "corrente"
    end;
    GRAHAM := inviluppo "corrente"
  end;

La dimostrazione di correttezza dell'algoritmo segue facilmente per induzione su i in accordo alle considerazioni fatte precedentemente. La fase di backtrack avviene all'interno del ciclo **for**, dove si eliminano i punti che sono vertici dell'inviluppo convesso di $p_1, \ldots, p_{i-1}$, ma che non lo sono più per $p_1, \ldots, p_i$.

Esempio 12.3 (Algoritmo di Graham). L'esecuzione dell'algoritmo di Graham è illustrata nella Fig. 12.4, dove sono evidenziati gli spigoli dell'inviluppo "corrente" (in nero), le rette R (tratteggiate), ed i punti che vengono eliminati ad ogni iterazione (con la croce), fino ad $i = 8$. Si noti che l'aggiunta del punto p_8 provoca l'eliminazione di due punti (p_7 e p_6), ma che ogni punto è inserito solo una volta ed è eliminato al più una volta. □

Il modo più intuitivo di memorizzare l'inviluppo "corrente" consiste nell'utilizzare una pila S nella quale sono accatastati i vertici dell'inviluppo. In questo modo, il nodo p_j eventualmente da eliminare è sempre quello in testa alla pila, mentre il nuovo

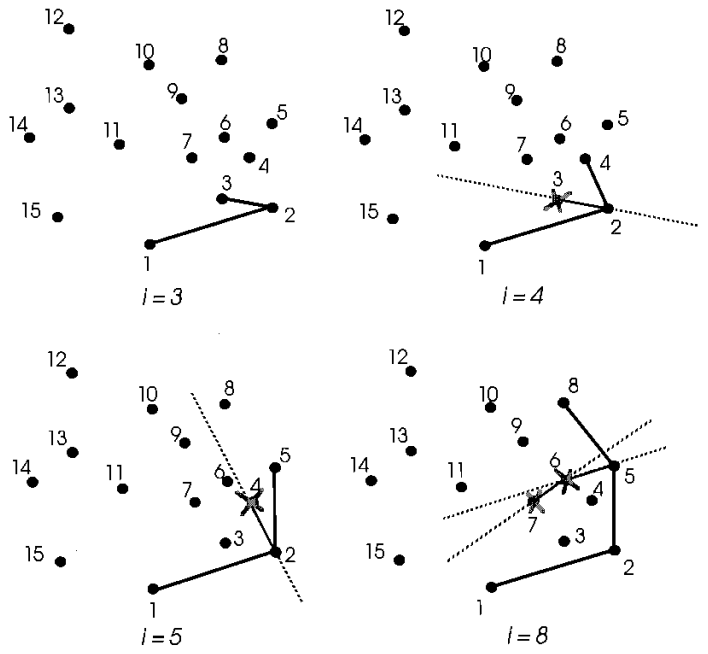


Figura 12.4: Esecuzione dell'algoritmo di Graham fino ad $i = 8$.

nodo p_i è anch'esso sempre inserito in testa alla pila. Per usare più semplicemente la pila, è conveniente introdurre la nuova operazione LEGGI2PILA(S), che restituisce il valore del secondo elemento a partire dalla testa della pila, ma senza modificare la pila. Il ciclo **for** dell'algoritmo di Graham diventa il seguente.

INPILA(p_1, S); INPILA(p_2, S);
for i := 3 **to** n **do begin**
  R := retta che passa per LEGGIPILA(S) e LEGGI2PILA(S);
  while not STESSAPARTE(R, p_1, p_i) **do begin**
    FUORIPILA(S);
    R := retta che passa per LEGGIPILA(S) e LEGGI2PILA(S);
  end;
  INPILA(p_i, S);
end;

Analizziamo la complessità del ciclo **for**. Tale ciclo è eseguito $n - 3$ volte ed al suo interno tutte le istruzioni, inclusa la condizione del ciclo **while**, richiedono tempo O(1). Pertanto la complessità del **for** dipende esclusivamente dal numero di ripetizioni del **while**, che a sua volta dipende dal numero di punti che sono cancellati dalla pila. Ogni punto è inserito nell'inviluppo "corrente" una ed una sola volta ed è

eliminato al più una volta. Quando un punto è eliminato, esso non viene più riconsiderato. Inoltre, ogni volta che è eseguito il corpo del **while** è eliminato un punto. Pertanto, la complessità del ciclo **for** è $\Theta(n)$. La complessità di tutto l'algoritmo è invece $O(n \log n)$, perché dominata dalla fase di ordinamento dei punti.

Realizzando l'insieme di punti con un vettore p di n elementi di tipo "punto", l'algoritmo di Graham può essere scritto in Pascal con la seguente funzione dove la pila è simulata direttamente con la porzione $p[1..j]$ del vettore, scambiando tra loro punti del vettore stesso.

```
function GRAHAM(var p: vettore; n: integer): integer;
    var i, j, min: integer; R: retta; T: punto;
    begin
        min := 1;
        for i := 2 to n do
            if p[i].Y < p[min].Y then min := i;
        T := p[1]; p[1] := p[min]; p[min] := T;
        ordina p[1], ... ,p[n] con HEAPSORT in base all'angolo α formato con
        l'asse orizzontale quando sono connessi con p[1], eliminando poi gli
        eventuali punti "allineati" ed aggiornando se necessario n;
        if n ≥ 2 then j := 2 else j := 1;
        for i := 3 to n do begin
            R.P1 := p[j]; R.P2 := p[j − 1];
            while not STESSAPARTE(R, p[1], p[i]) do begin
                j := j − 1;
                R.P1 := p[j];
                R.P2 := p[j − 1]
            end;
            j := j + 1;
            T := p[j];
            p[j] := p[i]; p[i] := T
        end;
        GRAHAM := j
    end;
```

La testa della pila è individuata dall'indice j. La cancellazione di $p[j]$ è effettuata semplicemente decrementando l'indice j all'interno del **while**. L'indice j è incrementato all'uscita per effettuare il successivo inserimento di $p[i]$, che avviene scambiando gli elementi $p[j]$ e $p[i]$, dato che $j \leq i$ e che $p[j]$ non è (più) un vertice dell'inviluppo "corrente" qualora $j < i$. La funzione restituisce come risultato l'indice j tale che i punti $p[1], \ldots, p[j]$ sono i vertici dell'inviluppo convesso, nell'ordine che si incontrano secondo il verso antiorario. L'ordinamento iniziale può essere fatto anch'esso direttamente sul vettore p con la procedura HEAPSORT, così che non sono necessarie strutture ausiliarie né per ordinare i dati di ingresso né per mantenere la soluzione, tranne ovviamente un campo aggiuntivo nel record di ciascun punto per contenere l'angolo α. I dettagli riguardanti le modifiche da apportare alla procedura HEAPSORT per ordinare il vettore di punti in base ad α e l'eliminazione degli eventuali punti "allineati" sono lasciati per esercizio.

12.2 String matching

Il problema dello "STRING MATCHING" consiste nel trovare un'occorrenza di una sequenza P di m caratteri, detta pattern, all'interno di un'altra sequenza T di n caratteri, detta testo. In altri termini, si chiede se esiste un indice k con $1 \leq k \leq n - m + 1$, tale che il j-esimo carattere di P sia uguale al $(k + j - 1)$-esimo carattere di T, per $j = 1, 2, \ldots, m$. Le due sequenze (o stringhe) P e T sono formate da caratteri tratti dallo stesso insieme A, di cardinalità finita, detto "alfabeto", e sono tali che m non supera n.

Esempio 12.4 (String matching). Siano $P =$ `10110110` e $T =$ `10110010101-101011011011`, dove $A = \{0, 1\}$, $n = 23$ ed $m = 8$. È facile vedere che c'è un'occorrenza del pattern P a partire dalla posizione $k = 14$ del testo T: `1011001-0101101011011011`. □

Un algoritmo ovvio per risolvere il problema consiste nel cercare di riconoscere il pattern a partire dalla prima posizione del testo, confrontando il primo carattere di P col primo di T, il secondo di P col secondo di T, ecc. Se il pattern non è interamente riconosciuto, si ripete il procedimento a partire però dalla seconda posizione nel testo, confrontando stavolta il primo carattere di P col secondo di T, il secondo di P col terzo di T, ecc. Se anche stavolta il pattern non è completamente riconosciuto, si riprova a partire dalla terza posizione del testo, poi dalla quarta, e così via, continuando fino al riconoscimento completo del pattern o all'esaurimento del testo. Realizzando le stringhe P e T con due vettori di caratteri, si ottiene la seguente funzione Pascal RICERCABRUTA, che utilizza tre indici: k (per avanzare nel testo), j (per scandire $P[1..m]$), ed i (per scandire la porzione $T[k..k + m - 1]$ quando cerca di riconoscere se $P[1..m] = T[k..k + m - 1]$). La funzione restituisce la posizione k di T a partire dalla quale si trova la prima occorrenza di P, se c'è una copia di P in T, oppure $i = n + 1$, se non c'è alcuna copia di P in T.

```
function RICERCABRUTA(var P, T: vettore; n, m: integer): integer;
    var i, j, k: integer;
    begin
        i := 1; j := 1; k := 1;
        while (i ≤ n) and (j ≤ m) do
            if T[i] = P[j]
                then begin i := i + 1; j := j + 1 end
                else begin k := k + 1; i := k; j := 1 end;
        if j > m then
            RICERCABRUTA := k
        else
            RICERCABRUTA := i;
    end;
```

L'algoritmo può essere interpretato graficamente come se la configurazione di caratteri individuata dal pattern fosse sovrapposta a quella del testo e si cerchi per quale traslazione del pattern sul testo tutti i caratteri del pattern siano uguali a tutti i caratteri del testo.

Esempio 12.5 (Sovrapposizione e traslazione). Eseguendo la procedura RICERCABRUTA sulle stringhe P e T dell'Esempio 12.4, si ottiene:

```
T = 10110010101101011011011   i = 1, j = 1, k = 1,
P = 10110110                  i = 6, j = 6,

T = 10110010101101011011011   i = 2, j = 1, k = 2,
P =  10110110                 i = 2, j = 1,

T = 10110010101101011011011   i = 3, j = 1, k = 3,
P =   10110110                i = 4, j = 2,

    ⋮

T = 10110010101101011011011   i = 14, j = 1, k = 14,
P =              10110110     i = 22, j = 8, EUREKA!
```

□

Se il pattern non è completamente riconosciuto a partire dalla posizione k del testo, la funzione RICERCABRUTA effettua un "backtrack" su entrambi gli indici i e j. Nel caso pessimo, l'indice k può assumere $n - m + 1$ valori e, per ogni valore di k, gli indici i e j possono assumerne m. Essendo in generale m più piccolo di n, la quantità $n - m + 1$ è $O(n)$, e quindi nel caso pessimo sono eseguiti $O(mn)$ confronti tra caratteri di P e T.

Esempio 12.6 (Caso pessimo RICERCABRUTA). Se $A = \{0, 1\}$, il caso pessimo si incontra quando sia P che T sono formate da tutti 0 seguiti da un unico 1 finale. □

Vediamo come un "backtrack" meno brutale permetta di progettare un algoritmo di complessità $O(m + n)$.

12.2.1 Algoritmo di Knuth, Morris e Pratt

L'idea di base dell'algoritmo proposto da Knuth, Morris e Pratt nel 1977 è la seguente: dopo aver riconosciuto $j - 1$ caratteri del pattern a partire da una certa posizione nel testo ed aver fallito al j-esimo, perché tornare indietro di $j - 2$ posizioni nel testo? In effetti, i $j - 1$ caratteri già riconosciuti fanno parte del pattern stesso e sono noti addirittura prima di iniziare la ricerca nel testo! Perché non trarre vantaggio da questa informazione nota in anticipo?

Esempio 12.7 (Backtrack inutile). Se $T =$ 10110010101101011011011 e $P =$ 10110110, i primi 5 caratteri di P sono uguali ai primi 5 caratteri di T, ma il sesto è diverso. Al momento del primo backtrack, $i = j = 6$. Ripartire con $i = 2$ e $j = 1$ corrisponde a traslare a destra di una posizione il pattern rispetto al testo. Ma ciò corrisponde anche a traslare a destra di una posizione rispetto a se stessa la sottosequenza 10110 di P che è stata riconosciuta nel testo! Nella sottosequenza 10110, i primi 4 caratteri non coincidono con gli utimi 4 ed è pertanto inutile ripartire con $i = 2$, perché sicuramente non potrà esserci una copia di P a partire da $T[2]$. Inoltre, neanche i primi 3 caratteri di 10110 coincidono con gli ultimi 3, ed è altrettanto inutile ripartire con $i = 3$, perché non potrà esserci una copia di P neanche a partire da $T[3]$. I primi 2 caratteri, invece, coincidono con gli ultimi 2 (10110 e 10110) e pertanto conviene ripartire direttamente con $i = 4$. Ma poiché i successivi due caratteri di P (cioè 10) coincidono sicuramente con quelli di T, tanto vale ripartire con $i = 6$ e $j = 3$ (anziché 1). Poiché 6 è proprio il vecchio valore di i al momento del backtrack, è inutile effettuare il backtrack sull'indice i, e basta effettuarlo soltanto sull'indice j, portandolo da 6 a 3. □

Si considerino due copie dei primi $j - 1$ caratteri del pattern, e si immagini di disporle orizzontalmente una sotto all'altra, in modo che il primo carattere della copia inferiore stia "sotto" il secondo carattere della copia superiore. Se tutti i caratteri sovrapposti nelle due copie non sono uguali, si traslino di una posizione a destra tutti i caratteri della copia inferiore. Si arresti tale procedimento di traslazione non appena tutti i caratteri sovrapposti nelle due copie siano identici, oppure quando non ci siano più caratteri sovrapposti. Il nuovo valore di "backtrack" da assegnare all'indice j, che indichiamo con "$back[j]$", è proprio uguale al numero di caratteri sovrapposti più uno (se non ci sono caratteri sovrapposti, $back[j]$ risulta ovviamente uguale ad 1, come nella RICERCABRUTA). Per $2 \leq j \leq m$, si ha:

$$back[j] = max\{h\colon h \leq j - 2 \text{ e } P[1..h - 1] = P[j - h + 1..j - 1]\}.$$

Per $j = 1$ è conveniente porre $back[1] = 0$, come sarà chiarito fra breve.

Esempio 12.8 (Valori di backtrack). I valori di $back[j]$ per il pattern $P =$ 10110110 sono i seguenti, accanto ai quali sono evidenziate per chiarezza anche le massime sovrapposizioni tra le due copie dei primi $j - 1$ caratteri di P che portano ai valori ottenuti:

```
back[1] = 0                 back[2] = 1   1
                                            1

back[3] = 1   10            back[4] = 2   101
                 10                         101

back[5] = 2   1011          back[6] = 3   10110
                 1011                       10110

back[7] = 4   101101        back[8] = 5   1011011
                 101101                     1011011
```

□

In altri termini, quando si verifica se c'è una copia di P a partire da $T[k]$ e risulta $P[1..j - 1] = T[k..i - 1]$ ma $P[j] \neq T[i]$, con $k \leq i \leq k + m - 1$, allora la prossima posizione di T per tentare di riconoscere P è $i - back[j] + 1$. Ma poiché, per definizione di $back[j]$, i primi $back[j] - 1$ caratteri di $P[1..j - 1]$ coincidono con gli ultimi, si ha che $P[1..back[j] - 1] = T[i - back[j] + 1..i - 1]$. Pertanto, i non viene modificato e si va a verificare se c'è una copia di $P[back[j]..m]$ a partire da $T[i]$.

L'algoritmo di Knuth, Morris e Pratt, sotto forma di funzione Pascal KMP, è ottenuto da RICERCABRUTA apportando alcune modifiche. Innanzitutto è introdotto in fase di inizializzazione il calcolo del vettore $back$, effettuato con una opportuna procedura CALCOLABACK. È poi eliminato l'indice k, divenuto un inutile doppione dell'indice i dato che non è effettuato più alcun "bactrack" su i. Il "backtrack" sull'indice j viene effettuato con l'assegnamento "$j := back[j]$", che sostituisce "$j := 1$", tranne in un caso particolare. Infatti, se j è uguale ad 1 al momento del "backtrack", allora significa che $P[1] \neq T[i]$ e che all'iterazione successiva occorre nuovamente considerare $P[1]$, ma confrontandolo con $T[i + 1]$. Siccome j è divenuto uguale a 0 con l'assegnamento "$j := back[j]$", essendo $back[1] = 0$ per definizione, allora j è subito reimpostato ad 1 ed i è incrementato. Questo è il motivo per il quale è stato definito $back[1] = 0$. La funzione KMP restituisce un intero con lo stesso significato di RICERCABRUTA; in particolare, se il pattern è stato interamente riconosciuto (cioè se $j > m$) allora la posizione di T a partire dalla quale si trova l'occorrenza di P è $i - m$.

```
function KMP(var P, T: vettore; n, m: integer): integer;
    var i, j: integer; back: vettore;
    begin
        CALCOLABACK(P, back, m);
        i := 1; j := 1;
        while (i ≤ n) and (j ≤ m) do
            if (j = 0) or (T[i] = P[j])[1]
                then begin
                    i := i + 1;
                    j := j + 1
                end else j := back[j];
        if j > m then KMP := i − m
            else KMP := i;
    end;
```

Per inizializzare il vettore *back* si riutilizza una lieve variante dello stesso algoritmo, in cui però si confronta il pattern P con se stesso. I valori di $back[j]$ sono computati per $j = 1, 2, \ldots, m$ in accordo alla definizione $max\{h\colon\ h \leq j-2$ e $P[1..h-1] = P[j-h+1..j-1]\}$. $back[1]$ e h sono inizializzati a 0 prima del ciclo, mentre $back[1] = 1$, poiché alla prima iterazione $h = 0$. Se $P[j] \neq P[h]$, allora si tenta con un valore più piccolo di h, il cui valore di "backtrack" $back[h]$ è stato quindi già computato. Quando invece j ed h sono entrambi incrementati, si ha che $P[1..h-1] = P[j-h-1..j-1]$ e quindi $back[j] = h$.

```
procedure CALCOLABACK(var P, back: vettore; m: integer);
    var j, h: integer;
    begin
        back[1] := 0;
        j := 1;
        h := 0;
        while j ≤ m do
            if (h = 0) or (P[j] = P[h])
                then begin
                    j := j + 1;
                    h := h + 1;
                    back[j] := h
                end else h := back[h];
    end;
```

Esempio 12.9 (Esecuzione CALCOLABACK). Eseguendo la procedura CALCOLABACK sulla stringa P dell'Esempio 12.4, si ha:

[1] Nel valutare un'espressione c_1 **or** c_1 **or** $\cdots$ **or** c_N, la maggior parte dei compilatori procede da sinistra verso destra, arrestando la valutazione al primo c_k = **true**. Pertanto, nella condizione "$(j = 0)$ **or** $(T[i] = P[j])$" la valutazione di $P[j]$ è effettuata solo se $j \neq 0$ e non dovrebbe essere riportato un errore di indirizzamento fuori dal vettore P. Comunque, è possibile definire P nell'intervallo $0..m$, utilizzando $P[0]$ come sentinella in modo da evitare eventuali errori di compilazione dovuti ad indirizzamenti fuori dal vettore.

$P =$	10110110	$j = 1, h = 0,$	
		$j = 2, h = 1,$	$back[2] = 1,$
$P =$	10110110	$j = 2, h = 1,$	
	1	$j = 2, h = back[1] = 0,$	
		$j = 3, h = 1,$	$back[3] = 1,$
$P =$	10110110	$j = 3, h = 1,$	
	1	$j = 4, h = 2,$	$back[4] = 2,$
	10	$j = 4, h = back[2] = 1,$	
$P =$	10110110	$j = 4, h = 1,$	
	1	$j = 5, h = 2,$	$back[5] = 2,$
	10	$j = 6, h = 3,$	$back[6] = 3,$
	101	$j = 7, h = 4,$	$back[7] = 4,$
	1011	$j = 8, h = 5,$	$back[8] = 5.$

□

La procedura CALCOLABACK può essere ulteriormente migliorata perché computa $back[j]$ senza tener conto dell'esito del confronto tra $P[j]$ e $P[h]$. Infatti, se la funzione KMP effettua un "backtrack" sull'indice j per un certo valore dell'indice i, allora deve essere $T[i] \neq P[j]$. Ma in tal caso, se $P[j] = P[h]$ allora anche $T[i] \neq P[h]$. Pertanto, poiché $back[j] = h$, la funzione KMP effettuerà un secondo backtrack su j senza modificare i. Ciò può essere evitato assegnando direttamente $back[h]$ a $back[j]$ qualora $P[j] = P[h]$. A tal fine, è sufficiente modificare la CALCOLABACK sostituendo l'assegnamento "$back[j] := h$" con l'istruzione:

if $P[j] = P[h]$ **then** $back[j] := back[h]$ **else** $back[j] := h$

Infatti, poiché il vettore *back* è riempito da sinistra verso destra ed $h < j$, il valore $back[h]$ è stato già computato e può essere utilizzato per calcolare $back[j]$. Questa semplice modifica garantisce che per ciascun valore dell'indice i ci sia al più un solo backtrack sull'indice j.

Esempio 12.10 (CALCOLABACK modificata). Si riconsideri il vettore *back* per $P =$ 10110110. Applicando la precedente modifica si ottiene:

$back[1] = 0,$	
$back[2] = 1,$	perché $0 = P[2] \neq P[1] = 1,$
$back[3] = back[1] = 0,$	perché $1 = P[3] = P[1] = 1,$
$back[4] = 2,$	perché $1 = P[4] \neq P[2] = 0,$
$back[5] = back[2] = 1,$	perché $0 = P[5] = P[2] = 0,$
$back[6] = back[3] = 0,$	perché $1 = P[6] = P[3] = 1,$
$back[7] = back[4] = 2,$	perché $1 = P[7] = P[4] = 1,$
$back[8] = back[5] = 1,$	perché $0 = P[8] = P[5] = 0.$

□

Durante l'esecuzione della funzione KMP, l'indice j può essere decrementato al più una volta per ciascun valore dell'indice i. Poiché i può assumere $O(n)$ valori diversi, il numero di confronti tra caratteri di P e di T eseguiti nel ciclo **while** è $O(n)$. D'altronde, per le stesse ragioni, la procedura CALCOLABACK modificata richiede $O(m)$ confronti tra caratteri. Pertanto, la complessità della funzione KMP è $O(m+n)$.

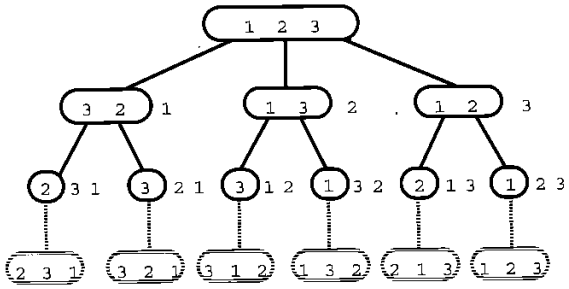


Figura 12.5: Generazione di tutte le permutazioni di A con $A[1] = 1, A[2] = 2$, e $A[3] = 3$.

Esempio 12.11 (Algoritmo di Knuth, Morris e Pratt). Eseguendo la funzione KMP sulle stringhe P e T dell'Esempio 12.4 ed utilizzando il vettore *back* dell'Esempio 12.10, si ottiene:

```
T =  101100101011010110110 11   i = 1, j = 1,
P =  10110110                   i = 6, j = 6, back[6] = 0,

T =  101100101011010110110 11   i = 7, j = 1,
P =        10110110             i = 10, j = 4, back[4] = 2,

T =  101100101011010110110 11   i = 10, j = 2,
P =          10110110           i = 15, j = 7, back[7] = 2,

T =  101100101011010110110 11   i = 15, j = 2,
P =                 10110110    i = 22, j = 8, EUREKA!
```

□

12.3 Esercizi

Esercizio 12.1 (Angoli α). Se dx e dy sono le differenze tra le ascisse e le ordinate dei punti p i e p_1, calcolare l'angolo $\alpha = \tan^{-1} dy/dx$ con la funzione di libreria Pascal per l'arcotangente richiede parecchio tempo. Dato che α serve solo nella fase di ordinamento dei punti dell'algoritmo di Graham, si può utilizzare una quantità diversa dall'angolo α, più veloce da calcolare e che produca lo stesso ordinamento. Si dimostri che $dy/(dy + dx)$ è adatta allo scopo e si scriva una opportuna funzione Pascal per calcolarla in tempo $O(1)$, tenendo conto dei casi in cui dx e dy possono essere positive, negative, o nulle.

Esercizio 12.2 (Calcolo vettore *back*). Si calcoli il vettore *back* utilizzato dalla funzione KMP per il pattern $P =$ `abracadabra`.

Esercizio 12.3 (Funzione KMP su lista). Si riscriva la funzione KMP assumendo che il pattern P ed il testo T siano memorizzati con liste realizzate con puntatori.

Esercizio 12.4 (Tutte le permutazioni). Si scriva una procedura Pascal basata sulla tecnica "backtrack" per stampare tutte le permutazioni degli n elementi contenuti in

un vettore $A[1..n]$. (Suggerimento: si applichi la definizione ricorsiva secondo la quale tutte le permutazioni di n elementi possono essere viste come tutte le permutazioni di $n - 1$ elementi seguite a turno dall'elemento mancante $A[i]$, per $i = 1, 2, \ldots, n$, come mostrato nella Fig. 12.5).

Capitolo 13

Greedy

Nella vita di tutti i giorni, spesso si adottano strategie che danno immediatamente buoni risultati, ma che a lungo andare si rivelano scadenti. È questa la strategia dell'ingordo (o "greedy"):

"Meglio un uovo oggi che una gallina domani".

Esempio 13.1 (Resto col numero minimo di monete). Si supponga di avere monete da 50, 10, 5 e 1 centesimo, e di dover fare un resto di 72 centesimi. Senza neanche rendercene conto, utilizziamo la filosofia dell'ingordo, scegliendo una moneta da 50 centesimi, poi due da 10, e infine due da 1. La soluzione prodotta è formata da cinque monete soltanto ed è ottima, nel senso che non ce n'è una con un numero minore di monete. L'ottimalità della soluzione, però, non dipende dalla furbizia dell'ingordo, ma da una proprietà delle monete! Si supponga di avere monete da 11, 5 e 1 centesimo, e di dover fare un resto di 15 centesimi. L'ingordo giudicherà la moneta da 11 più "appetibile" delle altre, e farà un resto con una moneta da 11 e quattro da 1, utilizzando cinque monete in tutto, mentre la soluzione ottima è data da tre monete da 5 centesimi! □

Il metodo greedy si può applicare a quei problemi di ottimizzazione in cui occorre selezionare un sottoinsieme S "ottimo" di oggetti, che verificano certe proprietà, da un insieme dato $\{a_1, \ldots, a_n\}$ di n oggetti. Uno "scheletro" di procedura che adotta tale metodo è il seguente:

procedure GREEDY(insieme $\{a_1, \ldots, a_n\}$ di oggetti);
  begin
    $S := \emptyset$;
    {ordina gli a_i per "appetibilità" decrescente};
    for $i := 1$ **to** n **do**
      if {a_i può essere aggiunto ad S}
        then $S := S \cup \{a_i\}$;
    {restituisci S come risultato}
  end;

Un algoritmo greedy ordina dapprima gli oggetti in base ad un criterio di "appetibilità". La soluzione del problema è poi costruita in modo incrementale considerando gli oggetti uno alla volta e aggiungendo ogni volta l'oggetto più appetibile, se possibile. In altri termini, l'algoritmo effettua una sequenza di scelte, preferendo ogni volta la scelta che fornisce immediatamente il miglior risultato. Fatta la scelta, è successivamente risolto un sottoproblema dello stesso tipo ma di dimensione più piccola.

Il problema più piccolo dipende dalla sequenza di scelte passate, ma non dalle scelte future. Da quanto detto, affinché un algoritmo greedy fornisca la soluzione ottima di un problema, occorre che siano verificate due proprietà, tra loro correlate:

1. *Scelta greedy*. Dopo aver fornito una caratterizzazione matematica della soluzione ottima, occorre dimostrare che tale soluzione può essere modificata in modo da utilizzare una prima scelta "greedy", che riduce il problema ad un sottoproblema più piccolo dello stesso tipo, e poi mostrare per induzione che il procedimento può essere esteso ad una sequenza di scelte "greedy";
2. *Sottostruttura ottima*. Per mostrare che una scelta "greedy" riduce il problema ad un sottoproblema più piccolo dello stesso tipo, occorre dimostrare che una soluzione ottima del problema contiene al suo interno le soluzioni ottime dei sottoproblemi.

Ovviamente, non sempre esiste un algoritmo greedy che trovi la soluzione ottima di un qualsiasi problema di ottimizzazione, come abbiamo già potuto osservare per il problema del resto col minimo numero di monete. Vediamo due esempi notevoli di problemi per i quali la tecnica greedy si comporta bene: trovare il minimo albero di copertura di un grafo pesato ed individuare l'ordine migliore (schedule) col quale eseguire un insieme di programmi su un processore in modo da minimizzare il numero di programmi che non rispettano certe scadenze temporali.

13.1 Minimo albero di copertura

Si consideri il seguente problema di ottimizzazione su grafi pesati:

MINIMO ALBERO DI COPERTURA: "Dato un grafo non orientato e connesso $G = (N, A)$, con pesi non negativi sugli archi, trovare un albero di copertura per G, cioè un albero avente tutti gli n nodi di N, ma solo $n - 1$ degli m archi in A, tale che la somma dei pesi degli archi nell'albero sia la più piccola possibile".

Esempio 13.2 (Minimo albero di copertura). La Figura 13.1 illustra un grafo non orientato pesato G e due alberi di copertura per G, dei quali il secondo è il minimo albero di copertura. □

Questo problema può essere risolto con molti algoritmi, dei quali i più noti sono quelli di Kruskal (1956) e Prim (1957). Di questi, il primo si basa sul principio greedy, in cui gli archi sono ordinati per pesi crescenti e poi considerati uno alla volta, ed un generico arco è aggiunto all'albero T se non forma circuiti con gli archi precedentemente inseriti in T.

procedure KRUSKAL(grafo non orientato $G = (N, A)$);
  begin
    $T := \emptyset$;
    {ordina gli m archi di G per "peso" crescente};
    for $a := 1$ **to** m **do**
      if {l'arco $a = [i, j]$ non forma un circuito con gli altri archi di T}
        then $T := T \cup \{a\}$;
  end;

L'algoritmo di Kruskal costruisce T per unione di componenti connesse. La sua correttezza può essere dimostrata come segue. Sia F_k la minima foresta di copertura

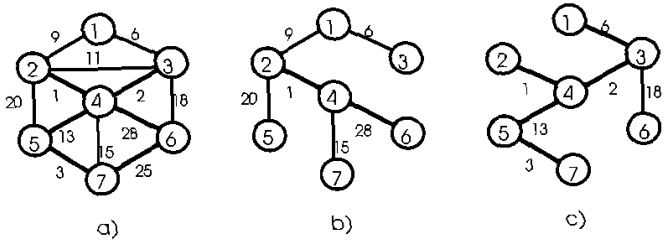


Figura 13.1: Minimo albero di copertura. a) Un grafo pesato; b) una soluzione ammissibile; c) una soluzione ottima.

per il grafo G, formata da k componenti connesse, cioè da k alberi. È facile provare per induzione che l'algoritmo computa successivamente $F_n, F_{n-1}, \ldots, F_1$, dove l'insieme A_n degli archi di F_n è vuoto, e l'insieme A_{k-1} degli archi di F_{k-1} è uguale ad $A_k \cup \{[i,j]: i \text{ e } j \text{ appartengono ad alberi distinti di } F_k\}$.

Esempio 13.3 (Algoritmo di Kruskal). La Fig. 13.2 mostra l'esecuzione dell'algoritmo di Kruskal sul grafo di Fig. 13.1. L'ordinamento degli archi è: [2,4], [3,4], [5,7], [1,3], [1,2], [2,3], [4,5], [4,7], [3,6], [2,5], [6,7], [4,6]. □

Il maggior sforzo computazionale dell'algoritmo di Kruskal è dato dall'ordinamento iniziale e dalla verifica che l'aggiunta di un nuovo arco non provochi la formazione di un circuito. L'ordinamento iniziale degli archi costa $O(m \log n)$ tempo, dato che $\log m \leq 2 \log n$. Inoltre, poiché la costruzione di T avviene per unione di insiemi disgiunti, cioè di due componenti connesse che si "fondono" in una sola con l'aggiunta del nuovo arco, è possibile utilizzare una struttura di dati di tipo "mfset" [cfr. § 5]. In questo modo, la verifica che un arco appartenga a due componenti distinte (cioè non provochi un circuito in T) ed anche il suo eventuale inserimento in T richiedono solo $O(\log n)$ tempo usando le operazioni TROVA e FONDI dell'"mfset". Poiché il ciclo **for** è ripetuto m volte, la complessità dell'algoritmo risulta essere $O(m \log n)$.

Per descrivere l'algoritmo di Kruskal con una procedura Pascal, occorre scegliere opportune realizzazioni per il grafo G in ingresso e per l'albero di copertura T fornito come risultato. Rappresentiamo il grafo semplicemente con un vettore $A[1..m]$ degli archi, dove ciascun arco $A[h]$ è un record contenente tre campi: i due nodi i e j che sono estremi dell'arco ed il peso dell'arco. Data l'usuale rappresentazione di un grafo con vettori di adiacenza, è facile vedere che il vettore degli archi può essere costruito in $O(n+m)$ tempo. L'albero T è rappresentato altrettanto semplicemente con un insieme, che può essere realizzato con una lista non ordinata, in modo che l'inserimento di un arco in T richieda tempo $O(1)$. Infine, è utilizzato un "mfset" S per mantenere le componenti connesse di T, tale che S è inizializzato con n componenti distinte, ciascuna formata da un singolo nodo del grafo.

```
type arco = record
    i, j: integer;
    peso: integer
end;

grafo = array[1..maxlung] of arco;
```

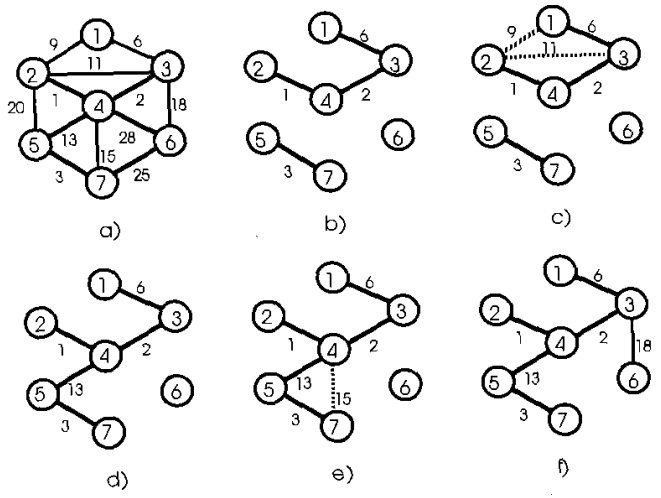



Figura 13.2: Algoritmo di Kruskal. a) Il grafo di ingresso. b) la foresta di copertura dopo l'inclusione di [2,4], [3,4], [5,7] e [1,3]; c) [1,2] e [2,3] non sono inclusi; d) inclusione di [4,5]; e) [4,7] non è incluso; f) inclusione di [3,6] e soluzione ottima.

La procedura Pascal KRUSKAL è la seguente, dove le modifiche da apportare alla procedura di ordinamento (p.e. HEAPSORT) usata per ordinare gli archi per pesi crescenti sono lasciate per esercizio. Come discusso precedentemente, la complessità di KRUSKAL è $O(m \log n)$.

```
procedure KRUSKAL(var A: grafo; n, m: integer; var T: insieme);
    var h: integer; S: mfset;
    begin
        CREAINSIEME(T);
        CREAMFSET(n, S);
        {ordina A[1], ..., A[m] in modo che A[1].peso ≤ ... ≤ A[m].peso};
        for h := 1 to m do
            if TROVA(A[h].i, S) ≠ TROVA(A[h].j, S) then begin
                FONDI(A[h].i, A[h].j, S);
                INSERISCI(A[h], T)
            end
    end;
```

Per risparmiare iterazioni inutili dopo che è stata raggiunta la soluzione ottima, la procedura KRUSKAL può essere modificata trasformando il ciclo **for** in un **while** ed introducendo una variabile intera che conta il numero di inserimenti di archi nell'albero di copertura T, uscendo dal ciclo appena sono state effettuate $n-1$ INSERISCI. L'ordine di grandezza della complessità resta comunque $O(m \log n)$.

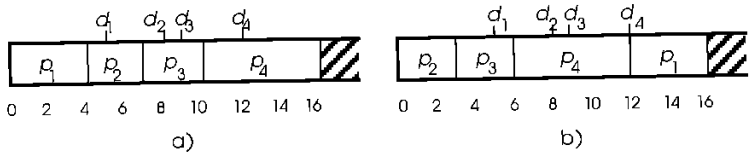


Figura 13.3: Algoritmo di Moore. a) Ordinamento iniziale; b) soluzione ottima.

13.2 Scheduling di programmi

La tecnica greedy è spesso utile per risolvere problemi di scheduling, in cui si hanno dei programmi da eseguire su un processore e si vuole trovarne l'ordine di esecuzione ottimo in base ad un prefissato criterio. A titolo di esempio, consideriamo il seguente problema.

PROGRAMMI IN RITARDO: "Dati n programmi $p_1, \ldots, p_n$, tali che ciascun p_i richiede t_i unità di tempo di esecuzione e deve essere eseguito entro una certa scadenza d_i, trovare un ordine S in cui eseguirli tutti in modo da minimizzare il numero di programmi per i quali la scadenza non è rispettata".

Il problema può essere risolto con un algoritmo ideato da Moore (1968). L'algoritmo è di tipo greedy, in cui viene effettuato occasionalmente anche un backtrack.

I programmi sono ordinati in una sequenza S per scadenze crescenti. Successivamente, si cerca il primo programma p in ritardo e si elimina il programma p' avente tempo di esecuzione più grande tra quelli che stanno nella sottosequenza iniziale di S che termina con p. Il procedimento è iterato sulla sottosequenza S' ottenuta da S eliminando p', fino ad ottenere una sottosequenza S^* senza programmi in ritardo. La sequenza ottima è ottenuta eseguendo dapprima i programmi della sottosequenza S^*, nell'ordine in cui appaiono in S^*, seguiti dai programmi in ritardo in un ordine qualsiasi.

La scelta greedy dell'algoritmo di Moore è basata sul fatto che se in una sottosequenza deve esserci un programma in ritardo, allora tanto vale eliminare il programma più lungo al fine di lasciare il maggior tempo disponibile prima delle scadenze dei programmi che seguiranno quello eliminato. La dimostrazione di correttezza non è qui riportata.

Esempio 13.4 (Programmi in ritardo). La Fig. 13.3 mostra l'esecuzione dell'algoritmo di Moore per i programmi $p_1, \ldots, p_4$ tali che: $t_1 = 4$, $d_1 = 5$, $t_2 = 3$, $d_2 = 8$, $t_3 = 3$, $d_3 = 9$, $t_4 = 6$, $d_4 = 12$. La sequenza S iniziale è $p_1, \ldots, p_4$. Essendo p_3 il primo programma in ritardo, si elimina p_1, perché $t_1 = 5$ è massimo tra t_1, t_2 e t_3. Dopo aver eliminato p_1, nessun altro programma oltre a p_1 è in ritardo, e la sequenza ottima è p_2, p_3, p_4, p_1. Si noti che l'eliminazione di p_2 o p_3 al posto di p_1 provocherebbe il ritardo di due programmi: quello eliminato e p_4. □

Una realizzazione poco accorta dell'algoritmo porta ad una complessità di $O(n^2)$. Infatti si possono calcolare, per tutti i programmi p nella sequenza S, gli istanti in cui terminano la loro esecuzione, che sono dati dalla somma dei tempi di esecuzione di p e dei programmi che lo precedono. Tale calcolo richiede $O(n)$ tempo per tutti i

programmi, effettuando una sola sommatoria. Confrontando poi gli istanti di terminazione dei programmi con le rispettive scadenze, si può determinare il primo programma p in ritardo e quindi eliminare il programma p' più lungo. Anche questa fase richiede $O(n)$ tempo. Poiché il procedimento di eliminazione di un programma dalla sottosequenza si potrebbe iterare per $O(n)$ volte, la complessità risulterebbe $O(n^2)$.

Utilizzando una coda con priorità Q, realizzata con uno heap, le cui operazioni sono opportunamente modificate per trattare elementi (programmi) che non coincidono con le priorità (tempi di elaborazione) [cfr. § 7 e 10], l'algoritmo di Moore può essere realizzato in modo che la sua complessità scenda ad $O(n \log n)$. La seguente procedura Pascal MOORE utilizza due vettori di interi $d[1..n]$ e $t[1..n]$ per le scadenze ed i tempi di esecuzione degli n programmi, rispettivamente, ed un vettore booleano $r[1..n]$ tale che $r[i]$ = **true** se e solo se il programma i è in ritardo. Dopo aver riordinato i vettori per $d[i]$ crescenti, la procedura considera ciascun programma i, per $i = 1, 2, \ldots, n$, e lo inserisce nella coda con priorità Q, usando una variabile intera T per calcolare la somma dei tempi di esecuzione dei programmi inseriti in Q. Se $T \geq d[i]$, allora il programma i è in ritardo e viene estratto da Q il programma j con priorità (tempo di esecuzione $t[j]$) massima, sottraendo quindi $t[j]$ da T, incrementando k, ed impostando $r[j]$ a **true**. Al termine dell'esecuzione, la variabile k contiene il numero dei programmi in ritardo, mentre il vettore r specifica quali programmi sono o non sono in ritardo. La sequenza ottima è ottenuta eseguendo dapprima i programmi con $r[i]$ = **false** in ordine di indici crescenti, seguiti dai programmi con $r[i]$ = **true** in un ordine qualsiasi.

```
procedure MOORE(d: vettore; n: integer; var r: vettore; var k: integer);
    var Q: prioricoda; i, j, T: integer;
    begin
        CREAPRIORICODA(Q); k := 0; T := 0;
        for i := 1 to n do r[i] := false;
        {ordina d[1..n] e t[1..n] per "scadenze" d[i] crescenti};
        for i := 1 to n do begin
            INSERISCI(i, t[i], Q);
            T := T + t[i];
            if T ≥ d[i] then begin
                j := MAX(Q); CANCELLAMAX(Q);
                T := T − t[j]; r[j] := true; k := k + 1
            end
        end
    end;
```

L'ordinamento iniziale richiede $O(n \log n)$ tempo. Poiché il ciclo **for** è ripetuto n volte, ed al suo interno le operazioni più gravose sono INSERISCI e CANCELLAMAX, che costano $O(\log n)$, il ciclo richiede $O(n \log n)$ tempo. Inoltre, la sequenza ottima può essere ottenuta da r in $O(n)$ tempo tramite una scansione. Pertanto, la complessità della procedura MOORE è $O(n \log n)$.

13.3 Matroidi

Moltissimi problemi (ma non tutti) per i quali un algoritmo greedy permette di trovare la soluzione ottima possono essere formulati come problemi su una struttura matematica chiamata matroide. La teoria dei matroidi è stata introdotta da Whitney

(1935) per studiare le proprietà algebriche delle dipendenze lineari, e viene richiamata molto brevemente in questo paragrafo. Una trattazione completa si può trovare nel testo di Lawler (1976).

Un matroide è una coppia $M = (S, F)$, dove S è un insieme finito di elementi ed F è una famiglia di sottoinsiemi di S, chiamati insiemi indipendenti di S, che soddisfa le proprietà seguenti:

1. *Ereditarietà.* L'insieme vuoto appartiene ad F e tutti i sottoinsiemi propri di un insieme di F appartengono anch'essi ad F, ovvero: $\emptyset \in F$, e se $I \in F$ e $J \subseteq I$, allora $J \in F$.
2. *Scambio.* Se due insiemi di F non hanno la stessa cardinalità, allora un elemento che sta soltanto nel sottoinsieme più grande può essere aggiunto all'insieme più piccolo, e l'insieme risultante appartiene ancora ad F, ovvero: se $I \in F$, $J \in F$, e $|I| < |J|$, allora esiste $x \in J - I$ tale che $I \cup \{x\} \in F$.

M è un matroide di una matrice A se S corrisponde all'insieme delle colonne di A, ed un insieme I di colonne appartiene ad F se le colonne in I sono linearmente indipendenti. M è un matroide di un grafo non orientato $G = (N, E)$ se S corrisponde all'insieme E degli archi, mentre un insieme I di archi sta in F se e solo se gli archi in I non formano un circuito. In altri termini, ciascun insieme indipendente di un matroide di un grafo forma una foresta. Il matroide di un grafo G è equivalente al matroide della matrice A di incidenza nodi-archi di G [cfr. § 9], dove gli elementi 0 e 1 della matrice A sono considerati nel campo degli interi modulo 2.

Sui matroidi sono state dimostrate moltissime proprietà, delle quali ne enunciamo nel seguito solo alcune e senza dimostrazione.

Dati un matroide M ed un insieme $I \in F$, un elemento $x \notin I$ è una estensione di I se può essere aggiunto ad I mantenendone l'indipendenza, cioè se $I \cup \{x\} \in F$. Per esempio, se M è un matroide di un grafo, allora un'estensione è un arco che può essere aggiunto ad un insieme di archi senza introdurre un circuito. Un insieme I per il quale non esiste alcuna estensione è detto massimale. Una proprietà molto utile dei matroidi, che si può facilmente dimostrare per assurdo, è che tutti i suoi insiemi indipendenti massimali hanno la stessa cardinalità.

Un matroide è pesato se c'è un peso positivo associato ad ogni elemento $x \in S$. Il peso di un insieme $I \in F$ è dato dalla somma dei pesi degli elementi che appartengono ad I. Molti problemi per i quali la soluzione ottima può essere trovata con un algoritmo greedy possono essere formulati come problemi di trovare un insieme indipendente massimale di peso massimo in un matroide pesato. Per esempio, il problema di trovare il minimo albero di copertura di un grafo G è equivalente al problema di trovare un insieme indipendente massimale di peso massimo nel matroide M del grafo G. Infatti, se c_a è il peso dell'arco a nel grafo G, allora il peso p_a dello stesso arco nel matroide M è definito come $p_a = C - c_a$, dove C è un numero più grande del massimo peso degli archi di G. In questo modo, i pesi di M sono positivi e trovare l'insieme indipendente massimale di peso massimo in M corrisponde a trovare il minimo albero di copertura di G. Infatti, $p(I) = \sum_{a \in I} p_a$ è il peso di un insieme indipendente I di M, mentre $c(I) = \sum_{a \in I} c_a$ è quello di un albero di copertura di G formato dagli stessi archi. Per costruzione, $p(I) = (n-1)C - c(I)$, dove n è il numero di nodi di G, e quindi massimizzare $p(I)$ è equivalente a minimizzare $c(I)$.

La caratteristica più importante dal punto di vista algoritmico dei matroidi pesati è che il metodo greedy verifica le proprietà della "scelta greedy" e della "sottostruttura ottima" e che pertanto trova sempre l'insieme indipendente massimale di peso massimo! Riformulato su un matroide pesato $M = (S, F)$ con funzione peso p (positiva), l'algoritmo greedy diventa:


```
procedure GREEDY-MATROID(matroide M = (S, F) con funzione peso p);
  begin
    I := ∅;
    {ordina gli elementi di S per peso decrescente};
    for ciascun x ∈ S considerato per peso p(x) decrescente do
      if I ∪ {x} ∈ F
        then I := I ∪ {x};
    {restituisci I come risultato}
  end;
```

La complessità dell'algoritmo dipende essenzialmente dal tempo richiesto per verificare che $I \cup \{x\}$ sia indipendente. Se tale tempo è $O(f(n))$, dove n è la cardinalità di S, allora la complessità risultante è $O(n \log n + nf(n))$.

13.4 Esercizi

Esercizio 13.1 (Zaino reale). Dati un insieme di n oggetti, con profitti $\{p_1, \ldots, p_n\}$ e volumi $\{v_1, \ldots, v_n\}$, tutti interi positivi, e un intero positivo c, la capacità dello zaino, il problema dello ZAINO REALE consiste nello scegliere oggetti o porzioni di essi in modo da massimizzare il profitto degli oggetti scelti senza superare la capacità dello zaino, ovvero determinare un insieme $\{x_1, \ldots, x_n\}$ di reali compresi tra 0 e 1 tali che $\sum_{1 \le i \le n} p_i x_i$ sia massima e $\sum_{1 \le i \le n} v_i x_i \le c$. Si progetti un algoritmo di tipo greedy per risolvere in tempo polinomiale il problema dello ZAINO REALE.

Si assuma che i profitti e i volumi siano memorizzati nei vettori $p[1..n]$ e $v[1..n]$. Per applicare la tecnica greedy si riordinano dapprima tali vettori per rapporto profitto/volume decrescente. Si cerca poi di scegliere la porzione più grande possibile dell'oggetto che dà il maggior profitto per unità di volume, senza superare la capacità c, ed una volta effettuata la scelta c è aggiornato con la capacità residua nello zaino. Se lo zaino non è riempito, allora si ripete il procedimento col secondo oggetto che dà maggior profitto per unità di volume, e così via, sempre aggiornando la capacità residua. Alla fine, saranno scelti per intero gli oggetti più appetibili, può essere scelta una frazione dell'oggetto immediatamente successivo, ed i rimanenti oggetti non saranno scelti. La seguente procedura ZAINO restituisce il vettore reale $x[1..n]$ e richiede tempo $O(n \log n)$.

```
procedure ZAINO(var p, v: vettore-intero; c: integer; var x: vettore-reale);
  var i: integer;
  begin
    {ordina p e v in modo che p[1]/v[1] ≤ p[2]/v[2] ≤ ... ≤ p[n]/v[n]};
    for i := 1 to n do x[i] := 0;
    i := 1;
    while (i ≤ n) and (c ≥ 0) do begin
      if v[i] ≥ c then begin x[i] := c/v[i]; c := 0; end
      else begin x[i] := 1; c := c − v[i]; end;
      i := i + 1;
    end;
  end;
```

Esercizio 13.2 (Zaino 0-1). Si dimostri con un controesempio che, al contrario dello ZAINO REALE, il problema dello ZAINO 0-1 non si può risolvere con una tecnica greedy (nello ZAINO 0-1 si richiede di prendere o lasciare un oggetto senza la possibilità di sceglierne una frazione, cioè x_i deve essere un intero uguale a 0 oppure ad 1, per $1 \leq i \leq n$).

Esercizio 13.3 (Algoritmo di Kruskal). Si assuma che i pesi degli archi di un grafo non orientato G siano compresi nell'intervallo $1..n$, dove n è il numero di nodi di G. È possibile abbassare la complessità dell'algoritmo di Kruskal?

Esercizio 13.4 (Algoritmo di Prim). Si consideri il seguente algoritmo per costruire il minimo albero di copertura T di un grafo G:

```
procedure PRIM(grafo non orientato pesato G = (N, A));
  begin
    considera T formato da un nodo qualsiasi e nessun arco;
    while esistono nodi non in T adiacenti a un nodo in T do begin
      seleziona l'arco di peso minimo tra quelli che connettono un
          nodo in T e un nodo non in T;
      aggiungi a T sia l'arco selezionato che il nodo che non era in T;
    end;
    restituisci T come il minimo albero di copertura di G;
  end;
```

Si esegua l'algoritmo di Prim sul grafo dell'Esempio 13.1. Si dimostri che l'algoritmo è corretto provando per induzione che l'insieme di archi selezionato ad ogni iterazione è un sottoinsieme degli archi di un minimo albero di copertura. Si realizzi l'algoritmo con una procedura Pascal in modo che abbia complessità $O(n^2)$. Quando è più vantaggioso l'algoritmo di Prim e quando invece quello di Kruskal?

Esercizio 13.5 (Programmi in ritardo). Si dimostri che ogni sequenza ottima per il problema dei PROGRAMMI IN RITARDO può essere trasformata in una sequenza che comprende dapprima i programmi non in ritardo ordinati per scadenze crescenti seguiti da quelli in ritardo in un ordine qualsiasi.

Data una sequenza ottima S_{OTT}, essa può essere sempre trasformata nella sequenza ottenuta con le scelte "greedy", dove i programmi in orario (cioè non in ritardo) sono ordinati in una sottosequenza iniziale per scadenze crescenti, mentre tutti i programmi in ritardo seguono quelli in orario. Infatti, se in S_{OTT} un programma in ritardo precede uno in orario, allora, spostando il programma in ritardo all'ultimo posto di S_{OTT}, il numero di programmi in ritardo non aumenta. Pertanto tutti i programmi in orario possono precedere quelli in ritardo. Consideriamo nella sottosequenza iniziale dei programmi in orario la minima scadenza d_j tale che p_j è preceduto da p_i con $d_i > d_j$ (se ci sono più p_i, allora sono considerati uno alla volta per d_i crescenti). Togliendo p_i e spostandolo immediatamente dopo p_j, i tempi di terminazione dei programmi che seguivano p_i fino a p_j (incluso) diminuiscono, mentre quelli di p_i e dei programmi che lo seguono restano immutati, e quindi il numero dei programmi in ritardo non cambia. Ripetendo il procedimento considerando sempre la minima scadenza d_j, tutti i programmi in orario possono essere ordinati per scadenze crescenti.

Esercizio 13.6 (Matrice di incidenza). Sia data la matrice di incidenza nodi-archi di un grafo non orientato. Si dimostri che un insieme di colonne della matrice è linearmente indipendente se e solo se gli archi corrispondenti a tali colonne non formano un circuito.

Esercizio 13.7 (Cardinalità insiemi indipendenti massimali). Si dimostri per assurdo che tutti gli insiemi indipendenti massimali di un matroide hanno la stessa cardinalità.

Esercizio 13.8 (Massimizzazione e minimizzazione). Si dimostri se MASSIMO ALBERO DI COPERTURA e CAMMINI MASSIMI sono equivalenti oppure no a MINIMO ALBERO DI COPERTURA e CAMMINI MINIMI.

Capitolo 14

Programmazione dinamica

Una filosofia simile al "divide et impera", ma portata all'estremo, si basa sulla massima seguente:

"Il pesce grosso mangia tutti i pesci più piccoli".

In altri termini, se non si sa con esattezza quali sottoproblemi risolvere, si può provare a risolverli tutti e conservare i risultati ottenuti per poterli usare successivamente nel risolvere il problema di dimensione più grande. Questa tecnica è detta programmazione dinamica, perché indica un processo di programmazione (o pianificazione) di un metodo risolutivo che sia applicabile dinamicamente per dimensioni via via crescenti del problema. Rispetto al "divide et impera", la programmazione dinamica presenta tre grandi differenze: è iterativa e non ricorsiva, affronta i sottoproblemi "dal basso verso l'alto" anziché "dall'alto verso il basso", e memorizza i risultati dei sottoproblemi in una tabella. La proprietà di risolvere i sottoproblemi per dimensioni crescenti differenzia la programmazione dinamica non solo dal "divide et impera", ma anche dalla strategia greedy, che invece effettua una sequenza di scelte ingorde riducendo il problema ad un sottoproblema sempre più piccolo. Comunque, mentre il "divide et impera" risulta vantaggioso quando i sottoproblemi più piccoli da risolvere sono indipendenti, la programmazione dinamica è conveniente quando i sottoproblemi non sono indipendenti, ma ne condividono altri. Procedendo dal basso verso l'alto, la programmazione dinamica risolve un sottoproblema comune una sola volta, mentre il "divide et impera" lo risolverebbe più volte.

Esempio 14.1 (Coefficiente binomiale). $C(n, k) = n!/(k!(n-k)!)$ rappresenta il numero di modi di scegliere k oggetti da un insieme di n oggetti, con $0 \leq k \leq n$, ed è definibile ricorsivamente come segue:

$$C(n,k) = \begin{cases} 1 & \text{se } k = 0 \text{ oppure } n = k, \\ C(n-1,k-1) + C(n-1,k) & \text{altrimenti} \end{cases}$$

Un algoritmo ricorsivo "divide et impera" si ottiene immediatamente dalla precedente definizione ricorsiva:

```
function C(n, k: integer): integer;
    begin
        if (n = k) or (k = 0)
            then C := 1
            else C := C(n − 1, k − 1) + C(n − 1, k)
    end;
```


Purtroppo, la complessità dell'algoritmo "divide et impera" cresce come il numero di chiamate ricorsive, che è proprio $C(n, k) = n!/(k!(n-k)!)$. Questo è dovuto all'elevato numero di sottoproblemi identici che vengono risolti più volte. Per esempio già $C(n-2, k-1)$ è richiamata due volte, sia per calcolare $C(n-1, k)$ che $C(n-1, k-1)$, mentre al decrescere dei valori degli argomenti ciascun sottoproblema è risolto per un numero sempre maggiore di volte. Un algoritmo di programmazione dinamica, invece, risolve i sottoproblemi per valori crescenti degli argomenti secondo il ben noto schema del "triangolo di Tartaglia", memorizzando i risultati una volta per tutte in una matrice intera $C[0..n, 0..n]$ in $\Theta(n^2)$ tempo:

```
procedure TARTAGLIA(n: integer; var C: matrice): integer;
    var i, j: integer;
    begin
        C[0, 0] := 1;
        for i := 1 to n do begin
            C[i, 0] := 1;
            C[i, i] := 1
        end;
        for i := 2 to n do
            for j := 1 to i − 1 do
                C[i, j] := C[i − 1, j − 1] + C[i − 1, j];
    end;
```

In questo modo, il valore di $C(m, k)$, per una qualsiasi coppia di interi m e k tali che $0 \leq k \leq m \leq n$, può essere successivamente letto direttamente dalla matrice C in tempo $O(1)$. □

La tecnica della programmazione dinamica è stata introdotta da Bellman (1957) per progettare algoritmi che risolvono problemi di ottimizzazione. Perché sia applicabile, occorre che:

1. sia possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più grande;
2. le decisioni prese per risolvere in modo ottimo un sottoproblema rimangano valide quando il sottoproblema diviene un "pezzo" di un problema più grande;
3. una tecnica "divide et impera" richieda di risolvere più volte gli stessi sottoproblemi e sia pertanto inutilizzabile da un punto di vista computazionale.

Le proprietà (1) e (2) corrispondono alla proprietà della sottostruttura ottima che deve essere verificata perché un algoritmo greedy fornisca la soluzione ottima. In pratica, perché un algoritmo basato sulla programmazione dinamica abbia complessità polinomiale, occorre inoltre che:

4. ci sia un numero polinomiale di sottoproblemi da risolvere;
5. per evitare di risolvere più di una volta lo stesso sottoproblema, si usi una tabella in cui si memorizzano le soluzioni di tutti i sottoproblemi, senza preoccuparsi se la soluzione di un particolare sottoproblema verrà poi utilizzata oppure no;
6. il tempo per combinare le soluzioni dei sottoproblemi e trovare la soluzione del problema più grande sia polinomiale.

In questo capitolo, si vedranno esempi di applicazione di questa tecnica per risolvere due generalizzazioni di problemi già affrontati precedentemente: lo string matching approssimato e i cammini minimi tra tutte le coppie di nodi.

14.1 String matching approssimato

Il problema dello STRING MATCHING APPROSSIMATO consiste nel trovare una occorrenza del pattern P nel testo T in cui sono ammessi errori (o differenze) tra caratteri di P e caratteri di T. Siano P e T due sequenze di m ed n caratteri, rispettivamente, con $m \leq n$, entrambe definite su un alfabeto A [cfr. § 12]. Una occorrenza k-approssimata di P in T, con $0 \leq k \leq m$, è una copia della stringa P nella stringa T in cui sono presenti k "errori" del seguente tipo:

1. i corrispondenti caratteri in P e in T sono diversi;
2. un carattere in P non compare in T;
3. un carattere in T non compare in P.

Il problema consiste nel trovare un'occorrenza k-approssimata di P in T per la quale k sia minimo. Si noti che lo STRING MATCHING discusso nel § 12 si ottiene imponendo $k = 0$.

Esempio 14.2 (Occorrenza 2-approssimata). Siano T = QUESTOÈUNOSCEMPIO e P = UNESEMPIO, dove $A = \{$A, B, ..., Z, È$\}$, $m = 9$, $n = 18$. Un'occorrenza 2-approssimata di P si ha a partire dalla posizione 9 di T: QUESTOÈUNOSCEMPIO. Infatti, la prima E di P corrisponde ad una O di T (errore di tipo 1), mentre la C di T è un carattere che non compare in P (errore di tipo 3). □

Per comodità, assumiamo che sia P che T inizino con un carattere "vuoto" λ, che si trova in posizione 0 con le funzioni di sentinella, ed indichiamo P e T con $p_0p_1 \cdots p_m$ e $t_0t_1 \cdots t_n$, rispettivamente. Per applicare la programmazione dinamica, definiamo $D[i,j]$, con $0 \leq i \leq m$ e $0 \leq j \leq n$, come segue:

$D[i,j]$ = il minimo numero di errori tra $p_0p_1 \cdots p_i$ ed un segmento di T che termina in t_j.

Chiaramente, $D[0,j] = 0$, per $0 \leq j \leq n$, poiché $p_0 = \lambda$ occorre in ogni posizione di T, mentre $D[i,0] = i$, per $1 \leq i \leq m$, perché $p_1 \cdots p_i$ ha i caratteri in più di $t_0 = \lambda$. È possibile definire $D[i,j]$, per $1 \leq i \leq m$ e $1 \leq j \leq n$, in funzione dei tre valori seguenti, che tengono conto di quale tipo di errore può occorrere tra p_i e t_j:

1. $D[i-1, j-1]$, se $p_i = t_j$, oppure $D[i-1, j-1] + 1$, se $p_i \neq t_j$, perché "avanzando" di un carattere sia nel pattern che nel testo il numero di errori resta identico se i due caratteri sono uguali mentre aumenta di uno se sono diversi (errore di tipo 1);
2. $D[i-1, j] + 1$, perché "avanzando" di un carattere solo nel pattern il numero di errori aumenta di uno in quanto p_i non compare in T (errore di tipo 2);
3. $D[i, j] = D[i, j-1] + 1$, perché "avanzando" di un carattere solo nel testo il numero di errori aumenta di uno in quanto t_j non compare in P (errore di tipo 3).

Come illustrato nella Fig. 14.1, si ottengono pertanto le seguenti relazioni di programmazione dinamica, per $1 \leq i \leq m$ e $1 \leq j \leq n$:

$$D[i,j] = \min\{D[i-1,j-1] + \delta, D[i-1,j] + 1, D[i,j-1] + 1\},$$

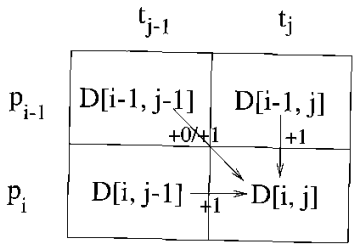


Figura 14.1: Calcolo di $D[i,j] = \min\{D[i-1,j-1] + \delta, D[i-1,j] + 1, D[i,j-1] + 1\}$, dove $\delta = 0$, se $p_i = t_j$, oppure $\delta = 1$, se $p_i \neq t_j$.

P \ T		A	B	A	B	A
	0	0	0	0	0	0
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

Figura 14.2: Risoluzione dello string matching approssimato con la programmazione dinamica.

dove $\delta = 0$, se $p_i = t_j$, oppure $\delta = 1$, se $p_i \neq t_j$. Le condizioni iniziali sono:

$$D[0,j] = 0, \quad \text{per } 0 \leq j \leq n,$$
$$D[i,0] = i, \quad \text{per } 1 \leq i \leq m.$$

È chiaro che $D[m,j] = k$ se e solo se c'è un'occorrenza k-approssimata di P in T che termina in t_j e che la soluzione del problema è data dal valore di $D[m,j]$ più piccolo, per $0 \leq j \leq n$.

Esempio 14.3 (String matching approssimato). La Fig. 14.2 mostra la matrice D ottenuta con le relazioni di programmazione dinamica per T = ABABA e P = BAB, dove $m = 3$ ed $n = 5$. Un'occorrenza 0-approssimata di P termina nella posizione 4 di T: ABABA. Nella figura, le frecce indicano la provenienza del minimo; ripercorrendo a ritroso tali frecce a partire dall'ultima riga si ricavano gli errori e si ricostruiscono le occorrenze approssimate di P in T. □

La seguente funzione STRINGMATCHING computa la matrice D e restituisce la posizione di T in cui termina un'occorrenza approssimata di P col numero minimo di errori:

```
function STRINGMATCHING(var P, T: vettore; m, n: integer): integer;
    var i, j, min: integer; D: matrice;
    begin
        for j := 0 to n do D[0, j] := 0;
        for i := 1 to m do D[i, 0] := i;
        for i := 1 to m do
            for j := 1 to n do begin
                if P[i] = T[j] then min := D[i − 1, j − 1]
                    else min := D[i − 1, j − 1] + 1;
                if D[i − 1, j] + 1 < min then min := D[i − 1, j] + 1;
                if D[i, j − 1] + 1 < min then min := D[i, j − 1] + 1;
                D[i, j] := min
            end;
        min := D[m, 0];
        for j := 1 to n do
            if D[m, j] < min then begin min := D[m, j]; i := j end;
        STRINGMATCHING := i
    end;
```

La complessità della funzione è $\Theta(mn)$, come la RICERCABRUTA vista nel § 12, poiché sono valutati esplicitamente tutti gli elementi di D. La funzione può essere facilmente modificata per esercizio senza aumentare la complessità in modo da poter ricavare le occorrenze approssimate, ed i relativi errori, come accennato nell'Esempio 14.3. Qualora sia richiesto, dato k, di trovare un'occorrenza approssimata di P in T con al più k errori, il problema può essere risolto in tempo $O(kn)$, che diviene $O(n)$ per $k = 0$, come per l'algoritmo di Knuth, Morris e Pratt, ma il relativo algoritmo è complicato e non può essere qui riportato. Tale algoritmo, ideato da Landau e Vishkin (1988), riesce a valutare esplicitamente solo quegli elementi di D che provocano un incremento lungo una diagonale, cioè quelli per cui $D[i, j] > D[i-1, j-1]$, ed utilizza una struttura di dati ad albero detta "albero dei suffissi".

14.2 Cammini minimi tra tutte le coppie

Nel § 10 sono stati trattati diversi algoritmi per trovare i cammini minimi tra un dato nodo r e tutti i rimanenti nodi. Per trovare i cammini minimi tra tutte le coppie di nodi, è possibile applicare n volte uno degli algoritmi visti, considerando a turno come nodo "radice" r ciascuno degli n nodi del grafo. Se non ci sono archi con lunghezze negative, tale procedimento è $O(nm \log n)$, utilizzando l'algoritmo di Johnson, e $O(n^3)$ utilizzando quello di Dijkstra. Se ci sono lunghezze negative, invece, la complessità è $O(n^4)$, poiché si deve usare n volte l'algoritmo di Bellman, Ford e Moore.

Vediamo come la programmazione dinamica permetta di progettare due algoritmi di complessità inferiore ad $O(n^4)$ anche per grafi con lunghezze negative degli archi.

14.2.1 Quadratura di matrici

Dato il grafo orientato pesato $G = (N, A)$, ove c_{ij} sono le lunghezze degli archi, sia $d_{ij}^{(k)}$ la lunghezza di un cammino minimo dal nodo i al nodo j che contiene al più k archi. È facile osservare che un cammino minimo da i a j che comprende al più $2k$ archi può essere ottenuto da un cammino minimo tra i ed un nodo intermedio h che comprende al più k archi, seguito da un cammino minimo tra h e j che comprende anch'esso al più k archi. Valgono pertanto le seguenti relazioni di programmazione dinamica:

$$d_{ij}^{(2k)} = \min_{1 \le h \le n} \{d_{ih}^{(k)} + d_{hj}^{(k)}\}, \qquad 1 \le i \le n, 1 \le j \le n,$$

con condizioni iniziali:

$$d_{ij}^{(1)} = \begin{cases} c_{ij} & \text{se } (i,j) \in A, \\ 0 & \text{se } i = j, \\ \infty & \text{altrimenti.} \end{cases}$$

Per risolvere il problema, è quindi sufficiente calcolare i $d_{ij}^{(k)}$, iterando per $k = 1, 2, 4, 8, \ldots, 2^{\lceil \log(n-1) \rceil}$, cioè raddoppiando ogni volta finché k non raggiunge o supera $n - 1$. Il calcolo di $d_{ij}^{(2k)}$ è analogo a quello di una moltiplicazione di una matrice quadrata $n \times n$ $D_k = [d_{ij}]$ per se stessa, dove $d_{ij} = d_{ij}^{(k)}$. Per convincersi di ciò, basta confrontare le formule

$$\min_{1 \le h \le n} \{d_{ih} + d_{hj}\} \qquad \text{e} \qquad \sum_{1 \le h \le n} d_{ih} \cdot d_{hj}$$

sostituendo gli operatori "min" e "+" della formula di sinistra con "$\sum$" e "·" della formula di destra, e tenendo conto che k è solo un indice di iterazione che può essere ignorato. Pertanto, ad ogni iterazione, il calcolo di $d_{ij}^{(2k)}$ può essere effettuato in tempo $\Theta(n^3)$ con un algoritmo analogo a quello "banale" per il prodotto di matrici [cfr. § 16.1.1]. Poiché il numero totale di iterazioni è $\Theta(\log n)$, la complessità dell'algoritmo è $\Theta(n^3 \log n)$.

14.2.2 Algoritmo di Floyd

Dato G, sia $d_{ij}^{(k)}$ la lunghezza di un cammino minimo dal nodo i al nodo j che non attraversa i nodi $k, k+1, k+2, \ldots, n-1, n$ (con l'esclusione dei nodi i e j). È facile osservare che un cammino minimo da i a j, che non attraversa i nodi $k+1, k+2, \ldots, n$, o non passa neanche per il nodo k, e allora $d_{ij}^{(k+1)} = d_{ij}^{(k)}$, o include anche il nodo k, e allora $d_{ij}^{(k+1)} = d_{ik}^{(k)} + d_{kj}^{(k)}$. Valgono quindi le seguenti relazioni di programmazione dinamica:

$$d_{ij}^{(k+1)} = \min_{1 \le k \le n} \{d_{ij}^{(k)}, d_{ik}^{(k)} + d_{kj}^{(k)}\}, \qquad 1 \le i \le n, 1 \le j \le n,$$

con le stesse condizioni iniziali per $d_{ij}^{(1)}$ viste per l'algoritmo precedente. Per ricavare, oltre alla lunghezza $d_{ij}^{(k)}$, anche il cammino minimo, si definisce in modo analogo $p_{ij}^{(k)}$

come il predecessore del nodo j in un cammino minimo dal nodo i al nodo j che non attraversa i nodi $k, k+1, k+2, \ldots, n-1, n$ (con l'esclusione dei nodi i e j). Per $1 \leq k \leq n$, si ottiene:

$$p_{ij}^{(k+1)} = \begin{cases} p_{ij}^{(k)} & \text{se } d_{ij}^{(k+1)} = d_{ij}^{(k)}, \\ p_{kj}^{(k)} & \text{se } d_{ij}^{(k+1)} = d_{ik}^{(k)} + d_{kj}^{(k)}, \end{cases}$$

con condizioni iniziali:

$$p_{ij1} = \begin{cases} i & \text{se } (i,j) \in A, \\ 0 & \text{altrimenti.} \end{cases}$$

È facile convincersi che $d_{ij}^{(n+1)}$ è la lunghezza di un cammino minimo dal nodo i al nodo j nel grafo G, mentre $p_{ij}^{(n+1)}$ è il predecessore del nodo j in un cammino siffatto. Si noti che se il nodo j non è raggiungibile dal nodo i, allora $d_{ij}^{(n+1)} = \infty$, mentre se esiste un ciclo negativo che passa per il nodo i, allora $d_{ii}^{(n+1)} < 0$.

Rappresentando il grafo con la matrice delle lunghezze $c = [c_{ij}]$ (ed con *maxint*), la computazione di $d_{ij}^{(n+1)}$ e $p_{ij}^{(n+1)}$, per $1 \leq i \leq n$, $1 \leq j \leq n$, può essere effettuata con la seguente procedura, proposta da Floyd (1962). La procedura usa iterativamente la riga k e la colonna k delle matrici d e p per aggiornare "in loco" i rimanenti elementi delle matrici stesse.

```
procedure FLOYD(var c, d, p: matrice; n: integer);
    var i, j, k: integer;
    begin
        {inizializza le matrici d e p utilizzando c};
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to n do
                    if d[i, j] > d[i, k] + d[k, j] then
                        begin
                            d[i, j] := d[i, k] + d[k, j];
                            p[i, j] := p[k, j]
                        end
    end;
```

Il tempo richiesto della procedura è $\Theta(n^3)$. La procedura utilizza soltanto $O(n^2)$ spazio di memoria, in quanto i valori $d_{ij}^{(k)}$ e $p_{ij}^{(k)}$, per ogni iterazione k, sono sempre riscritti nelle matrici d e p stesse.

Esempio 14.4 (Algoritmo di Floyd). La Fig. 14.3 mostra l'esecuzione dell'algoritmo di Floyd su un grafo G avente 4 nodi e 6 archi. Per ogni iterazione k, sono evidenziate in grassetto la k-esima riga e la k-esima colonna delle matrici d e p. Gli elementi delle matrici all'iterazione $(k+1)$-esima sono ottenuti applicando la cosiddetta regola del rettangolo rispetto alla la k-esima riga e alla k-esima colonna. Per esempio, per $k+1 = 3$, l'elemento $d_{14}^{(3)}$ (Fig. 14.3c) è ottenuto da $\min\{d_{14}^{(2)}, d_{12}^{(2)} + d_{24}^{(2)}\} =$

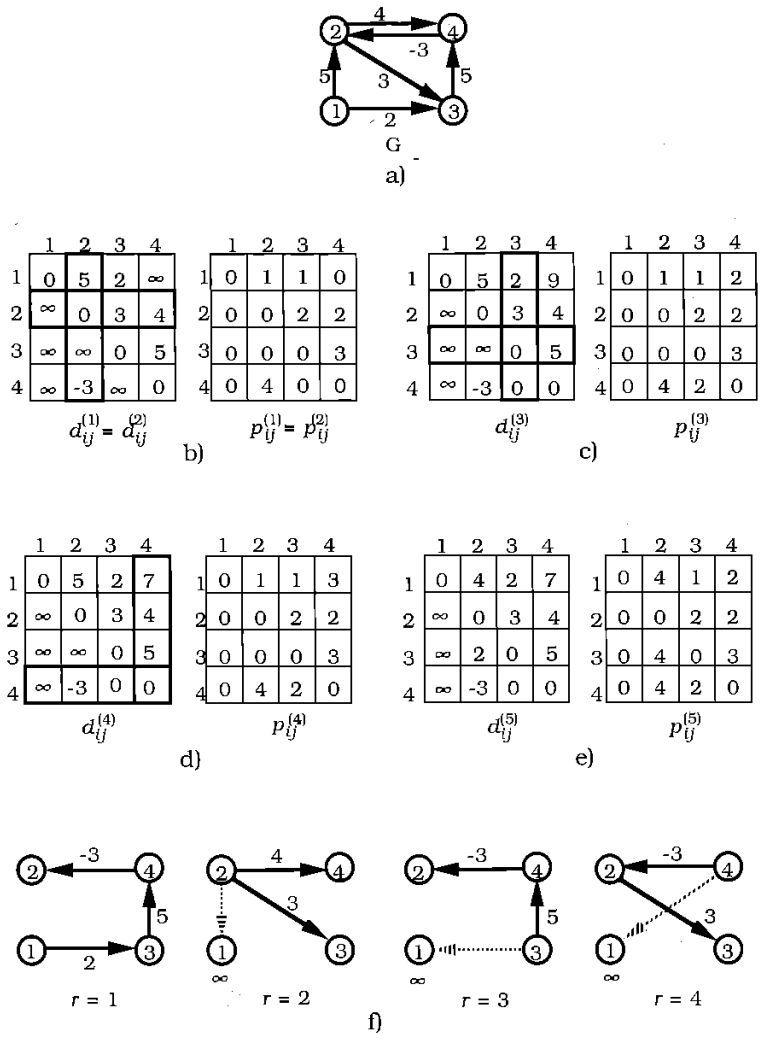


Figura 14.3: Algoritmo di Floyd. a) Un grafo G di 4 nodi; b)-e) matrici d e p per $k = 1, 2, 3, 4, 5$; f) alberi dei cammini minimi per $r = 1, 2, 3, 4$.

$\min\{\infty, 5+4\} = 9$ (Fig. 14.3b), mentre $p_{14}^{(3)}$ è ottenuto da $p_{24}^{(2)} = 2$. L'i-esima riga della matrice p finale contiene l'albero dei cammini minimi di radice i, rappresentato come vettore dei padri, dove un nodo non raggiungibile da i ha padre uguale a 0 e distanza uguale ad ∞. □

14.3 Esercizi

Esercizio 14.1 (Numeri di Fibonacci). Dato un intero $n \geq 0$, l'n-esimo numero di Fibonacci F_n è definito ricorsivamente come segue: $F_0 = 0$, $F_1 = 1$, ed $F_n = F_{n-1} + F_{n-2}$ per $n > 1$. È possibile dimostrare che il valore di F_n è circa $(1.6)^n$ e pertanto il calcolo di F_n con un algoritmo "divide et impera" basato sulla definizione ricorsiva richiede un numero di chiamate che cresce anch'esso come $(1.6)^n$. Si scriva una procedura Pascal basata sulla programmazione dinamica per calcolare F_n in tempo $O(n)$.

Esercizio 14.2 (Massima sottostringa comune). Date due stringhe $P = p_1p_2 \cdots p_m$ e $T = t_1t_2 \cdots t_n$ di caratteri alfabetici, si progetti un algoritmo di programmazione dinamica per individuare la più lunga sottostringa comune tra P e T.

Per comodità, assumiamo che P e T inizino con un carattere "vuoto" λ, che si trova in posizione 0 con le funzioni di sentinella. Si definisce una matrice $D[i,j]$, con $0 \leq i \leq m$ e $0 \leq j \leq n$, come segue:

$D[i,j]$ è il numero dei caratteri della sottostringa comune a $p_0p_1 \cdots p_i$ e $t_0t_1 \cdots t_j$ che termina in posizione i di P e in posizione j di T.

Chiaramente, $D[0,j] = 0$, per $0 \leq j \leq n$, poiché t_0 è il carattere vuoto, ed anche $D[i,0] = 0$, per $0 \leq i \leq m$, perché p_0 è vuoto. Valgono le seguenti relazioni di programmazione dinamica:

$$D[i,j] = \begin{cases} 0 & \text{se } p_i \neq t_j, \\ D[i-1,j-1]+1 & \text{se } p_i = t_j. \end{cases}$$

Il massimo elemento $D[i,j]$ della matrice così ottenuta individua la sottostringa più lunga comune a P e T. L'algoritmo, che può essere facilmente realizzato con una procedura Pascal, ha complessità $\Theta(mn)$.

Esercizio 14.3 (Prodotto di una sequenza di matrici). Si possono moltiplicare due matrici rettangolari A e B solo se il numero di colonne di A è uguale al numero di righe di B. Se A è una matrice $p \times q$ e B una matrice $q \times r$, il loro prodotto è una matrice $p \times r$ e richiede pqr operazioni aritmetiche di moltiplicazione tra elementi di A e B. La moltiplicazione tra matrici è associativa, ma il numero di operazioni aritmetiche dipende dall'ordine in cui le matrici sono moltiplicate. Si progetti un algoritmo di programmazione dinamica per trovare il minimo numero di moltiplicazioni aritmetiche necessarie per eseguire il prodotto di una sequenza di matrici $A_1 \cdots A_n$ tali che, per $i = 2, \ldots, n$, ciascuna A_i abbia r_{i-1} righe e r_i colonne.

Il costo per moltiplicare una qualsiasi sequenza $A_iA_{i+1} \cdots A_j$ è dato dal costo per moltiplicare due sottosequenze $A_i \cdots A_k$ e $A_{k+1} \cdots Aj$ più $r_{i-1}r_kr_j$ moltiplicazioni aritmetiche per moltiplicare tra loro le due matrici $r_{i-1} \times r_k$ ed $r_k \times r_j$ risultanti. Sia $M[i,j]$ il minimo numero di moltiplicazioni aritmetiche necessarie per calcolare $A_iA_{i+1} \cdots A_j$, per $1 \leq i \leq j \leq n$. Si ottengono le seguenti relazioni di programmazione dinamica:

$$M[i,j] = \min_{i \leq k \leq j-1} \{M[i,k] + M[k+1,j] + r_{i-1}r_kr_j\}, \quad \text{per } 1 \leq i < j \leq n,$$
$$M[i,i] = 0, \quad \text{per } 1 \leq i \leq n.$$

La seguente procedura Pascal calcola la matrice M, che viene riempita considerando il fatto che il costo per moltiplicare le sequenze di $h = j - i + 1$ matrici dipende soltanto dal costo per moltiplicare sequenze con meno di h matrici, per $h = 1, 2, 3, \ldots, n$. Fissato h, gli estremi i e j della sequenza $A_iA_{i+1} \cdots A_j$ di h matrici sono tali che $1 \leq i \leq n - h + 1$, mentre $j = i + h - 1$.

```
procedure CONTA-PRODOTTI(var M: matrice; r: vettore; n: integer);
    var i, j, h, k, min: integer;
    begin
        for i := 1 to n do M[i, i] := 0;
        for h := 2 to n do
            for i := 1 to n − h + 1 do begin
                j := i + h − 1;
                M[i, j] := maxint;
                for k := i to j − 1 do begin
                    min := M[i, k] + M[k + 1, j] + r[i − 1] * r[k] * r[j];
                    if min < M[i, j] then M[i, j] := min;
                end;
            end;
    end;
```

La complessità della procedura CONTA-PRODOTTI è $O(n^3)$.

Esercizio 14.4 (Prodotto di una sequenza di matrici). Si riconsideri l'Esercizio 14.3 dove, oltre al numero minimo di moltiplicazioni aritmetiche, si vuole determinare anche l'ordine in cui moltiplicare le matrici.

Esercizio 14.5 (Algoritmo di Bellman,Ford e Moore). Si consideri il problema dei cammini minimi da un singolo nodo r a tutti i rimanenti $n - 1$ nodi in un grafo orientato pesato (con eventuali lunghezze negative degli archi). Si definisca $d_j^{(k)}$ come la lunghezza di un cammino minimo dal nodo r al nodo j che contiene al più k archi. Si dimostri che l'algoritmo di Bellman-Ford-Moore [cfr. § 10] equivale a risolvere le seguenti relazioni di programmazione dinamica:

$$d_j^{(k+1)} = \min\{d_j^{(k)}, \min_{h \neq j}\{d_h^{(k)} + c_{hj}\}\}.$$

Qual'è la complessità dell'algoritmo nella versione di programmazione dinamica?

Esercizio 14.6 (Cammini minimi su grafi aciclici). Si consideri il problema dei cammini minimi da un singolo nodo r a tutti i rimanenti nodi in un grafo (con eventuali lunghezze negative degli archi) nel quale esiste l'arco (i,j) solo se $i < j$. Si dimostri che le relazioni di programmazione dinamica dell'Esercizio 14.5 diventano semplicemente $d_j = \min_{h<j}\{d_h + c_{hj}\}$, dove d_j è la lunghezza di un cammino minimo dal nodo r al nodo j. Qual'è la complessità dell'algoritmo?

Esercizio 14.7 (Massima sottosequenza crescente). Sia data una sequenza L di n numeri interi distinti. Si scriva una procedura Pascal efficiente basata sulla programmazione dinamica per trovare la più lunga sottosequenza crescente di L (per esempio, se $L = 9, 15, 3, 6, 4, 2, 5, 10, 3$, allora la più lunga sottosequenza crescente è: 3, 4, 5, 10).

Esercizio 14.8 (Massima sottosequenza comune). Siano date due stringhe $P = p_1p_2 \cdots p_m$ e $T = t_1t_2 \cdots t_n$ di caratteri alfabetici. Si progetti un algoritmo di programmazione dinamica per individuare la più lunga sottosequenza comune tra P e T (per esempio, se $P = 9, \underline{15}, 3, \underline{6}, 4, \underline{2}, 5, 10, \underline{3}$ e $T = 8, \underline{15}, \underline{6}, 7, 9, \underline{2}, 11, \underline{3}, 1$ allora il risultato è: 15, 6, 2, 3).

Si noti che, a differenza dell'Esercizio 14.1, non si richiede che i caratteri della massima sottosequenza comune appaiano consecutivamente in P e T. Assumendo come al solito che P e T inizino con un carattere "vuoto" λ, che si trova in posizione 0, si definisce una matrice D tale che $D[i, j]$ è il numero dei caratteri della massima sottosequenza comune tra $p_0p_1 \cdots p_i$ e $t_0t_1 \cdots t_j$. Valgono le seguenti relazioni di programmazione dinamica:

$$D[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ oppure } j = 0, \\ D[i-1,j-1]+1 & \text{se } i > 0, j > 0, \text{ e } p_i = t_j, \\ \max\{D[i,j-1], D[i-1,j]\} & \text{se } i > 0, j > 0, \text{ e } p_i \neq t_j. \end{cases}$$

Il valore $D[m, n]$ individua la lunghezza della massima sottosequenza comune di P e T e può essere calcolato con complessità $\Theta(mn)$. L'algoritmo può essere facilmente modificato in modo da ricavare, oltre alla lunghezza della massima sottosequenza comune, anche la sottosequenza stessa.

Capitolo 15

Ricerca locale

Talvolta, anche una strategia estremamente "miope" può dare buoni risultati:

"Coltiviamo senza guardare più in là del proprio orticello".

In altri termini, se si conosce una soluzione ammissibile di un problema di ottimizzazione, si può cercare di trovarne una migliore che sia in qualche modo "vicina" a quella precedente. Data una soluzione ammissibile SOL del problema, si definisce un intorno I(SOL) di questa soluzione come un insieme di soluzioni ammissibili che sono ottenibili da essa in base a certi criteri di trasformazione. Si può successivamente sostituire la soluzione con una che fa parte del suo intorno ed è "migliore" di essa, e ripetere il procedimento finché non si riesce più a migliorare. Tale strategia è detta ricerca locale e gli algoritmi basati su di essa hanno la seguente struttura generale:

```
procedure RICERCALOCALE;
  begin
    SOL := una soluzione ammissibile del problema;
    while esiste S ∈ I(SOL) che è migliore di SOL do
      SOL := S;
  end;
```

Esempio 15.1 (Intorno). Si consideri il problema MINIMO ALBERO DI COPERTURA, introdotto nel § 13. Una soluzione ammissibile SOL del problema è un albero di copertura, cioè uno avente tutti i nodi del grafo, ma solo un sottoinsieme dei suoi archi. Un possibile intorno è il seguente:

I(SOL) = {T : T è un albero di copertura ottenuto da SOL aggiungendo un arco non in SOL e cancellando un altro arco dall'unico circuito che si è formato}.

Un criterio di miglioramento consiste nello scegliere un albero $T \in I$(SOL) in cui l'arco aggiunto ha un peso più piccolo dell'arco cancellato. Un esempio è fornito nella Fig. 15.1. □

In generale, il procedimento di ricerca locale si arresta su una soluzione che è un ottimo locale, cioè che non può essere migliorata attraverso le trasformazioni possibili, ma che può non essere l'ottimo globale del problema. Inoltre, partendo da soluzioni iniziali diverse si possono raggiungere ottimi locali diversi, come illustrato nella Fig. 15.2. Ovviamente, se I(SOL) comprende tutte le soluzioni ammissibili del problema, allora ogni ottimo locale è anche ottimo globale, e la ricerca locale dà

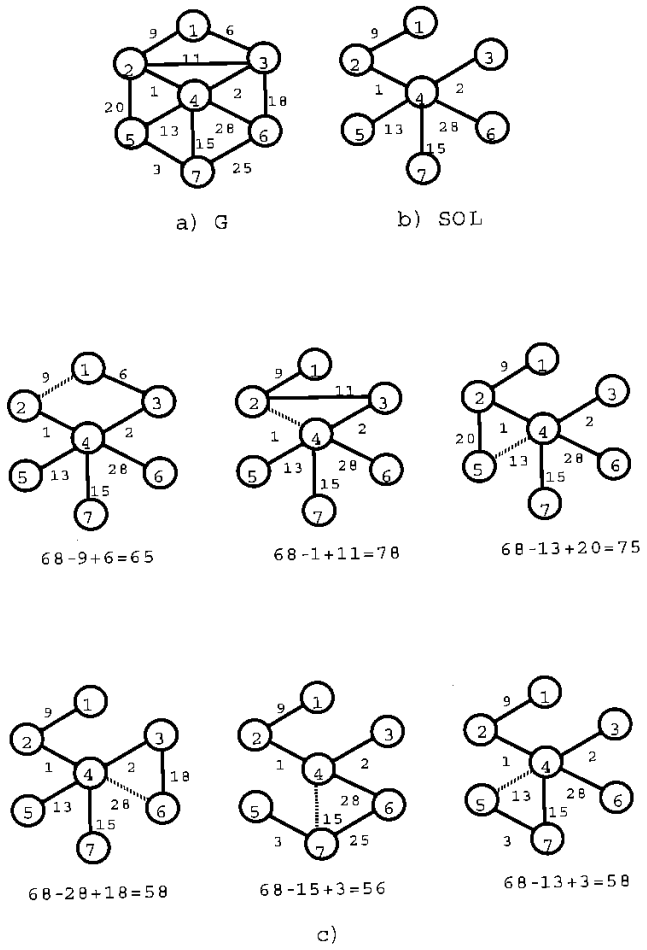


Figura 15.1: Albero di copertura. a) il grafo G della Fig. 13.1; b) una soluzione ammissibile SOL; c) sei delle tredici soluzioni dell'intorno I(SOL) ottenute aggiungendo un arco e togliendone uno dal circuito formatosi: la migliore si ottiene sostituendo [4,7] con [5,7].

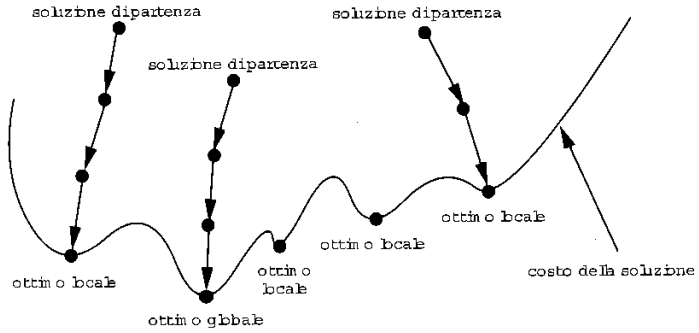


Figura 15.2: Ottimi locali e globali.

sempre la soluzione ottima del problema. Una tale ricerca, però, ha senso se la cardinalità di I(SOL) è molto più piccola della cardinalità dell'insieme di tutte le soluzioni ammissibili, e tale restrizione sulla cardinalità di I(SOL) può far arrestare la ricerca su un ottimo locale che non è quello globale!

Perché l'intero procedimento abbia complessità polinomiale, occorre che la ricerca della soluzione S nell'intorno I(SOL) abbia complessità polinomiale e che il numero complessivo di passi di "miglioramento", cioè di ripetizioni del ciclo **while**, sia anch'esso polinomiale. Per alcuni problemi, ma non per tutti, è possibile definire intorni di cardinalità "piccola" per i quali ogni ottimo locale è anche un ottimo globale che inoltre può essere trovato in tempo polinomiale con una ricerca locale.

Esempio 15.2 (Minimo albero di copertura). È possibile dimostrare che l'intorno definito nell'Esempio 15.1 per il MINIMO ALBERO DI COPERTURA è tale che ogni minimo locale è anche globale. Pertanto una procedura di ricerca locale è in grado di trovare la soluzione ottima del problema. La complessità dell'algoritmo dipende dal numero di miglioramenti. Una sequenza di miglioramenti, fino all'ottimo globale, per l'esempio di Fig. 15.1, è mostrata nella Fig. 15.3. Purtroppo, già la sola verifica che nessuna trasformazione possa essere effettuata, cioè che una soluzione non sia migliorabile, richiede $O(nm)$ tempo, poiché devono essere provati al più $m - n + 1$ archi, e ciascuno può formare un circuito lungo al più n. Pertanto, l'algoritmo di Kruskal è nettamente più veloce di questo algoritmo. □

Il più famoso algoritmo di ricerca locale è senza dubbio il "metodo del simplesso", ideato da Dantzig nel 1947 per risolvere il problema della PROGRAMMAZIONE LINEARE [cfr. § 9]. Tale algoritmo non può però essere qui riportato perché una sua descrizione dettagliata richiede buone conoscenze matematiche di ricerca operativa. In questo capitolo, consideriamo invece il progetto di altri due algoritmi di ricerca locale. Il primo algoritmo illustra un utilizzo piuttosto sorprendente della ricerca locale, per risolvere addirittura un problema che apparentemente non sembra neanche un problema di ottimizzazione: il solito problema dell'ordinamento degli elementi di un vettore. Il secondo invece ne mostra un utilizzo "classico", per risolvere un ben noto problema di ottimizzazione su grafi che è un caso particolare della programmazione lineare: il problema del flusso massimo.

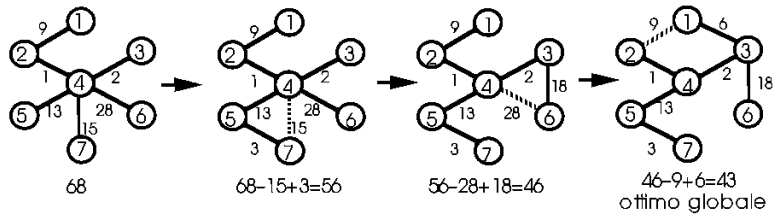


Figura 15.3: Sequenza di miglioramenti ottenuta scegliendo ad ogni passo della ricerca locale la soluzione dell'intorno che dà il massimo miglioramento.

15.1 Shellsort

In una sequenza di n elementi $a_1, \ldots, a_n$, una inversione è data da una coppia di elementi a_i e a_j tali che $i < j$ e $a_i > a_j$ [cfr. Esercizio 11.4]. Il problema dell'ordinamento degli n elementi può essere formulato come problema di ottimizzazione nel modo seguente:

ORDINAMENTO (SORTING): "Data la sequenza $a_1, \ldots, a_n$, trovare una permutazione degli n elementi che minimizzi il numero totale di inversioni".

Esempio 15.3 (Inversioni). La sequenza 1, 7, 3, 4 ha due inversioni, perché $a_2 = 7 > a_3 = 3$ e $a_2 = 7 > a_4 = 4$. La soluzione ottima 1, 3, 4, 7 ha il numero minimo (zero) di inversioni, mentre la sequenza ordinata alla rovescia 7, 4, 3, 1 ne ha il numero massimo (6). In generale, il numero minimo di inversioni per una sequenza di n elementi è sempre zero, mentre quello massimo è $\sum_{i=1}^{n-1} n - i = (n-1)n/2$. □

Si supponga per semplicità che gli n elementi $a_1, \ldots, a_n$ siano tutti distinti, e si indichino con π e a_π, rispettivamente, una permutazione degli indici $1, \ldots, n$ e la corrispondente permutazione degli elementi. Indichiamo inoltre con $a_\pi - \{a_i\}$ la sequenza a_π dalla quale è stato tolto un generico elemento a_i. Un intorno $I(a_\pi)$ può essere definito come l'insieme di tutte le permutazioni di a_π che, a meno di un elemento a_i, hanno elementi uguali nella stessa posizione relativa:

$$I(a_\pi) = \{a'_\pi : a_\pi - \{a_i\} = a'_\pi - \{a_i\}, 1 \le i \le n\}.$$

Esempio 15.4 (Intorno). Si consideri la sequenza 1, 7, 3, 4; "scalando" l'1 a destra, si ottiene la sequenza 7, 1, 3, 4, indi 7, 3, 1, 4, e infine 7, 3, 4, 1. Queste quattro sequenze sono tra loro identiche, a meno dell'elemento 1: cancellando 1 in tutte, si ottiene infatti la medesima sequenza 7, 3, 4. Ragionando in modo analogo per i rimanenti elementi, si ricava: $I(1,7,3,4) = \{(7,1,3,4), (7,3,1,4), (7,3,4,1), (1,3,7,4), (1,3,4,7), (1,7,4,3), (3,1,7,4), (4,1,7,3), (1,4,7,3)\}$. □

Un criterio di miglioramento è chiaramente quello di diminuire il numero di inversioni, cioè passare da a_π ad $a'_\pi \in I(a_\pi)$ tale che il numero di inversioni di a'_π sia minore del numero di inversioni di a_π.

Esempio 15.5 (Miglioramento). Le sequenze di $I(1,7,3,4)$ che diminuiscono il numero di inversioni sono soltanto 1, 3, 7, 4 e 1, 3, 4, 7. □

1	2	3	4	5	6	7
7	4	2	1	8	3	5
4	7	2	1	8	3	5
2	4	7	1	8	3	5
1	2	4	7	8	3	5
1	2	4	7	8	3	5
1	2	3	4	7	8	5
1	2	3	4	5	7	8

Figura 15.4: Esecuzione della procedura INSERTIONSORT

Questo criterio di miglioramento suggerisce un semplice algoritmo di ordinamento basato sulla ricerca locale, detto "insertion sort", che viene usato da molti giocatori di carte per ordinare una mano di ramino o scopone. Si considerino le carte una per volta consecutivamente, per esempio da sinistra verso destra: ogni volta che viene considerata una nuova carta, la si inserisca nella posizione giusta rispetto alle altre carte già considerate, traslando di una posizione verso destra tutte le carte maggiori.

Assumendo, come al solito, che gli n elementi da ordinare siano interi contenuti in un vettore A, si ottiene la seguente procedura Pascal INSERTIONSORT:

```
procedure INSERTIONSORT(var A: vettore; n: integer);
    var i, j, k: integer; fuorischiera: boolean;
    begin
        for i := 2 to n do begin
            k := A[i]; j := i; fuorischiera := false;
            while (A[j − 1] > k) and (not fuorischiera) do begin
                A[j] := A[j − 1];
                j := j − 1
                if j = 1 then begin
                    fuorischiera := true;
                    j := j + 1
                end;
            end;
            if fuorischiera then A[j − 1] := k else A[j] := k;
        end
    end;
```

La procedura INSERTIONSORT considera, ad ogni iterazione, la "carta" $k = A[i]$ da inserire, per $i = 2, 3, \ldots, n$. La porzione $A[1..i]$ è scandita a ritroso con l'indice j, traslando di una posizione a destra (da $A[j-1]$ ad $A[j]$) ogni elemento maggiore di k. Al termine di ciascuna iterazione, k è inserito in $A[j]$. La variabile logica *fuorischiera* è usata per evitare di "uscire a sinistra" dal vettore A con $j = 0$ qualora k sia il più piccolo elemento di $A[1..i]$. In tal caso, se j raggiunge il valore 1, allora j è subito incrementato, così che con il successivo assegnamento k è inserito in $A[j-1] = A[1]$.

Esempio 15.6 (Insertion sort). La Fig. 15.4 mostra l'ordinamento della sequenza di elementi 7, 4, 2, 1, 8, 3, 5 con la procedura INSERTIONSORT. □

La complessità della procedura dipende dal numero di "inversioni" della sequenza iniziale. Il caso peggiore si ha quando il vettore A è ordinato alla rovescia e il numero di inversioni nella sequenza iniziale è quindi massimo. In tal caso, ad ogni iterazione del **for**, l'elemento k deve essere sempre trasferito in prima posizione, e la complessità della procedura è $O(n^2)$. Per sequenze già ordinate, però, la procedura INSERTIONSORT è $\Theta(n)$, poiché il numero di inversioni è nullo e la condizione "$A[j-1] > k$" è sempre falsa. Curiosamente, il caso ottimo di INSERTIONSORT coincide con quello pessimo di QUICKSORT, la cui complessità è invece $O(n^2)$ per sequenze già ordinate!

La "pecca" dell'insertion sort è quella di poter spostare solo elementi tra loro adiacenti, richiedendo pertanto $O(n)$ tempo per trasferire un elemento piccolo dall'estremità destra a quella sinistra della sequenza. Il suo "pregio", d'altronde, è quello di richiedere poco tempo di elaborazione se la sequenza è "quasi" ordinata. Per eliminare la "pecca" e mantenerne il "pregio", si può pensare di spostare elementi che non siano adiacenti, ma "distanti" h, con $h > 1$. È questa l'idea venuta a Shell (1959), che ha proposto una estensione dell'insertion sort che porta il suo nome: shellsort. L'idea fondamentale consiste nell'usare una sequenza di distanze decrescenti, in modo da ordinare dapprima elementi che sono tra loro molto lontani e solo successivamente considerare elementi tra loro adiacenti che, per l'effetto degli ordinamenti precedenti, dovrebbero essere ormai "quasi" ordinati.

La seguente procedura Pascal SHELLSORT si ottiene da INSERTIONSORT sostituendo "1" con "h" e "2" con "$h+1$", ed iterando per valori decrescenti di h uguali a ... 1093, 364, 121, 40, 13, 4, 1 (la scelta di questi particolari valori di h sarà chiarita in seguito):

```
procedure SHELLSORT(var A: vettore; n: integer);
    var i, j, h, k: integer; fuorischiera: boolean;
    begin
        h := 1;
        while h ≤ n do h := 3 * h + 1;
        h := h div 3;
        while h ≥ 1 do begin
            for i := h + 1 to n do begin
                k := A[i]; j := i; fuorischiera := false;
                while (A[j − h] > k) and (not fuorischiera) do begin
                    A[j] := A[j − h];
                    j := j − h;
                    if j ≤ h then begin
                        fuorischiera := true;
                        j := j + h
                    end;
                end;
                if fuorischiera then A[j − h] := k else A[j] := k;
            end;
            h := h div 3
        end
    end;
```

In pratica, la procedura riordina la sequenza di elementi con la proprietà che, scelto un elemento qualsiasi, la sottosequenza contenente un elemento ogni h è riordinata. La sequenza complessiva può essere vista come formata da h sottosequenze

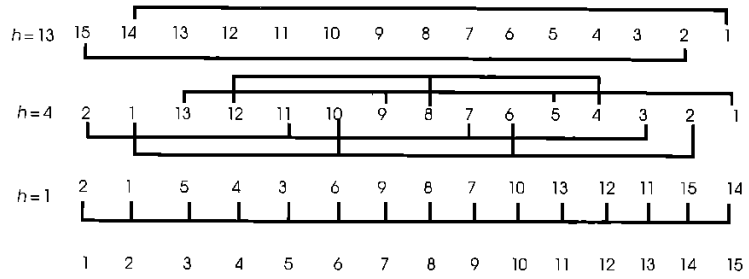


Figura 15.5: Esecuzione della procedura SHELLSORT, con evidenziate le sottosequenze che sono ordinate per $h = 13, 4, 1$.

indipendenti ma "incastrate" tra loro, ognuna contenente all'incirca n/h elementi. Ciascuna sottosequenza è ordinata con una ricerca locale basata sull'insertion sort. Per h grande, sono spostati elementi della sequenza in ingresso molto "lontani" tra loro, mentre quando h diventa 1 la procedura è proprio una INSERTIONSORT sull'intera sequenza in ingresso, in cui però ci si aspetta di spostare solo pochi elementi "vicini" tra loro.

Esempio 15.7 (Shellsort). La Fig. 15.5 mostra l'ordinamento della sequenza di 15 elementi 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 con la procedura SHELLSORT. I valori di h usati sono: 13, 4 ed 1. □

La procedura SHELLSORT è chiaramente corretta per un qualsiasi insieme di valori per h che comprenda il valore 1, perché in quest'ultimo caso essa si riduce ad INSERTIONSORT. La sua complessità non solo dipende dalla sequenza in ingresso, ma anche, e soprattutto, dai particolari valori di h usati. Un'analisi completa della complessità non può essere qui riportata, perché troppo complicata. Ci limitiamo pertanto a citare i seguenti risultati:

1. per valori di h uguali a $(3^i - 1)/2$, con $i = t, t-1, \ldots, 1$, dove t è il massimo indice s tale che $(3^{s+1} - 1)/2$ è maggiore di n (cioè proprio per i valori ... 1093, 364, 121, 40, 13, 4, 1 utilizzati precedentemente), è stato dimostrato che SHELLSORT ha complessità $O(n^{1.5})$ nel caso pessimo;
2. per valori di h uguali soltanto ad $1.72n^{1/3}$ ed 1, è stato dimostrato che SHELLSORT ha complessità $O(n^{5/3})$ nel caso medio;
3. per valori di h uguali a tutti gli interi della forma $3^i 2^j$ minori o uguali ad n, la complessità si riduce ad $O(n \log^2 n)$ nel caso pessimo;
4. esistono altri insiemi di valori di h per i quali la complessità è $O(n^{1.25})$ nel caso pessimo.

In pratica, i valori di h maggiormente utilizzati sono proprio quelli del caso (1), cioè ... 1093, 364, 121, 40, 13, 4, 1, perché generabili molto facilmente. Per questi valori, c'è chi ipotizza addirittura una complessità di $O(n \log^2 n)$ oppure $O(n^{1.25})$ nel caso medio, ma nessuno finora è stato capace di provare tale congettura. Tali considerazioni mostrano come SHELLSORT sia in pratica una procedura di ordinamento molto efficiente, da preferire addirittura ad altri algoritmi più sofisticati, specie per n non troppo elevato (p.e. $n \leq 5000$). I maggiori pregi di questa procedura, oltre

alla facilità di programmazione e alla velocità di esecuzione, consistono nel fatto che l'ordinamento avviene iterativamente, cioè senza ricorsione, ed "in loco", cioè senza bisogno di memoria ausiliaria né per copiare porzioni di A, al contrario di MERGESORT, né per gestire la ricorsione, come avviene invece sia con MERGESORT che con QUICKSORT.

15.2 Flusso massimo

Sia $G = (N, A)$ un grafo orientato pesato con due nodi particolari $s \in N$ (la sorgente) e $p \in N$ (il pozzo) e pesi c_{ij} degli archi $(i, j) \in A$ interi positivi (detti capacità). Per convenienza, si assume che $c_{ij} = 0$ se $(i, j) \notin A$ e che ogni nodo $i \in N$ stia su un cammino da s a p. Un flusso f da s a p in G è un vettore con n^2 componenti (tante quante sono tutte le coppie di nodi) che soddisfa le seguenti proprietà:

1. *Simmetria opposta*: $f_{ij} = -f_{ji}$ per ogni coppia $i, j \in N$;
2. *Vincoli di capacità*: $f_{ij} \leq c_{ij}$ per ogni coppia $i, j \in N$;
3. *Conservazione del flusso*: $\sum_j f_{ij} = 0$ per ogni nodo $i \in N - \{s, p\}$.

La quantità f_{ij}, che può essere positiva o negativa, è detta flusso netto dal nodo i al nodo j, mentre il valore v del flusso f è il flusso netto totale che esce dalla sorgente (o che entra nel pozzo), ovvero $v = \sum_j f_{sj} = \sum_i f_{ip}$. Il problema del FLUSSO MASSIMO richiede di trovare un flusso f^* che ha valore v^* più grande possibile.

Si osservi che la proprietà di simmetria opposta implica che $f_{ii} = 0$ per ogni $i \in N$. La stessa proprietà e i vincoli di capacità implicano che se sia $(i, j) \in A$ e $(j, i) \in A$, allora $f_{ij} = f_{ji} = 0$. Pertanto, un flusso netto $f_{ij} \neq 0$ implica che $(i, j) \in A$ oppure $(j, i) \in A$. Infine, la conservazione del flusso è la stessa legge della corrente elettrica di Kirchhoff che afferma che il flusso netto totale che entra in un nodo (esclusi la sorgente ed il pozzo) eguaglia il flusso netto totale che esce dallo stesso nodo.

Il problema di trovare un flusso massimo su un grafo è intimamente correlato a quello di trovare un taglio minimo sullo stesso grafo. Dato G, un taglio (S, P) è una partizione dell'insieme dei nodi tale che $s \in S$, $p \in P$, $S \cup P = N$, ed $S \cap P = \emptyset$. La capacità del taglio è $c(S, P) = \sum_{i \in S, j \in P} c_{ij}$. Un taglio di capacità minima è detto taglio minimo. Se f è un flusso ed (S, P) è un taglio, allora il flusso che attraversa il taglio è

$$f(S, P) = \sum_{i \in S, j \in P} f_{ij}.$$

Una proprietà importante è che il valore di un flusso è uguale al flusso che attraversa un taglio, il quale a sua volta non supera la capacità del taglio, cioè che $v = f(S, P) \leq c(S, P)$. Infatti, essendo $\sum_{i \in S, j \in N} f_{ij} = v$ per la proprietà di conservazione del flusso e $\sum_{i \in S, j \in S} f_{ij} = 0$ per la proprietà di simmetria opposta, si ricava $f(S, P) = \sum_{i \in S, j \in P} f_{ij} = \sum_{i \in S, j \in N} f_{ij} - \sum_{i \in S, j \in S} f_{ij} = v - 0 = v$, mentre $f(S, P) \leq c(S, P)$ deriva banalmente dai vincoli di capacità.

Un importante teorema, detto del "flusso massimo / taglio minimo", afferma che, se vale l'uguaglianza, cioè $f(S, P) = c(S, P)$, allora f è un flusso massimo ed (S, P) è un taglio minimo. Inoltre tale teorema dà lo spunto per progettare algoritmi di ricerca locale per trovare il flusso massimo.

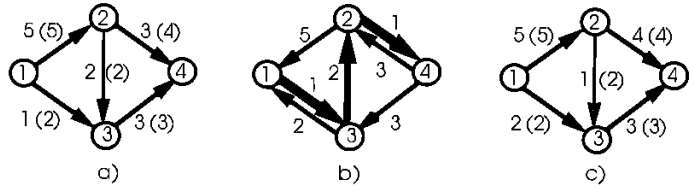


Figura 15.6: a) Un grafo G con $s = 1$, $p = 4$, ed un flusso f di valore $v = 6$, dove per ciascun arco (i, j) è indicata la coppia di valori: f_{ij} (c_{ij}). b) Il grafo residuo R ed un cammino aumentante con capacità residua $\delta = 1$. c) Il grafo G con il flusso f aggiornato aggiungendovi δ lungo gli archi del cammino aumentante.

Per dimostrare il teorema, si definisce capacità residua per un flusso f la quantità $r_{ij} = c_{ij} - f_{ij}$. Il grafo residuo $R = (N, E)$ rispetto al flusso f ha lo stesso insieme N di nodi di G, ed un arco $(i, j) \in E$ con capacità r_{ij} per ogni coppia di nodi $i, j \in N$ tale che $r_{ij} > 0$. Un cammino aumentante per f è un cammino da s a p in R. La capacità residua δ di un cammino aumentante è uguale alla capacità residua r_{ij} più piccola tra quelle di tutti gli archi (i, j) che compaiono nel cammino. È possibile definire un intorno $I(f)$ di un flusso f come l'insieme di tutti i flussi ottenibili da f variandone il valore, su ogni arco di un cammino aumentante, di una quantità non più grande della capacità residua del cammino aumentante stesso. In altri termini, dato un cammino aumentante per f, sia g il flusso di valore δ tale che $g_{ij} = \delta$ e $g_{ji} = -\delta$ se l'arco (i, j) appartiene al cammino aumentante, e $g_{ij} = 0$ altrimenti. Si può passare dalla soluzione f ad una soluzione migliore che appartiene all'intorno $I(f)$ sommando, componente per componente, il flusso g al flusso f, ottenendo così il nuovo flusso $f' = f + g$ di valore $v' = v + \delta$.

Esempio 15.8 (Grafo residuo e cammini aumentanti). La Fig. 15.6a mostra un grafo G di quattro nodi, ove $s = 1$ e $p = 4$, ed un flusso f per G di valore $v = 6$ (per chiarezza, sono indicate soltanto le componenti del flusso relative agli archi di G, seguite dalle capacità tra parentesi tonde). La Fig. 15.6.b mostra invece il corrispondente grafo residuo R; ad esempio, l'arco $(2, 4)$ di R ha capacità residua $r_{24} = c_{24} - f_{24} = 4 - 3 = 1$, mentre l'arco $(4, 2)$ ha capacità residua $r_{42} = c_{42} - f_{42} = 0 - (-3) = 3$. La Fig. 15.6b) evidenzia in grassetto anche il cammino aumentante 1, 3, 2, 4 con capacità residua $\delta = r_{13} = r_{24} = 1$, sul quale si può definire un flusso g tale che $g_{13} = g_{32} = g_{24} = \delta = 1, g_{31} = g_{23} = g_{42} = -\delta = -1$, e $g_{ij} = 0$ altrove. Sommando g ad f si ottiene il flusso mostrato nella Fig. 15.6c), di valore $v = 7$. □

È facile verificare che se f è un flusso qualsiasi per G ed f^* è un flusso massimo per G, allora $f^* - f$ è un flusso massimo per R di valore $v^* - v$. Si può quindi dimostrare il seguente teorema fondamentale:

Teorema (flusso massimo / taglio minimo). Le condizioni seguenti sono tra loro equivalenti:

i. f è un flusso massimo;
ii. non esiste alcun cammino aumentante per f;
iii. $v = c(S, P)$ per un taglio (S, P).

Dimostrazione. È immediato osservare che (i) implica (ii), poiché se esistesse un cammino aumentante allora il flusso potrebbe essere aumentato e quindi non sarebbe

massimo. Per verificare che (ii) implica (iii), siano S l'insieme dei nodi di R raggiungibili da s con un cammino e $P = N - S$. (S, P) è un taglio e poiché non esiste alcun cammino aumentante, per ogni $i \in S$ e $j \in P$ risulta che (i, j) non è un arco di R e quindi $c_{ij} = f_{ij}$. Ma ciò implica che $v = f(S, P) = c(S, P)$. Infine (iii) implica (i), poiché da $v \leq c(S, P)$ per ogni flusso f ed ogni taglio (S, P) segue che se $v = c(S, P)$ allora f è un flusso massimo. □

Esempio 15.8 (continua). Il flusso mostrato nella Fig. 15.6c) è un flusso massimo. Infatti, considerando il taglio (S, P) dove $S = \{1\}$ e $P = \{2, 3, 4\}$, si vede immediatamente che $v = 7 = f(S, P) = c(S, P)$. □

Basandosi sulla caratterizzazione matematica della soluzione fornita dal precedente teorema, sono stati proposti svariati algoritmi di ricerca locale che, partendo da un flusso f nullo (cioè con $v = 0$), lo aumentano ripetutamente lungo cammini aumentanti finché, non esistendo più alcun cammino aumentante nel grafo residuo, viene individuato il flusso massimo. L'aumento può essere fatto in base a molti criteri diversi, per esempio lungo un singolo cammino aumentante o su più cammini aumentanti contemporaneamente, preferendo quelli con massima capacità residua oppure col minimo numero di archi. Vediamo a grandi linee un algoritmo proposto da Kumar, Malhotra e Maheswari (1978) come raffinamento di altri algoritmi precedenti, che è noto come l'"algoritmo dei 3 indiani" e richiede $O(n^3)$ tempo.

15.2.1 Algoritmo dei 3 indiani

L'algoritmo dei 3 indiani si basa sulla strategia di aumentare ripetutamente lungo tutti i cammini aumentanti più corti, cioè aventi il minimo numero di archi. Il procedimento ha inizio con flusso f nullo e consta delle seguenti tre fasi, che vengono iterate:

1. si costruisce un grafo residuo R, semplificato in modo che sia formato da tutti e soli i cammini aumentanti per f comprendenti il minimo numero k archi;
2. si saturano in una sola volta tutti i cammini di R, in modo da non avere più cammini aumentanti per f con k archi, determinando un flusso saturante g di valore w per R;
3. si aggiorna f sommandoci il flusso saturante g.

Come sarà dimostrato nel seguito, le tre fasi dell'algoritmo sono iterate per $O(n)$ volte, in quanto il numero k di archi dei cammini minimi di R cresce ad ogni iterazione ed un cammino non può avere più di $n - 1$ archi. Poiché la fase (2) richiede $O(n^2)$ tempo, mentre le fasi (1) e (3) ne richiedono $O(m)$, la complessità dell'algoritmo è $O(n^3)$. La seguente procedura informale KMM descrive l'algoritmo dei 3 indiani. Le procedure RESIDUO, SATURA e AUMENTA sono usate per realizzare le tre fasi, rispettivamente, mentre la procedura AZZERA inizializza f al flusso nullo di valore $v = 0$. Per semplicità, si assume che i flussi f e g siano rappresentati con due matrici.

```
procedure KMM(var G: grafo; n : integer);
    var f, g: flusso; v, w: integer; R: grafo;
    begin
        AZZERA(G, f, v);
        w := maxint;
        while w > 0 do begin
            RESIDUO(G, R, f, v);
            SATURA(R, g, w);
            AUMENTA(G, f, v, g, w);
        end;
    end;
```


La procedura AZZERA utilizza una visita BFS per inizializzare a 0 tutte le componenti f_{ij} ed f_{ji} del flusso f tali che $(i, j) \in A$, e richiede $O(m)$ tempo.

La procedura RESIDUO costruisce in $O(m)$ tempo il grafo residuo $R = (N, E)$ con una BFS modificata, dove un arco $(i, j) \in E$ con capacità $r_{ij} = c_{ij} - f_{ij}$ per ogni arco $(i, j) \in A$ o $(j, i) \in A$ tale che $r_{ij} > 0$. Si assume che G ed R siano rappresentati con gli insiemi di adiacenza sia degli archi uscenti che degli archi entranti di ciascun nodo. Durante la visita, è assegnata ad ogni nodo i la sua "distanza minima" d_i dalla sorgente s, intesa come il minimo numero di archi in un cammino tra s e i. Il grafo R è semplificato, con un'altra eventuale passata e sempre in $O(m)$ tempo, per eliminare tutti quei nodi e quegli archi che non risiedono sui cammini aumentanti più corti da s a p. Infatti, si possono eliminare tutti i nodi j diversi da p con $d_j \geq d_p$, insieme a tutti gli archi incidenti in essi. Inoltre, poiché un cammino più corto può contenere solo archi da un nodo i a distanza d_i ad un altro nodo j a distanza $d_j = d_i + 1$, si può eliminare ogni arco (i, j) tale che $d_j \leq d_i$. Il grafo R così semplificato è aciclico e "stratificato", cioè ogni arco (i, j) unisce due nodi tali che $d_j = d_i + 1$, ed ogni nodo $i \in N - \{s, p\}$ ha distanza d_i tale che $0 = d_s < d_i < d_p$. Inoltre, R può essere ulteriormente semplificato con un'altra passata, che parte dal pozzo p ed esamina "a ritroso" gli archi, marcando così i nodi dai quali si può raggiungere p, e cancellando successivamente tutti i nodi non marcati, insieme agli archi incidenti. Dopo quest'ultima semplificazione, ogni nodo di R sta su un cammino minimo tra s e p, se esiste un tale cammino. In altri termini, R contiene tutti e soli i cammini aumentanti per f di lunghezza minima $k = d_p$.

Una volta costruito il grafo residuo semplificato R in $O(m)$ tempo, la procedura SATURA cerca un flusso saturante per esso, cioè un flusso g che saturi tutti i cammini presenti in R. Un arco è saturo se il flusso che lo attraversa è pari alla capacità dell'arco, ed un cammino di R è saturo se contiene almeno un arco saturo. Un flusso saturante per R non è detto che sia massimo per R (si potrebbe infatti aumentare percorrendo qualche arco nel verso opposto all'orientamento), ma garantisce comunque che in futuro non potranno più essere considerati cammini aumentanti di lunghezza $k = d_p$. Infatti, sia R' il grafo residuo all'iterazione successiva, con distanze d'. Ciascun arco (i, j) di R ha $d_i = d_j + 1$, mentre ciascun arco di R' o è un arco di R o è un arco di R con orientamento opposto. Ma allora ogni arco in R' ha $d_i \leq d_j + 1$, e quindi $d'_p \geq d_p$. Se per assurdo fosse $d'_p = d_p$, allora esisterebbe un cammino minimo da s a p in R che non è stato saturato tale che ogni arco (i, j) ha $d_i = d_j + 1$. Ma ciò contraddice il fatto che almeno un arco di ogni cammino in R è saturato e quindi non compare in R'. Pertanto, $d'_p > d_p$ e in futuro non potranno più essere considerati cammini aumentanti di lunghezza $k \leq d_p$.

Esempio 15.9 (Flusso saturante). La Fig. 15.7a) mostra un grafo G con sorgente $s = 1$, pozzo $p = 6$, e capacità degli archi indicate tra parentesi. Alla prima iterazione il flusso f è nullo e il grafo residuo semplificato R, che contiene tutti i cammini aumentanti di lunghezza minima (uguale a 3), è mostrato nella Fig. 15.7b), insieme ad un flusso saturante g per R. Le componenti del flusso g sono indicate senza parentesi, seguite dalle capacità residue di R tra parentesi. Alla seconda iterazione, il flusso f è uguale al flusso saturante g trovato alla prima iterazione, ed il grafo residuo

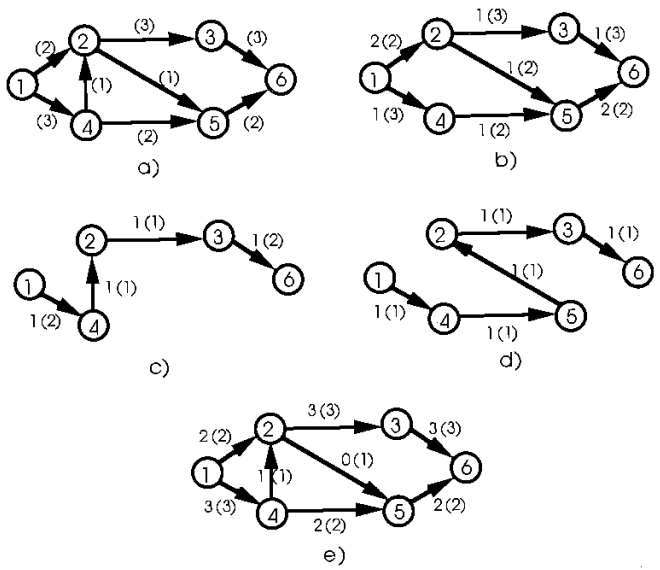


Figura 15.7: a) Un grafo G, con capacità degli archi indicate tra parentesi. b) Il grafo residuo R semplificato alla prima iterazione, con un flusso saturante g e le capacità residue tra parentesi. c)-d) R alla seconda e terza iterazione, con un flusso saturante g. e) Il flusso massimo f per G.

semplificato R (che contiene cammini lunghi 4) è mostrato nella Fig. 15.7c), insieme a un flusso saturante. Il grafo R alla terza iterazione (con cammini lunghi 5) ed un flusso saturante sono illustrati nella Fig. 15.7d). Infine, la Fig. 15.7e) mostra il grafo G con il flusso massimo f così trovato. Un taglio minimo si può individuare dal grafo residuo R per f ponendo in S la sorgente s e tutti i nodi raggiungibili da s con archi di R (nessuno nell'esempio di Fig. 15.7e), perché f satura entrambi gli archi uscenti da 1 in G), ottenendo così $S = \{1\}$ e $P = \{2, 3, 4, 5, 6\}$. □

Per trovare un flusso saturante g per R, si definisce per ogni nodo i di R la sua "portata", ovvero l'ammontare massimo di flusso che può attraversare il nodo i, come il minimo tra la somma delle capacità residue degli archi entranti in i e quella degli archi uscenti da i:

$$portata_i = min\{\sum_j r_{ji}, \sum_j r_{ij}\}$$

dove, per applicare la definizione anche ad s ed a p, si può assumere di avere un arco "fittizio di ritorno" (p, s) con capacità infinita. Il vettore *portata* può essere calcolato in $O(m)$ tempo. Selezionando poi in $O(n)$ tempo il nodo i con portata minima, si è sicuri di poter trasmettere $portata_i$ unità di flusso da s a p in R con una opportuna procedura INSTRADA. Questa procedura, partendo dal nodo i, modifica il flusso g, aggiungendovi $portata_i$ unità di flusso che sono instradate da i verso p ed a ritroso da i

verso s, ed aggiorna le portate dei nodi. Effettuato l'instradamento, con una semplice procedura AGGIORNA sono eliminati i nodi diversi da s con portata nulla, tra i quali c'è almeno i, e gli archi incidenti in essi. Se il flusso g non è saturante, cioè la portata della sorgente è positiva, allora si instradano altre unità di flusso che sono sommate al flusso g. Il procedimento è iterato finché il flusso g non è saturante.

```
procedure SATURA(var R: grafo; var g: flusso; var w: integer);
    var portata: vettore;
    begin
        AZZERA(R, g, w);
        for each i ∈ N do
            portata_i := min { Σ_j r_ji, Σ_j r_ij };
        while portata_s > 0 do begin
            seleziona il nodo i con portata_i minima;
            INSTRADA(R, i, portata, g);
            w := w + portata_i;
            AGGIORNA(R, portata, g)
        end
    end;
```

La complessità della procedura SATURA è determinata da quella del ciclo **while**, che a sua volta dipende da una accorta realizzazione della procedura INSTRADA. Questa procedura aggiunge $x = portata_i$ unità di flusso a g, instradandole da i verso p ed a ritroso da i verso s, secondo una visita BFS. Ad ogni chiamata, esamina ciascun nodo j di R al più una volta allontanandosi progressivamente dal nodo i nel grafo stratificato. In questo modo, quando è considerato il nodo j, il flusso sugli archi entranti in j (se j sta in un cammino tra i e p) oppure su quelli uscenti da j (se j sta tra s ed i) è già stato tutto instradato. Per semplicità, nella procedura INSTRADA non è mostrato tutto l'instradamento, ma solo quello da i verso p poiché l'altro è perfettamente analogo. Durante la visita, INSTRADA calcola e aggiorna la quantità $esci_j$, che denota quante unità di flusso devono uscire dal nodo j, ed aggiorna le portate dei nodi.

```
procedure INSTRADA(var R: grafo; i: nodo; var portata: vettore; var g: flusso);
    var Q: coda; M: integer; esci: vettore;
    begin
        {instrada x = portata_i unità di flusso da i verso p, modificando g;
        l'instradamento a ritroso da i verso s è speculare e non è mostrato};
        CREACODA(Q); INCODA(i, Q);
        for each j ∈ N − {i} do esci_j := 0;
        esci_i := x; portata_i := 0;
        while not CODAVUOTA(Q) do begin
            j := LEGGICODA(Q); FUORICODA(Q);
            while esci_j > 0 do begin
                considera un arco (j, u) uscente da R;
                M := min{r_ju, esci_j};
                r_ju := r_ju − M;
                if r_ju = 0 then cancella (j, u) da R;
                esci_j := esci_j − M;
                esci_u := esci_u + M;
                g_ju := g_ju + M;
```

```
            g_uj := -g_ju;
            portata_u := portata_u - M;
            INCODA(u, Q);
        end;
    end;
end;
```

Il tempo richiesto della procedura SATURA dipende dal numero di ripetizioni del ciclo **while**, ove è richiamata la INSTRADA. Poiché ad ogni iterazione è eliminato almeno un nodo saturo (con portata divenuta nulla), il numero di iterazioni è $O(n)$. Ogni arco di R può essere cancellato da AGGIORNA oppure saturato e cancellato da INSTRADA, ma una sola volta, oppure può essere riempito con un po' di flusso ma non saturato, ma solo una volta per ciascuna iterazione e per non più di un arco per nodo. Inoltre, non sono mai esaminati tutti gli archi uscenti da j, ma soltanto i primi y_j, dei quali i primi $y_j - 1$ sono saturati e cancellati, mentre l'y_j-esimo è eventualmente riempito ma non saturato.

Pertanto, le saturazioni e cancellazioni di archi sono in totale $O(m)$, mentre i riempimenti senza saturazione sono in totale $O(n^2)$, ed il tempo richiesto da SATURA è così $O(n^2)$. La procedura AUMENTA aggiorna il flusso f sommandoci il flusso g mediante una visita BFS e richiede $O(m)$ tempo, come la procedura RESIDUO. Ne segue che ciascuna iterazione della procedura KMM prende $O(n^2)$ tempo, dovuto alla SATURA. Poiché KMM effettua al più $n - 1$ iterazioni in ciascuna delle quali cerca un flusso saturante, la complessità dell'algoritmo dei 3 indiani per trovare il flusso massimo risulta essere $O(n^3)$.

15.3 Esercizi

Esercizio 15.1 (Bubblesort). Si progetti un algoritmo per ordinare un vettore di n elementi che abbia complessità $O(n^2)$ e utilizzi scambi tra elementi adiacenti, basandosi sullo stesso intorno di INSERTIONSORT, ma cercando ad ogni iterazione di sistemare l'elemento di valore massimo nella posizione che gli compete nell'ordinamento.

Esercizio 15.2 (Scheduling). Dati n programmi con tempi di esecuzione $t_1, \ldots, t_n$, si vuole eseguirli su un processore in modo da minimizzarne il tempo medio di completamento $(1/n)\sum_i c_i$, dove il tempo di completamento c_i del programma i è la somma del suo tempo di esecuzione e dei tempi di esecuzione di tutti i programmi che lo precedono nell'ordinamento (schedule) finale. Si progetti un algoritmo di ricerca locale per risolvere il problema.

Esercizio 15.3 (Ciclo Euleriano). Dato un grafo orientato $G = (N, A)$, si vuole trovare un ciclo Euleriano, cioè un cammino chiuso (ma non semplice) che include ciascun arco una e una sola volta. Si determini una condizione necessaria e sufficiente affiché G ammetta un ciclo Euleriano, e si progetti un algoritmo di ricerca locale per trovarlo.

Il famoso Teorema di Eulero assicura che G contiene un ciclo Euleriano se e solo se è connesso (trascurando l'orientamento degli archi), e per ciascun nodo il numero di archi entranti nel nodo è uguale al numero di archi uscenti dal nodo. Si assuma quindi che G verifichi tali condizioni.

Una partizione di Eulero P per G è una partizione di A in cammini chiusi (non necessariamente semplici) disgiunti sugli archi. Una partizione di Eulero può essere

determinata specificando per ogni arco $a = (i, j)$ un unico successore $succ(a) = b = (j, k)$ in modo che non esistano mai due archi con lo stesso successore. Ovviamente, la cardinalità di una partizione di Eulero è compresa tra 1 ed $\lfloor n/2 \rfloor$ ed un ciclo Euleriano corrisponde ad una partizione di Eulero di minima cardinalità. Una partizione iniziale P è facilissima da trovare, per esempio considerando un nodo alla volta di G e definendo l'h-esimo arco uscente dal nodo come il successore dell'h-esimo arco entrante. Dati due archi a e b, con $succ(a) = c$ e $succ(b) = d$, si può definire un'operazione SCAMBIO(a, b) che scambia tra loro i successori in modo che $succ(a) = d$ e $succ(b) = c$. Si può allora definire l'intorno

$$I(P) = \{P' : P' \text{ è ottenuta da } P \text{ con una operazione SCAMBIO}\}.$$

Ovviamente, uno scambio può far aumentare o diminuire di uno la cardinalità di una partizione, a seconda che divida un cammino chiuso in due cammini disgiunti sugli archi, oppure che fonda due cammini chiusi in uno solo. L'algoritmo di ricerca locale sceglierà ad ogni iterazione un'operazione di scambio che decrementi la cardinalità della partizione e si fermerà quando non saranno più possibili miglioramenti, cioè quando sarà stato individuato un ciclo Euleriano.

Esercizio 15.4 (Riempimento di matrice). Dati $2n$ interi non negativi $c_1, c_2, \ldots, c_n$, $r_1, r_2, \ldots, r_n$, si vuole determinare una matrice $n \times n$ con elementi interi non negativi tali che la somma degli elementi della colonna i-esima sia uguale a c_i e la somma degli elementi della riga i-esima sia uguale ad r_i. Quali condizioni devono valere su $c_1, \ldots, c_n, r_1, \ldots, r_n$ affinché esista una soluzione? Si formuli il problema come un problema di flusso massimo. Si fornisca poi un algoritmo di complessità $O(n^2)$ senza usare il flusso massimo.

Esercizio 15.5 (Somma di matrici). Siano dati una matrice $m \times n$ M con elementi non negativi e 4 interi non negativi l_1, l_2, t_1, t_2. Si vogliono determinare due matrici $m \times n$ M_1 ed M_2 con elementi interi non negativi tali che $M = M_1 + M_2$, la somma degli elementi di ogni linea (cioé riga o colonna) di M_i non superi l_i, e la somma di tutti gli elementi di M_i non superi t_i, per $i = 1, 2$. Si formuli il problema come un problema di flusso massimo.

Esercizio 15.6 (Formulazioni flusso massimo). Si dimostri che le seguenti varianti del problema del flusso massimo possono essere ricondotte alla formulazione originale del problema descritta nel presente capitolo:

1. Il grafo ha più di una sorgente e più di un pozzo;
2. I nodi, oltre agli archi, hanno capacità;
3. Il grafo non è orientato.

Esercizio 15.7 (Caso pessimo algoritmo 3 indiani). Si costruisca un grafo G, per una infinità di valori di n, per il quale l'algoritmo dei 3 indiani ha complessità $\Theta(n^3)$.

Esercizio 15.8 (Intorno per abbinamento massimo). Dato un grafo non orientato, un abbinamento (o "matching") M è un sottoinsieme di archi tali che non esistono due archi incidenti nel medesimo nodo. Si definisca un intorno per il problema di trovare un abbinamento di massima cardinalità.

Sia dato un abbinamento qualsiasi M. Si definisca un nodo "scoperto" se non ha alcun arco di M incidente in esso. Si definisca "catena alternante" una catena di archi costituita, alternatamente, da un arco non appartenente ad M e da uno appartenente ad M. Si definisca "catena aumentante" una catena alternante tra due nodi scoperti.

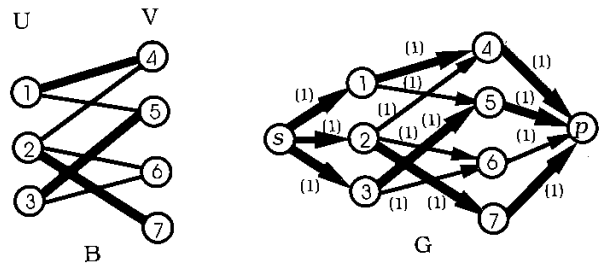


Figura 15.8: Un grafo non orientato bipartito B ed il corrispondente grafo orientato G con capacità unitarie (tra parentesi). Gli archi in grassetto evidenziano un abbinamento massimo (di cardinalità 3) di B e le corrispondenti componenti unitarie del flusso massimo di G.

In una catena aumentante, il numero di archi che non fanno parte dell'accoppiamento è uno più degli archi che ne fanno parte. Pertanto, si può definire un intorno

$$I(M) = \{M' : M' \text{ è ottenibile da } M \text{ scambiando in una catena aumentante gli archi che fanno parte dell'accoppiamento con gli altri}\}$$

(Trovare una catena aumentante richiede tempo polinomiale, ma l'algoritmo non è affatto banale).

Esercizio 15.9 (Abbinamento bipartito). Dato un grafo non orientato bipartito , si vuole trovare un abbinamento di massima cardinalità [cfr. Esercizio 15.8]. Si dimostri che il problema è risolubile con l'algoritmo dei 3 indiani con complessità $O(m\sqrt{n})$.

Sia $B = (N, A)$ un grafo non orientato bipartito. Per definizione, esiste una partizione dell'insieme N dei nodi in due insiemi disgiunti U e V tale che per ogni arco $[i, j]$ risulta $i \in U$ e $j \in V$ [cfr. Esercizi 9.4 e 9.5]. Si consideri un grafo orientato G con capacità unitarie sugli archi, ottenuto da B orientando gli archi da U verso V, aggiungendo un nodo sorgente s collegato con un arco ad ogni nodo di U ed un nodo pozzo p collegato con un arco da ogni nodo di V. Formalmente, si costruisce $G = (N', A')$ tale che:

$$\begin{aligned} N' &= N \cup \{s,p\}, \\ A' &= \{(i,j) : [i,j] \in A, i \in U, j \in V\} \cup \{(s,i) : i \in U\} \cup \{(j,p) : j \in V\}, \\ c_{ij} &= 1 \text{ per ogni } (i,j) \in A'. \end{aligned}$$

Come illustrato nella Fig. 15.8, è facile verificare che un flusso massimo f^* di valore v^* in G corrisponde ad un abbinamento M di cardinalità v^* in B, dove gli archi $[i, j] \in M$ sono dati dagli archi (i, j) con $i \in U, j \in V$ ed $f^*_{ij} = 1$.

Per dimostrare che l'algoritmo dei 3 indiani ha complessità $O(m\sqrt{n})$ sul grafo G così definito, mostriamo che la procedura SATURA richiede $O(m)$ tempo e che KMM effettua $O(\sqrt{n})$ iterazioni in ciascuna delle quali cerca un flusso saturante.

Poiché ogni arco del grafo G ha capacità unitarie, anche il grafo residuo R ha capacità unitarie. Inoltre, dato che il flusso ha componenti intere, non ci sono mai riempimenti senza saturazioni, ed ogni arco di R o è saturato e cancellato oppure non è neanche esaminato. Pertanto, il tempo richiesto da SATURA è $O(m)$. Vediamo che il numero di volte che è cercato un flusso saturante è al più $2\lceil\sqrt{n-2}\rceil$. Siano f il flusso ad una generica iterazione, f^* il flusso massimo, ed R il grafo residuo per f. Poiché $f^* - f$ è un flusso per R ed ogni componente del flusso vale 0 o 1 su ogni arco, è possibile partizionare gli archi di R per i quali il flusso è 1 in un insieme di $v^* - v$ cammini tra s e p (più qualche eventuale ciclo). Poiché le capacità residue sono unitarie ed ogni nodo di R ha al più un arco entrante oppure al più un arco uscente, ogni nodo (tranne s e p) appare in al più uno di tali cammini, e pertanto f ha un cammino aumentante di lunghezza $k \leq (n-2)/(v^* - v) + 1$. Ma dopo $2\lceil\sqrt{n-2}\rceil$ iterazioni, $k \geq \sqrt{n-2} + 1$, perché ad ogni iterazione la lunghezza k dei cammini aumentanti più corti aumenta. Combinando quest'ultime due relazioni, si ricava che $v^* - v \leq \sqrt{n-2}$ e quindi dopo al più altre $\lfloor\sqrt{n-2}\rfloor$ iterazioni viene trovato il flusso massimo. Pertanto, il numero di iterazioni effettuate da KMM è $O(\sqrt{n})$.

Capitolo 16

Analisi di algoritmi

Analizzare la complessità di un algoritmo non sempre è un compito agevole. Le maggiori difficoltà si incontrano generalmente nell'analisi di algoritmi ricorsivi, nell'analisi del caso medio, e nell'analisi "ammortizzata" di una sequenza di operazioni. Fortunatamente, sono state individuate alcune tecniche che risultano di grandissima utilità nell'analisi di algoritmi.

Per analizzare la complessità di un algoritmo ricorsivo, occorre impostare relazioni di ricorrenza e individuarne l'ordine di grandezza. Ci sono tre metodi diversi per risolvere tali relazioni:

1. espandere le relazioni stesse, sostituendo successivamente nella relazione di partenza scritta per $T(n)$ quella riscritta per $T(n-1)$, poi quella per $T(n\text{-}2)$, e così via finché, chiarita la struttura della somma così ottenuta, non resta che una formula "chiusa" per $T(n)$ che comprende termini in n e costanti;
2. usare la soluzione generale di alcune relazioni di ricorrenza "comuni" che sono espresse in forma parametrica, e ricavare la soluzione di $T(n)$ semplicemente sostituendo al posto dei parametri i valori costanti che appaiono nella relazione per $T(n)$;
3. "tentare" con una soluzione $f(n)$, lasciando qualche parametro non specificato, e dimostrare, generalmente per induzione su n, che $T(n)$ è $O(f(n))$ oppure $\Omega(f(n))$.

In mancanza di metodi più generali, finora è stata applicata la tecnica (1), che in linea di principio garantisce sempre di trovare una soluzione, ma che in pratica può richiedere di risolvere o maggiorare sommatorie molto complicate. In questo capitolo, verranno trattate le altre due tecniche. In particolare, si vedranno soluzioni generali per due tipi di ricorrenze lineari a coefficienti costanti molto "comuni": quelle di ordine costante e quelle con partizione bilanciata dei dati.

Per l'analisi del caso medio, purtroppo, sono note poche tecniche generali e le formule che vengono impostate per analizzare la complessità sono in genere di difficilissima risoluzione. In pratica, le analisi trattabili si riducono ai due casi già visti per il QUICKSORT e le tabelle hash, e poco più. Comunque, la tecnica (3) può essere usata per semplificare notevolmente l'analisi nel caso medio. In questo capitolo verrà presentato un esempio di analisi per tentativi per valutare la complessità nel caso medio di un algoritmo ricorsivo che risolve efficientemente il problema della SELEZIONE, dove si vuole individuare il k-esimo elemento più piccolo di un vettore.

Infine, per l'analisi ammortizzata di una sequenza di operazioni su una struttura di dati, è molto usata la cosiddetta "tecnica del potenziale", che sarà anch'essa trattata in questo capitolo. Inoltre, sarà mostrato un utilizzo diverso della medesima tecnica per

analizzare algoritmi "on line" ("in linea"), che elaborano cioè i dati di una sequenza di ingresso uno alla volta, non appena sono disponibili, senza attendere di averli raccolti tutti.

16.1 Relazioni di ricorrenza lineari a coefficienti costanti

In molti algoritmi ricorsivi, accade spesso che l'algoritmo richiami se stesso un numero costante a_i di volte su dati di dimensione $n - i$, con i costante. Le chiamate ricorsive possono essere effettuate prima o dopo aver eseguito un numero polinomiale di operazioni. In tal caso, la funzione di complessità $T(n)$ dell'algoritmo può essere espressa in termini di $a_iT(n-i)+cn^b$. I parametri a_i e i sono costanti intere tali che $a_i \geq 1$, perché viene fatta almeno una chiamata ricorsiva, e $i \geq 1$, perché la dimensione del problema deve essere diminuita di almeno uno. Invece, i parametri c e β sono costanti reali tali che $c > 0$ e $\beta \geq 0$, in quanto è eseguita almeno una operazione oltre alle chiamate ricorsive.

Generalizzando, date $a_1, a_2, \ldots, a_h$ costanti intere non negative, con h costante positiva, si considerano relazioni di ricorrenza nelle quali $T(n)$ dipende da $\sum_{1\leq i\leq h} a_iT(n-i) + cn^\beta$, con c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Una relazione di ricorrenza di questo tipo è detta lineare (perché n compare nei $T(n-i)$ con grado 1), a coefficienti costanti (essendo $a_1, a_2, \ldots, a_h$ costanti), di ordine costante (poiché anche h è una costante), e con lavoro polinomiale di suddivisione e/o ricombinazione (in quanto cn^β è un polinomio). Si ottiene il seguente risultato.

Teorema (delle ricorrenze lineari di ordine costante): Siano $a_1, a_2, \ldots, a_h$ costanti intere non negative, con h costante positiva, c e β costanti reali tali che $c > 0$ e $\beta \geq 0$, e sia $T(n)$ definita dalla relazione di ricorrenza:

$$T(n) = \text{costante}, \qquad \text{per } n \leq m \leq h,$$
$$T(n) = \sum_{1\leq i\leq h} a_iT(n-i) + cn^\beta, \qquad \text{per } n > m.$$

Posto $a = \sum_{1\leq i\leq h} a_i$, allora:

i. $T(n)$ è $O(n^{\beta+1})$, se $a = 1$,
ii. $T(n)$ è $O(a^n n^\beta)$, se $a \geq 2$.

Dimostrazione. Si supponga dapprima che ci sia un solo coefficiente non nullo, ovvero esista j, $1 \leq j \leq h$, tale che $a_j = a$, mentre $a_i = 0$, per $1 \leq i \leq h$ ed $i \neq j$. Si assuma per semplicità che n sia uguale ad $m + kj$, per un k intero. Sostituendo

successivamente si ottiene:

$$\begin{aligned}
T(n) &= aT(n-j) + cn^\beta = \\
&= a(aT(n-2j) + c(n-j)^\beta) + cn^\beta = \\
&\vdots \\
&= a(a \ldots a(a(T(m) + c(m+j)^\beta) + c(m+2j)^\beta) + \cdots + \\
&\quad + c(n-j)^\beta) + cn^\beta = \\
&= a^k T(m) + \sum_{0 \le \nu \le k-1} a^\nu c(n-\nu j)^\beta \le \\
&\le a^k T(m) + cn^\beta \sum_{0 \le \nu \le k-1} a^\nu .
\end{aligned}$$

Si individuano due casi possibili:

I. Se $a = 1$, allora:

$$\begin{aligned}
T(n) &\le T(m) + cn^\beta[1 + 1 + \cdots + 1] = \\
&= T(m) + cn^\beta k = \\
&= T(m) + cn^\beta (n-m)/j,
\end{aligned}$$

e quindi $T(n)$ è $O(n^{\beta+1})$.

II. Se $a \ge 2$, allora:

$$\begin{aligned}
T(n) &\le a^k T(m) + cn^\beta[(a^k - 1)/(a-1)] \le \\
&\le a^k[T(m) + cn^\beta] = \\
&= a^{(n-m)/j}[T(m) + cn^\beta],
\end{aligned}$$

e quindi $T(n)$ è $O(a^n n^\beta)$.

Per completare la dimostrazione, si consideri il caso più generale in cui $\prod_{i=1}^h a_i \neq 0$, cioè tutti gli h coefficienti sono non nulli, con $h > 1$. In tal caso, risulta necessariamente $\sum_{i=1}^h a_i = a \ge 2$. Pertanto, si ha:

$$\begin{aligned}
T(n) &= \sum_{1 \le i \le h} a_i T(n-i) + cn^\beta \le \\
&\le \sum_{1 \le i \le h} a_i T(n-1) + cn^\beta = \\
&= aT(n-1) + cn^\beta,
\end{aligned}$$

e ci si riconduce al caso precedente con $j = h = 1$ e $a = a_j \ge 2$. Infine, la dimostrazione è analoga anche per $n \neq m + kj$. □

Esempio 16.1 (Quicksort). Per la procedura QUICKSORT [cfr. § 10] le relazioni di ricorrenza, nel caso pessimo, sono del tipo: $T(n) \le T(n-1) + cn$. Poiché $a = 1$ e $\beta = 1$, si ha che $T(n)$ è $O(n^2)$. □

Esempio 16.2 (MIN ricorsiva). Per la versione ricorsiva della procedura MIN introdotta nell'Introduzione, le relazioni di ricorrenza sono del tipo: $T(n) \le T(n-1) + c$. Essendo $a = 1$ e $\beta = 0$, risulta che $T(n)$ è $O(n)$. □

Esempio 16.3 (Torri di Hanoi). Per la procedura TORRIDIHANOI [cfr. § 3] si ottengono relazioni del tipo: $T(n) \le 2T(n-1) + c$. Poiché $a = 2$ e $\beta = 0$, ne segue che $T(n)$ è $O(2^n)$. □

Per algoritmi di tipo "divide et impera" in cui si partizionino i dati in modo bilanciato, il problema originario di dimensione n è diviso in a sottoproblemi di dimensione n/b ciascuno. Se si dividono i dati e/o si ricombinano i risultati in tempo polinomiale, la funzione di complessità $T(n)$ può essere espressa in termini di $aT(n/b) + cn^\beta$. I parametri a e b sono costanti intere tali che $a \ge 1$, perché è fatta almeno una chiamata, e $b \ge 2$, in quanto il problema è suddiviso in almeno due sottoproblemi, mentre c e β sono, come prima, costanti reali tali che $c > 0$ e $\beta \ge 0$. Una relazione di ricorrenza di questo tipo è ancora lineare, a coefficienti costanti, e con lavoro polinomiale di suddivisione e ricombinazione, ma non è più di ordine costante, bensì con partizione bilanciata (perché n è diviso per una costante b). In questo caso, si ottiene il seguente risultato.

Teorema (delle ricorrenze lineari con partizione bilanciata): Siano a e b costanti intere tali che $a \ge 1$ e $b \ge 2$, e c, d e β costanti reali tali che $c > 0$, $d \ge 0$, e $\beta \ge 0$. Sia $T(n)$ data dalla relazione di ricorrenza:

$$\begin{aligned}
T(n) &= d, && \text{per } n = 1, \\
T(n) &= aT(n/b) + cn^\beta, && \text{per } n > 1.
\end{aligned}$$

Posto $\alpha = \log a / \log b$, allora:

i. $T(n)$ è $O(n^\alpha)$, se $\alpha > \beta$,
ii. $T(n)$ è $O(n^\alpha \log n)$, se $\alpha = \beta$,
iii. $T(n)$ è $O(n^\beta)$, se $\alpha < \beta$,

Dimostrazione. Si supponga dapprima per semplicità che n sia una potenza di b, cioè $n = b^k$ per un k intero. Sostituendo successivamente si ottiene:

$$\begin{aligned}
T(n) &= aT\left(b^{k-1}\right) + cb^{k\beta} = \\
&= a\left(aT\left(b^{k-2}\right) + cb^{(k-1)\beta}\right) + cb^{k\beta} = \\
&= a\left(a\left(aT\left(b^{k-3}\right) + cb^{(k-2)\beta}\right) + cb^{(k-1)\beta}\right) + cb^{k\beta} = \\
&\vdots \\
&= a\left(a\left(\cdots a\left(aT(1) + cb^\beta\right) + cb^{2\beta}\right) + \cdots + cb^{(k-1)\beta}\right) + cb^{k\beta} = \\
&= a^k d + ca^{k-1}b^\beta + ca^{k-2}b^{2\beta} + \cdots + cab^{(k-1)\beta} + cb^{k\beta} = \\
&= a^k d + cb^{k\beta}\left[\left(a/b^\beta\right)^{k-1} + \left(a/b^\beta\right)^{k-2} + \cdots + \left(a/b^\beta\right)^2 + a/b^\beta + 1\right].
\end{aligned}$$

Si osservi che $a^k = a^{\log n/\log b} = 2^{\log a \log n/\log b} = n^{\log a/\log b} = n^\alpha$, e che $a = b^\alpha$, poiché $\alpha = \log a/\log b$. Ponendo $q = a/b^\beta = b^\alpha/b^\beta = b^{\alpha-\beta}$, la relazione può essere scritta come:

$$T(n) = n^\alpha d + cb^{k\beta}\left[q^{k-1} + q^{k-2} + \cdots + q^2 + q + 1\right].$$

Si individuano tre casi, a seconda che α sia maggiore, uguale, o minore di β, e che quindi q risulti, rispettivamente, maggiore, uguale, o minore di 1.

I. Se $\alpha > \beta$, allora:

$$\begin{aligned} T(n) &= n^\alpha d + cb^{k\beta}[(q^k - 1)/(q-1)] \leq \\ &\leq n^\alpha d + cb^{k\beta} q^k/(q-1) = \\ &= n^\alpha d + ca^k/(q-1) = \\ &= n^\alpha[d + c/(q-1)], \end{aligned}$$

e quindi $T(n)$ è $O(n^\alpha)$;

II. Se $\alpha = \beta$, allora:

$$\begin{aligned} T(n) &= n^\alpha d + cb^{k\beta}[1 + 1 + \cdots + 1] = \\ &= n^\alpha d + cn^\beta k = \\ &= n^\alpha[d + c\log n/\log b], \end{aligned}$$

e pertanto $T(n)$ è $O(n^\alpha \log n)$;

III. Se $\alpha < \beta$, allora:

$$\begin{aligned} T(n) &= n^\alpha d + cb^{k\beta}[(1 - q^k)/(1-q)] \leq \\ &\leq n^\alpha d + cb^{k\beta}[1/(1-q)] = \\ &= n^\alpha d + cn^\beta/(1-q), \end{aligned}$$

e quindi $T(n)$ è $O(n^\beta)$.

Nel caso che n non sia una potenza di b, si può considerare un problema di dimensione più grande, p.e. n', tale che n' sia la più piccola potenza di b maggiore di n. Poiché $n' \leq bn$ e b è una costante, la dimensione aumenta solo di un fattore costante, e la soluzione rimane valida in ordine di grandezza. Inoltre, poiché si è interessati all'analisi del caso pessimo, la soluzione della relazione di ricorrenza si applica anche al caso in cui valga il "$\leq$" anziché l'"=". Infine, la soluzione resta valida anche se $T(n)$ è uguale ad una costante per $n \leq h$, con h costante intera non necessariamente uguale ad 1, oppure se al posto di n/b si ha $\lfloor n/b \rfloor$, oppure $\lceil n/b \rceil$. □

Esempio 16.4 (Mergesort). Per la procedura MERGESORT [cfr. § 11] le relazioni di ricorrenza sono del tipo: $T(n) \leq 2T(n/2) + cn$. Poiché $a = b = 2$ ed $\alpha = \log 2/\log 2 = 1 = \beta$, si ha che $T(n)$ è $O(n \log n)$. □

Esempio 16.5 (Ricerca binaria). Per la procedura RICBIN [cfr. § 6] si ha: $T(n) \leq T(n/2) + c$. Poiché $a = 1$, $b = 2$, $\alpha = \log 1/\log 2 = 0 = \beta$, ne segue che $T(n)$ è $O(\log n)$. □

Vediamo un altro problema che è risolvibile con un algoritmo più sofisticato di tipo "divide et impera", la cui complessità è valutabile utilizzando il teorema sulle ricorrenze lineari con partizione bilanciata.

16.1.1 Moltiplicazione di matrici

Si consideri il problema di moltiplicare tra loro due matrici quadrate $n \times n$ $A = [a_{ij}]$ e $B = [b_{ij}]$. Per definizione, la matrice prodotto $C = AB$ ha il generico elemento $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$, per $1 \leq i \leq n$ ed $1 \leq j \leq n$. Il problema può essere risolto facilmente con il seguente brano di programma Pascal:

```
for i := 1 to n do
    for j := 1 to n do begin
        C[i, j] := 0;
        for k := 1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j]
    end;
```

Questo banale algoritmo ha complessità $O(n^3)$. Vediamo come ridurre la complessità utilizzando la tecnica "divide et impera". Assumendo per semplicità $n = 2^h$, la matrice A può essere ripartita in quattro sottomatrici A_{11}, A_{12}, A_{21} e A_{22}, ciascuna di dimensione $n/2 \times n/2$. Ripartendo in modo analogo anche B e C, poiché $c_{ij} = \sum_{k=1}^{n/2} a_{ik}b_{kj} + \sum_{k=n/2+1}^{n} a_{ik}b_{kj}$, ogni sottomatrice C_{ij} della matrice prodotto può essere espressa come somma delle sottomatrici seguenti:

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} \qquad \text{per } 1 \leq i \leq 2 \text{ ed } 1 \leq j \leq 2.$$

Si definiscano adesso le sette matrici $X_1, \ldots, X_7$, ciascuna delle quali ha dimensione $n/2 \times n/2$ e può essere calcolata con un solo prodotto tra matrici:

$$\begin{aligned} X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ X_2 &= (A_{21} + A_{22})B_{11} \\ X_3 &= A_{11}(B_{12} - B_{22}) \\ X_4 &= A_{22}(B_{21} - B_{11}) \\ X_5 &= (A_{11} + A_{12})B_{22} \\ X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ X_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

Le matrici C_{ij} possono essere ricavate in funzione delle matrici $X_1, \ldots, X_7$ come segue:

$$\begin{aligned} C_{11} &= X_1 + X_4 - X_5 + X_7 \\ C_{12} &= X_3 + X_5 \\ C_{21} &= X_2 + X_4 \\ C_{22} &= X_1 + X_3 - X_2 + X_6 \end{aligned}$$

L'algoritmo risultante, proposto da Strassen (1969), può essere riassunto informalmente con la seguente procedura MOLTMATRICI:

```
procedure MOLTMATRICI(var A, B, C: matrice; n: integer);
    var X1, ... , X7: matrice;
    begin
        if n = 1 then
            c11 := a11 * b11
        else begin
            {ripartisci A in A11, A12, A21 e A22};
            {ripartisci B in B11, B12, B21 e B22};
            {calcola X1, ... , X7 richiamando MOLTMATRICI};
            {ricava C11, C12, C21 e C22 in funzione delle X1, ... , X7};
        end
    end;
```

Poiché le somme di matrici di dimensione $n/2 \times n/2$ richiedono $O(n^2)$ operazioni, e la procedura richiama se stessa sette volte per calcolare $X_1, \ldots, X_7$, il tempo $T(n)$ richiesto dall'algoritmo verifica le seguenti relazioni di ricorrenza, dove c e d sono costanti positive:

$$\begin{aligned} T(n) &= d, && \text{per } n = 1, \\ T(n) &\leq 7T(n/2) + cn^2, && \text{per } n = 2^h, h > 0. \end{aligned}$$

Applicando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene $\alpha = \log 7/\log 2 \approx 2.81 > 2 = \beta$. Pertanto, $T(n)$ risulta essere $O(n^{2.81})$.

L'algoritmo di Strassen è valido per n qualsiasi, non necessariamente per $n = 2^h$. Basta infatti bordare le matrici A e B con righe e colonne di zeri fino ad ottenere $n = 2^h$ e l'ordine di grandezza di $T(n)$ non cambia. Inoltre, il metodo può essere adattato per calcolare, con la stessa complessità, il determinante e l'inversa di una matrice. L'algoritmo è efficiente per n grande, mentre per n piccolo l'algoritmo banale di complessità $O(n^3)$ può risultare più veloce. Ciò è dovuto alle costanti "nascoste" nell'ordine di grandezza della complessità, che sono abbastanza grosse per l'algoritmo di Strassen. Sono noti persino algoritmi asintoticamente più efficienti di quello di Strassen (il più veloce ha complessità $O(n^{2.376})$, mentre una limitazione inferiore alla complessità del problema è ovviamente $\Omega(n^2)$), ma tali algoritmi non sono di utilità pratica a causa delle enormi costanti moltiplicative coinvolte negli ordini di grandezza delle funzioni di complessità.

16.2 Analisi per tentativi

Consideriamo adesso un metodo per individuare il comportamento asintotico della soluzione di una relazione di ricorrenza qualsiasi, che non sia necessariamente né lineare, né a coefficienti costanti, né di ordine costante. Tale metodo procede "per tentativi": si suppone che la soluzione sia di un certo tipo, per esempio $T(n) \leq cn^\beta$ con c e β costanti, la si sostituisce nella relazione di ricorrenza, e si cerca di dimostrare per induzione su n se è effettivamente una soluzione della ricorrenza. Purtroppo, non c'è una regola generale per individuare la soluzione corretta al primo tentativo, ed in questi casi occorre un po' di esperienza. Spesso si inizia tentando con soluzioni che sono sicuramente limitazioni superiori oppure inferiori "lasche", restringendo successivamente con tentativi di limitazioni "più strette". Per dimostrare correttamente un ordine di grandezza tramite induzione, però, bisogna prestare particolare

attenzione. Infatti, per non cadere miseramente in errori banali, occorre provare che la relazione vale per ogni $n \geq m$ (con m costante) ma sempre per le medesime costanti m e c.

Esempio 16.6 (Tentativi). Si consideri la relazione di ricorrenza

$$\begin{aligned} T(n) &= 1, && \text{per } n = 1, \\ T(n) &\leq 2T(n/2) + n, && \text{per } n = 2^h, h > 0, \end{aligned}$$

Si tenti di dimostrare per induzione che $T(n) \leq cn$, per qualche costante c. Chiaramente, $T(1) = 1$ verifica la base dell'induzione per $c \geq 1$. Assumendo per ipotesi induttiva che valga $T(n/2) \leq cn/2$, si ricava che $T(n) \leq 2(cn/2) + n \leq cn + n = cn(1 + 1/c)$. Ciò non autorizza a concludere che, poiché $(1 + 1/c)$ è $O(1)$, allora $T(n)$ è $O(n)$. Infatti, $(1 + 1/c) > 1$ e quindi non si si è trovato che $T(n) \leq cn$, ma che $T(n) \leq (c + 1)n$, dove $c + 1$ è una costante più grande di c. Pertanto, non è vero che $T(n/2) \leq cn/2$ e $T(n) \leq cn$ per la stessa costante c, e quindi $T(n)$ non è $O(n)$. Si tenti adesso con $T(n) \leq cn^2$. Dopo passaggi analoghi si ottiene che $T(n) \leq 2c(n/2)^2 + n = cn^2(1/2 + 1/(cn))$. Poiché la quantità $(1/2 + 1/(cn))$ è minore di 1 per n sufficientemente grande, la relazione $T(n) \leq cn^2$ vale per ogni $n \geq m$ per le stesse costanti c ed m, e quindi $T(n)$ è $O(n^2)$. Con qualche altro tentativo si può dimostrare che $T(n)$ è $O(n \log n)$, risultato che d'altronde sapevamo già dimostrare per altra via (sia per sostituzione che applicando il teorema sulle ricorrenze lineari con partizione bilanciata). □

Esemplifichiamo ulteriormente la tecnica considerando una relazione di ricorrenza che sorge nell'analisi del caso medio di un algoritmo ricorsivo per il problema della SELEZIONE e che ha una struttura ancora più complicata della relazione di ricorrenza incontrata nell'analisi della complessità del QUICKSORT nel caso medio [cfr. § 11].

16.2.1 Selezione

Dati un vettore A di n interi, ed un intero k tale che $1 \leq k \leq n$, il problema della SELEZIONE chiede di trovare il k-esimo elemento più piccolo. Si noti che per $k = 1$ ($k = n$) il problema si riduce a quello di trovare il minimo (massimo), mentre per $k = n/2$ il problema diventa quello del MEDIANO, dove si vuole trovare l'elemento di A che ha lo stesso numero sia di elementi minori che di elementi maggiori di esso.

Si può usare la procedura PERNO [cfr. § 11] in modo da trovare il k-esimo elemento più piccolo senza ordinare il vettore. Ricordiamo che la procedura PERNO scambia tra loro elementi di $A[primo..ultimo]$ e restituisce l'indice j del "perno" tale che $A[i] \leq A[j]$, per $primo \leq i \leq j$, e $A[i] \geq A[j]$, per $j \leq i \leq ultimo$. La seguente procedura SELEZIONE, proposta da Hoare (1961), richiama la PERNO ed individua così il q-esimo elemento più piccolo di $A[primo..ultimo]$, dove $q = j - primo + 1$ è il numero di elementi di $A[primo..j]$. Se $k \leq q$, allora SELEZIONE è riapplicata ad $A[primo..j]$; se invece $k > q$, allora SELEZIONE è riapplicata ad $A[q + 1..ultimo]$, ma per ricercare il $(k - q)$-esimo elemento più piccolo.

```
function SELEZIONE(var A: vettore; primo, ultimo, k: integer): integer;
    var q, j: integer;
    begin
        if primo = ultimo then
            SELEZIONE := A[primo]
        else begin
            PERNO(A, primo, ultimo, j);
            q := j − primo + 1;
            if k ≤ q then SELEZIONE := SELEZIONE(A, primo, j, k)
                else SELEZIONE := SELEZIONE(A, j + 1, ultimo, k − q)
        end
    end;
```

Assumendo che PERNO restituisca con la stessa probabilità una qualsiasi posizione j del vettore A, il numero medio di confronti $T(n)$ tra elementi di A effettuati da SELEZIONE è dato dalla relazione di ricorrenza

$$
\begin{aligned}
T(n) &= 0, && \text{per } n = 0, 1,\\
T(n) &= n - 1 + \frac{1}{n}\sum_{q=1}^{n-1} T(\max\{q, n-q\}) \le \\
&\le n - 1 + \frac{2}{n}\sum_{q=\lceil n/2\rceil}^{n-1} T(q), && \text{per } n \ge 2,
\end{aligned}
$$

perché $\max\{q, n-q\}$ è uguale a q, se $q \ge \lceil n/2\rceil$, mentre è uguale ad $n-q$, se $q < \lceil n/2\rceil$.

Effettuiamo un ragionamento informale per cercare di ricavare una buona soluzione sulla quale applicare la tecnica di dimostrazione per tentativi. In media, SELEZIONE dovrebbe essere richiamata ogni volta su una porzione dimezzata del vettore A. Poiché la PERNO richiede tempo lineare nella lunghezza della porzione di A, il numero complessivo di confronti dovrebbe crescere come $n+n/2+n/4+\cdots+1 \le 2n$. Pertanto, una soluzione promettente di $T(n)$ con la quale tentare sembra essere $O(n)$.

Tentiamo quindi con $T(n) \le cn$ per qualche costante c. La base dell'induzione è banalmente verificata perché $0 = T(0) = T(1) \le c$. Si assuma che valga l'ipotesi induttiva $T(q) \le cq$ per ogni $q \le n-1$. Sostituendo, si ricava:

$$
\begin{aligned}
T(n) &\le n - 1 + \frac{2c}{n}\sum_{q=\lceil n/2\rceil}^{n-1} q \le n + \frac{2c}{n}\left(\sum_{q=\lceil n/2\rceil}^{n-1} q - \sum_{q=1}^{\lceil n/2\rceil - 1} q\right) = \\
&= n + \frac{2c}{n}\left(n(n-1)/2 - \lceil n/2\rceil(\lceil n/2\rceil - 1)/2\right) \le \\
&\le n + \frac{2c}{n}\left(n(n-1)/2 - (n/2)(n/2-1)/2\right) = \\
&= n + c(n-1) - (c/2)(n/2-1) == n + cn - c - cn/4 + c/2 = \\
&= cn\left(\frac{1}{c} + \frac{3}{4} - \frac{1}{2n}\right)
\end{aligned}
$$

Poiché l'ultima quantità tra parentesi è minore di 1 per n sufficientemente grande e per una opportuna scelta di c, la relazione $T(n) \le cn$ vale per ogni $n \ge m$ per le stesse costanti c ed m. Pertanto, la funzione SELEZIONE trova il k-esimo elemento più piccolo di A in tempo lineare nel caso medio. Tanto per curiosità, esistono altri algoritmi più sofisticati per la SELEZIONE che richiedono tempo $O(n)$ nel caso pessimo, il primo dei quali è stato quello di Blum-Floyd-Pratt-Rivest-Tarjan (1973).

16.3 Potenziale

Nell'analizzare la complessità "ammortizzata" di una sequenza qualsiasi di n operazioni su una struttura di dati, è stata finora usata implicitamente la cosiddetta tecnica dell'"aggregazione" con la quale, basandosi sulla definizione stessa di complessità ammortizzata, si valuta prima il tempo complessivo $T(n)$ richiesto nel caso pessimo da qualsiasi sequenza di n operazioni e si ricava poi il tempo ammortizzato di ciascuna operazione dividendo il tempo complessivo per il numero di operazioni, ottenendo così $T(n)/n$.

Una tecnica molto usata nell'analisi ammortizzata è detta del "potenziale" perché impiega una funzione potenziale, che in un certo senso mantiene l'"energia potenziale" della struttura di dati. Tale potenziale rappresenta una sorta di "credito prepagato", accumulato durante l'esecuzione di operazioni che sono costate in qualche modo più del dovuto. Questo credito è usato in seguito per compensare il costo di operazioni sottopagate.

Formalmente, una funzione potenziale Φ associa ciascuna configurazione della struttura di dati ad un numero reale. Siano t_i e Φ_i, rispettivamente, il costo effettivo dell'i-esima operazione ed il valore della funzione potenziale dopo che è stata eseguita l'i-esima operazione, per $i = 1, 2, \ldots, n$.

Il costo ammortizzato a_i dell'i-esima operazione è il costo effettivo dell'operazione più l'incremento di potenziale dovuto all'operazione:

$$a_i = t_i + \Phi_i - \Phi_{i-1}.$$

Il costo ammortizzato della sequenza di n operazioni è pertanto:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n}(t_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} t_i + \Phi_n - \Phi_0,$$

dove Φ_n e Φ_0 sono il potenziale della configurazione finale ed iniziale, rispettivamente. Di solito, è opportuno che $\Phi_0 = 0$. Se $\Phi_i - \Phi_{i-1} > 0$, allora il costo ammortizzato a_i dell'i-esima operazione è più del costo effettivo t_i, e l'eccedenza va ad accrescere il potenziale. Se invece $\Phi_i - \Phi_{i-1} < 0$, il costo ammortizzato è meno di quello effettivo ed il potenziale accumulato in precedenza viene usato per compensare il debito. Se $\Phi_n - \Phi_0 \ge 0$, allora il costo ammortizzato della sequenza di operazioni rappresenta una limitazione superiore al costo effettivo della sequenza. Si osservi che, se $\Phi_i - \Phi_0 \ge 0$ per ogni i, allora si ha la garanzia di pagare sempre in anticipo il costo di ciascuna operazione.

Esempio 16.7 (Ristrutturazione di una tabella hash). Si riconsideri il problema della ristrutturazione di una tabella hash, discusso nel § 6, dove per semplicità si assume di eseguire una sequenza di sole INSERISCI. La ristrutturazione avviene quando il numero n di elementi presenti nella tabella raggiunge la metà della dimensione m

della tabella, e consiste nell'inserire tutti gli elementi in una nuova tabella con dimensione raddoppiata. Occorre definire una opportuna funzione potenziale che valga 0 immediatamente dopo una ristrutturazione, e che cresca in proporzione ai successivi inserimenti in modo da valere $m/2$ prima della successiva ristrutturazione, così da permettere l'inserimento di tutti gli elementi nella nuova tabella. Poiché $n = m/2$ prima della ristrutturazione ed $n = m/4$ dopo, una scelta ragionevole è

$$\Phi(n,m) = \begin{cases} 0 & \text{se } n < m/4, \\ 2n - m/2 & \text{se } n \geq m/4, \end{cases}$$

in quanto si ottiene proprio $\Phi(m/2, m) = m/2$ e $\Phi(m/4, m) = 0$. Ovviamente, il costo effettivo di una operazione di inserzione è 1, se non c'è ristrutturazione, mentre è n, se c'è ristrutturazione. Siano n_i, m_i, e Φ_i i valori di n, m, e Φ dopo l'i-esima INSERISCI, dove m_0 è la dimensione iniziale della tabella, mentre $n_0 = 0$, e $\Phi_0 = 0$.

Chiaramente, $n < m/4$ solo per le prime $m_0/4 - 1$ INSERISCI, dopodiché n non potrà più scendere sotto $m/4$ per come è stata progettata la ristrutturazione della tabella. Poiché per $i \leq m_0/4$ non ci può essere alcuna ristrutturazione ed il potenziale Φ_i è sempre nullo, risulta che $a_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 0 - 0 = 1$.

Per $i > m_0/4$, invece, si hanno due possibilità. Quando non c'è ristrutturazione, $m_{i-1} = m_i$ e quindi, essendo $n_{i-1} = n_i - 1$, si ottiene

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} = \\ &= 1 + (2n_i - m_i/2) - (2n_{i-1} - m_{i-1}/2) = \\ &= 1 + 2n_i - m_i/2 - 2(n_i - 1) + m_i/2 = \\ &= 3 \end{aligned}$$

Se invece c'è ristrutturazione, allora $m_i/2 = m_{i-1} = 2n_{i-1} = 2(n_i - 1)$, e si ha

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} = \\ &= n_i + (2n_i - m_i/2) - (2n_{i-1} - m_{i-1}/2) = \\ &= n_i + (2n_i - 2(n_i - 1)) - (2(n_i - 1) - (n_i - 1)) = \\ &= n_i + 2n_i - 2n_i + 2 - 2n_i + 2 + n_i - 1 = \\ &= 3 \end{aligned}$$

Intuitivamente, quando $i \leq m/4$, il costo ammortizzato è uguale a quello effettivo e si paga per l'inserimento immediato di una singola chiave, mentre il potenziale resta nullo. Quando invece $i > m/4$, ogni INSERISCI costa 3 inserimenti di chiavi, dei quali 1 è per l'inserimento immediato della chiave e 2 vanno ad accrescere il potenziale per inserimenti futuri. In questo modo, dopo ulteriori $m/4$ inserimenti, il numero di elementi diventa $n = m/4 + m/4 = m/2$, ed il potenziale raggiunge $2m/4 = m/2$. Pertanto, il potenziale può essere interamente utilizzato per inserire tutte le $m/2$ chiavi nella nuova tabella ristrutturata. □

Consideriamo adesso un altro utilizzo della tecnica del potenziale, per analizzare l'efficienza di algoritmi "on line".

16.3.1 Algoritmi "on line"

Un algoritmo è detto on line ("in linea") quando deve prendere decisioni irrevocabili senza avere una conoscenza completa dell'intera sequenza dei dati di ingresso. In altri termini, i dati in ingresso sono considerati uno alla volta, non appena disponibili, ed il risultato viene prodotto immediatamente in base ai soli dati conosciuti finora. Il risultato non solo è fornito senza conoscere i dati che saranno ricevuti successivamente, ma non può essere più modificato. Gli algoritmi on line sono in un certo senso opposti agli usuali algoritmi off line ("fuori linea") che abbiamo considerato finora, dove i dati in ingresso non sono elaborati via via che si rendono disponibili, ma sono elaborati insieme dopo che sono stati raccolti tutti.

Per valutare l'efficienza di un algoritmo on line nel caso pessimo, si usa confrontarlo con un algoritmo off line "ottimo", che minimizza cioè il costo per elaborare la sequenza in ingresso, come proposto da Sleator e Tarjan (1985). Sia A un algoritmo on line e sia S una sequenza di input. Si denoti con $A(S)$ il costo dell'algoritmo A per elaborare la sequenza S, e si denoti con $\mathrm{OPT}(S)$ il costo di un algoritmo off line ottimo per la stessa sequenza S. L'algoritmo A è detto c-competitivo se, per ogni S,

$$A(S) - c \cdot \mathrm{OPT}(S)$$

è limitata da una costante. Il minimo c per cui A resta c-competitivo è detto competitività di A. In pratica, un algoritmo on line è confrontato su tutte le sequenze col miglior algoritmo che può "vedere" tutta la sequenza in anticipo, e viene valutato il costo aggiuntivo dovuto dall'elaborazione in linea.

Poiché i dati, e quindi le operazioni che l'algoritmo deve effettuare, si presentano in sequenza, la competività di un algoritmo on line è spesso valutata tramite un'analisi ammortizzata che utilizza la tecnica del potenziale. Vediamone l'applicazione ad un problema su liste.

Nel problema dell'AGGIORNAMENTO ("List Update"), l'ingresso è dato da una sequenza di operazioni da eseguire su un dizionario realizzato con una lista monodirezionale non ordinata. Ciascuna operazione può essere APPARTIENE, INSERISCI, oppure CANCELLA. Per accedere ad un elemento, occorre scandire la lista dall'inizio ed il costo è uguale ad uno più il numero di elementi che precedono quello acceduto. Mentre si scandisce la lista, si può mantenere un puntatore alla posizione dove l'elemento deve essere eventualmente spostato. Pertanto, una volta che un elemento è stato raggiunto, può essere avvicinato alla testa della lista in tempo costante. L'elemento può essere spostato anche verso il fondo della lista, e in tal caso il costo è uguale alla distanza percorsa nello spostamento. Il costo per la cancellazione di un vecchio elemento o per l'inserzione di uno nuovo è uguale ad uno più il numero di elementi che precedono quello cancellato o inserito.

L'algoritmo on line MF ("Move-to-Front") consiste nello spostare l'elemento acceduto in testa alla lista. Dimostriamo che MF è 2-competitivo. Si assuma che sia MF che OPT (l'algoritmo off line ottimo), mentre elaborano le operazioni in ingresso, mantengano le loro liste $\mathrm{L_{MF}}$ ed $\mathrm{L_{OPT}}$ di elementi. Quando le due liste sono identiche, una qualsiasi operazione provocherà lo stesso costo sia in MF che in OPT. Pertanto, OPT ha la potenzialità di costare meno di MF solo quando le due liste sono diverse. Ciò suggerisce di definire la funzione potenziale seguente:

$$\Phi(\mathrm{L_{MF}}, \mathrm{L_{OPT}}) = \text{numero di inversioni tra coppie di elementi in } L_{\mathrm{MF}} \text{ ed } \mathrm{L_{OPT}}$$

dove c'è un'inversione tra una coppia di elementi se appaiono in $\mathrm{L_{MF}}$ con un ordine diverso da quello di $\mathrm{L_{OPT}}$. Usando questa funzione Φ, è possibile dimostrare che

per ciascuna operazione (cioè APPARTIENE, INSERISCI, e CANCELLA) il costo ammortizzato di MF non supera il doppio di quello di OPT. Per definizione, Φ è non negativa e vale 0 se MF ed OPT partono con la stessa lista. Supponiamo per esempio che l'i-esima operazione in ingresso sia una APPARTIENE. Si assuma che l'elemento acceduto x sia il j-esimo elemento di L_{OPT} dopo l'accesso. Allora il costo dell'algoritmo off line ottimo è almeno j. Si assuma che x sia l'h-esimo elemento di L_{MF} prima dell'accesso. Infine, sia m il numero di elementi che precedono x in L_{MF} ma che lo seguono in L_{OPT}. Allora, il numero di elementi che precedono x in entrambe le liste è $h - 1 - m$. Quando x è spostato in testa da MF, sono introdotte $h - 1 - m$ inversioni e ne sono eliminate m. Quindi il costo ammortizzato di MF se l'i-esima operazione è una APPARTIENE risulta:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} = \\ &= h + (h - 1 - m) - m = \\ &= 2(h - m) - 1 \leq \\ &\leq 2j - 1. \end{aligned}$$

Infatti, $h - m \leq j$ perché, degli $h - 1$ elementi che precedono x in L_{MF}, soltanto $j - 1$ elementi lo precedono in L_{OPT}. Pertanto, l'algoritmo on line MF non costa più del doppio dell'algoritmo off line ottimo.

16.4 Esercizi

Esercizio 16.1 (Numeri di Fibonacci). Si imposti la relazione di ricorrenza per il tempo $T(n)$ richiesto dall'algoritmo ricorsivo dell'Esercizio 14.1, e si risolva applicando il teorema delle ricorrenze lineari di ordine costante.

Esercizio 16.2 (Minimo in una matrice / Torneo di tennis). Si impostino le relazioni di ricorrenza per il tempo $T(n)$ richiesto dagli algoritmi "divide et impera" degli Esercizi 11.8 e 11.9, e si risolvano applicando il teorema delle ricorrenze lineari con partizione bilanciata.

Esercizio 16.3 (Moltiplicazione di polinomi). Si progetti un algoritmo di tipo "divide et impera" per moltiplicare tra loro due polinomi $p(x)$ e $q(x)$ di grado $n - 1$, in modo da utilizzare tre soli prodotti di polinomi di grado $n/2 - 1$ nella ricombinazione dei risultati dei sottoproblemi, e se ne analizzi la complessità.

Si supponga per semplicità che il numero n di coefficienti sia una potenza di 2. Si può dividere il polinomio $p(x)$ di grado $n - 1$ in due polinomi, ciascuno avente $n/2$ coefficienti:

$$\begin{aligned} p^S(x) &= p_{n-1}x^{n/2-1} + p_{n-2}x^{n/2-2} + \cdots + p_{n/2+1}x + p_{n/2} \\ p^D(x) &= p_{n/2-1}x^{n/2-1} + p_{n/2-2}x^{n/2-2} + \cdots + p_1x + p_0 \end{aligned}$$

Dividendo anche $q(x)$ nello stesso modo, si ha:

$$\begin{aligned} p(x) &= x^{n/2}p^S(x) + p^D(x) \\ q(x) &= x^{n/2}q^S(x) + q^D(x) \end{aligned}$$

e il prodotto $r(x) = p(x)q(x)$ può essere scritto come:

$$r(x) = x^n p^S(x)q^S(x) + x^{n/2}[p^D(x)q^S(x) + q^D(x)p^S(x)] + p^D(x)q^D(x).$$

Calcolando una volta per tutte, con solo tre prodotti, i polinomi:

$$\begin{aligned} r^S(x) &= p^S(x)q^S(x) \\ r^D(x) &= p^D(x)q^D(x) \\ r^M(x) &= [p^S(x) + p^D(x)][q^S(x) + q^D(x)] \end{aligned}$$

il prodotto $r(x)$ si può ottenere nel modo seguente:

$$r(x) = x^n r^S(x) + x^{n/2}[r^M(x) - r^D(x) - r^S(x)] + r^D(x).$$

Questa formula permette di calcolare il prodotto tra due polinomi di dimensione n risolvendo tre sottoproblemi di dimensione $n/2$, e addizionando tra loro polinomi per "preparare" i sottoproblemi e "combinare" le soluzioni. Osservando che la somma di polinomi ed il prodotto per $x^{n/2}$ e x^n richiedono $O(n)$ operazioni e che per $n = 1$ il problema si riduce ad una singola moltiplicazione tra due coefficienti costanti, il tempo $T(n)$ richiesto dall'algoritmo verifica le seguenti relazioni di ricorrenza:

$$\begin{aligned} T(n) &= c && \text{per } n = 1, \\ T(n) &\leq 3T(n/2) + dn, && \text{per } n = 2^h \text{ con } h > 0, \end{aligned}$$

dove c e d sono costanti. Essendo $\alpha = \log 3/\log 2 \approx 1.59 > 1 = \beta$, la complessità $T(n)$ dell'algoritmo è $O(n^{1.59})$.

Esercizio 16.4 (Analisi di algoritmi ricorsivi). Per ciascuna delle seguenti funzioni Pascal si valuti l'ordine di grandezza del tempo $T(n)$.

```
function PIPPO(n: integer): integer;
    var i: integer;
    begin
        if n ≤ 2 then PIPPO := 1
        else if n > 999 then begin
            i := n div 2;
            PIPPO := PIPPO(i) + n * i
        end else PIPPO := PIPPO(n − 2) + 5
    end;

function FOLLIA(n: integer): integer;
    var i, j: integer;
    begin
        if n > 10 then begin
            for i := n downto n − 3 do i := n div 2;
            FOLLIA := FOLLIA(n div 2) div FOLLIA(n div 2) + n * 5
        end else
            if n > 20 then FOLLIA := FOLLIA(n + 1)
                else FOLLIA := n
    end;
```

Per la funzione PIPPO si ha:

$$T(n) = d, \qquad \text{per } n \leq 999,$$
$$T(n) = T(n/2) + c, \qquad \text{per } n > 999,$$

con c e d costanti. Applicando il teorema delle ricorrenze lineari con partizione bilanciata, si ha: $a = 1$, $b = 2$, $\alpha = 0$, $\beta = 0$, ed essendo $\alpha = \beta = 0$, $T(n)$ risulta essere $O(\log n)$.

Per la funzione FOLLIA si ricava:

$$T(n) = d, \qquad \text{per } n \leq 10,$$
$$T(n) = 2T(n/2) + c, \qquad \text{per } n > 10,$$

con c e d costanti, poiché "**then** FOLLIA := FOLLIA($n + 1$)" non è mai eseguito. Per il il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene: $a = 2$, $b = 2$, $\alpha = 1$, $\beta = 0$, ed essendo $\alpha > \beta$, $T(n)$ è $O(n)$.

Esercizio 16.5. Scrivere una funzione Pascal che riceve come parametro un vettore di n elementi tale che il suo tempo di esecuzione $T(n)$ verifichi la relazione

$$T(n) = d, \qquad \text{per } n \leq k,$$
$$T(n) = 5T(n/2) + 2T(n-1) + cn^2 \qquad \text{per } n > k,$$

con c, d e k costanti opportune. L'ordine di grandezza di $T(n)$ è polinomiale oppure no?

Esercizio 16.6 (Analisi ricorrenze). Si dimostri che la ricorrenza

$$T(n) = d, \qquad \text{per } n \leq k,$$
$$T(n) = 2T(\sqrt{n}) + \log n \qquad \text{per } n > k,$$

con d e k costanti opportune, ha soluzione $O(\log n \log\log n)$ (Suggerimento: si effettui il cambio di variabile $m = \log n$).

Esercizio 16.7 (Analisi per tentativi). Si dimostri per tentativi che la ricorrenza

$$T(n) = d, \qquad \text{per } n \leq k,$$
$$T(n) = 3T(n/4) + n\log n, \qquad \text{per } n > k,$$

con d e k costanti opportune, ha soluzione $O(n \log n)$.

Esercizio 16.8 (Analisi ammortizzata). Si consideri un contatore binario di k bit, che conta "circolarmente" a partire da 0. Si scriva una procedura Pascal per incrementare il contatore, realizzandolo con un vettore di k interi che valgono ciascuno 0 o 1, secondo il ben noto schema di addizione di 1 (modulo 2^k) con propagazione del riporto. Si analizzi la complessità ammortizzata di una sequenza di n operazioni di incremento, prima col metodo di aggregazione e poi con quello del potenziale.

Parte terza

Complessità e decidibilità

Capitolo 17

Non determinismo ed enumerazione

Per trascorrere il tempo durante un viaggio in aereo, ci si può essere cimentati con il seguente rompicapo, o con qualche sua variante semplificata:

DOMINO LIMITATO: "Dati un intero positivo n ed un insieme finito D di m tipi di "tessere orientate", cioè di quadrati di lato unitario, divisi in quattro parti dalle diagonali con ciascun quarto colorato con un colore, è possibile ricoprire un'area quadrata di lato n con copie delle tessere in D in modo che:

1. nessuna tessera è "ruotata";
2. una particolare tessera $d \in D$ occupa la posizione più in basso a sinistra;
3. due tessere che si toccano hanno i lati adiacenti dello stesso colore?"

Esempio 17.1 (Domino limitato). La Fig. 17.1 illustra due insiemi D e D', con $m = 3$ tipi di tessere ciascuno. Posto $n = 3$, è facile vedere che esiste una copertura legale con tessere di D (Fig. 17.1a), mentre non ne esiste alcuna con tessere di D' (Fig. 17.1b), dove tutte le scelte effettuate sono obbligate e non c'è alcun modo di coprire la casella rimasta vuota). Si noti che se fossero ammesse rotazioni, allora la sola tessera d, qualunque essa fosse, sarebbe sufficiente a coprire ogni area di dimensione arbitraria! □

Per dati di ingresso banali, come quelli dell'Esempio 17.1, il DOMINO LIMITATO può essere risolto a colpo d'occhio, ma per n ed m "grandi" e tessere "perverse", cercare una soluzione può essere alquanto frustrante. Ciò accade specialmente quando esistono molte soluzioni "parziali", che coprono cioè solo alcune porzioni del quadrato di lato n, ma ci sono pochissime soluzioni "complete" che coprono l'intero quadrato.

Un algoritmo banale che risolve il problema "provando tutte le possibilità" non è però difficile da trovare. Si osservi infatti che nel quadrato di lato n ci sono n^2 caselle da coprire con copie delle tessere. Per la casella in basso a sinistra la scelta è obbligata, mentre ognuna delle rimanenti caselle può essere coperta con una copia di una tessera scelta tra m possibili tipi diversi. Il numero di coperture distinte è pertanto m^k, con $k = n^2 - 1$, e l'algoritmo deve semplicemente generarle tutte e verificare, per ognuna di esse, che la copertura sia "legale", ovvero che tessere adiacenti abbiano i lati che si toccano dello stesso colore. Purtroppo, questo banale algoritmo non è utilizzabile in pratica, poiché richiede un inaccettabile tempo esponenziale!

Il DOMINO LIMITATO è il problema "più difficile" tra tutti quelli visti finora. Infatti, per quasi tutti i problemi trattati nei capitoli precedenti (p.e. CAMMINI MINIMI,

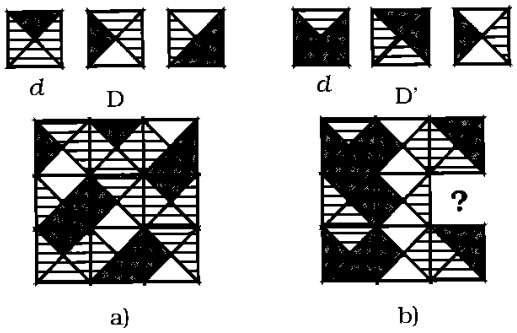


Figura 17.1: DOMINO LIMITATO ($m = 3$, $n = 3$). a) Insieme D di tessere con copertura (risposta SI); b) Insieme D' di tessere senza copertura (risposta NO).

ORDINAMENTO, INVILUPPO CONVESSO, STRING MATCHING), oltre ad algoritmi banali del tipo "prova tutte le possibilità", erano stati introdotti algoritmi molto efficienti se non addirittura ottimi. Per il DOMINO LIMITATO, al contrario, non solo non è conosciuto un algoritmo migliore di quello che "prova tutte le possibilità", ma si suppone addirittura che un tale algoritmo non esista! Nella stessa situazione stanno molti altri problemi, come, per esempio, i seguenti:

COLORAZIONE: "Dati un grafo non orientato ed un intero k, è possibile colorarne i nodi usando al più k colori in modo che ogni nodo sia colorato con un colore diverso da tutti i nodi adiacenti?"

CRICCA: "Dati un grafo non orientato ed un intero k, esiste un sottoinsieme di almeno k nodi tutti mutuamente adiacenti?"

COMMESSO VIAGGIATORE (o "TSP"): "Date n città, le distanze tra esse, ed un intero k, è possibile partire da una città, attraversare ogni città esattamente una volta, e ritornare alla città di partenza, percorrendo una distanza complessiva non superiore a k?"

PROGRAMMAZIONE LINEARE 0/1: "Data una matrice A di elementi interi e di dimensione $m \times n$, ed un vettore b di m elementi interi, esiste un vettore x di n elementi 0/1 tale che $Ax \leq b$?"

SODDISFATTIBILITÀ: "Data un'espressione booleana in forma normale congiuntiva, esiste un'assegnazione di valori di verità vero/falso alle variabili booleane che rende l'espressione vera?"

Esempio 17.2 (COLORAZIONE). La Fig. 17.2 mostra una colorazione con 3 colori per un grafo che rappresenta la relazione di adiacenza tra le 10 province della Toscana. Per ogni k maggiore o uguale a 3 il problema ha risposta SI, mentre per k minore di 3 la risposta è NO. □

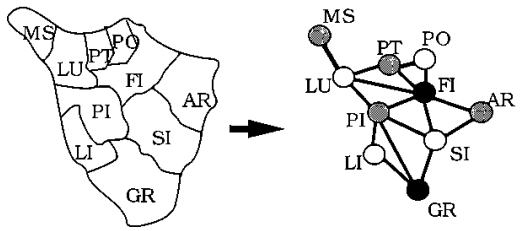


Figura 17.2: Una colorazione con 3 colori del grafo che rappresenta la relazione di adiacenza tra province della Toscana.

Esempio 17.3 (CRICCA). Nel grafo illustrato nella Fig. 17.2, esiste una cricca con 3 nodi (p.e. {FI, PT, PO} o {PI, GR, SI}), ma non esiste alcuna cricca con più di 3 nodi. □

Esempio 17.4 (COMMESSO VIAGGIATORE). La Fig. 17.3 mostra la tabella delle distanze (in linea d'aria) tra sette località della Toscana ed un percorso ottimo per il commesso viaggiatore. Per ogni k inferiore alla distanza ottima il problema ha risposta NO, mentre per ogni k maggiore o uguale a tale distanza la risposta è ovviamente SI. □

Esempio 17.5 (PROGRAMMAZIONE LINEARE 0/1). Il sistema:

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &\geq 2 \\ x_1 - x_2 - x_3 + x_4 &\geq 0 \\ x_1 \qquad + x_3 + x_4 &\leq 1 \end{aligned}$$

è verificato per $x_1 = x_2 = 1$ ed $x_3 = x_4 = 0$. □

Esempio 17.6 (SODDISFATTIBILITÀ). La formula su u_1, u_2 e u_3:

$$(\textbf{not } u_1 \textbf{ or not } u_2 \textbf{ or not } u_3) \textbf{ and } (u_2 \textbf{ or not } u_3) \textbf{ and } (u_1 \textbf{ or not } u_2 \textbf{ or } u_3)$$

è soddisfatta per $u_1 =$ **false** e $u_2 = u_3 =$ **true**. □

Oltre a questi problemi, centinaia di altri problemi di rilevante interesse pratico si trovano nella stessa "barca". Tutti questi problemi sono tra loro computazionalmente correlati e destinati ad "affogare" o a "salvarsi" insieme. Ognuno può essere risolto in tempo superpolinomiale, ma per nessuno è noto un algoritmo polinomiale. Inoltre, se uno solo fosse risolubile in tempo polinomiale, allora lo sarebbero tutti, ma si suppone che non esista alcun algoritmo polinomiale anche per uno solo di questi problemi!

17.1 Certificati polinomiali

Oltre ad essere tutti di tipo decisionale, i problemi elencati precedentemente hanno in comune un'altra semplice proprietà. Benché sia estremamente difficile rispondere SI o NO, è abbastanza facile convincersi se una ipotetica soluzione dia risposta SI. Formalmente, tutti questi problemi possiedono un certificato polinomiale,

	1	2	3	4	5	6	7
1		68	42	89	44	37	85
2	68		29	30	75	28	19
3	42	29		42	41	14	54
4	89	30	42		80	52	36
5	44	75	41	80		50	94
6	37	28	14	52	50		49
7	85	19	54	36	94	49	

distanze in linea d'aria

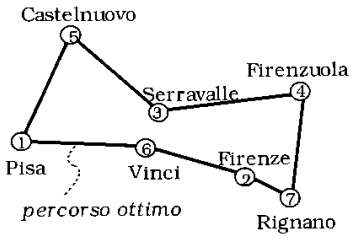


Figura 17.3: Il percorso ottimo per il COMMESSO VIAGGIATORE tra sette località della Toscana.

cioè un algoritmo che, data una presunta soluzione del problema, verifica in tempo polinomiale che tale soluzione sia effettivamente una soluzione che dà risposta SI.

Esempio 17.7 (Certificato per DOMINO LIMITATO). Dati D, n, ed una copertura, il certificato verifica in tempo $O(n^2)$ che tale copertura sia "legale", cioè che una tessera di tipo d copra l'angolo in basso a sinistra, che ogni tessera usata sia una copia di una in D, e che le 4 tessere ad essa adiacenti abbiano i lati colorati con gli opportuni colori. □

Esempio 17.8 (Certificato per CRICCA). Dati $G = (N, A)$, k, ed un insieme S, il certificato verifica in tempo $O(n^2)$ che S sia un sottoinsieme di N, che per ogni coppia u, $v \in S$ risulti $[u, v] \in A$, e che la cardinalità di S sia maggiore o uguale a k. □

Esempio 17.9 (Certificato per COMMESSO VIAGGIATORE). Dati k, la matrice $D = [d_{ij}]$ delle distanze tra le n città, ed una presunta permutazione π di $1, 2, \ldots, n$, il certificato verifica in tempo $O(n)$ che π sia effettivamente una permutazione, cioè contenga ogni i, $1 \leq i \leq n$, una e una sola volta, e che $\sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$ non superi k. □

Un modo equivalente di descrivere questo fenomeno consiste nella possibilità di risolvere tutti questi problemi con algoritmi non deterministici di complessità polinomiale.

17.2 Non determinismo

Un algoritmo non deterministico è un algoritmo che, posto di fronte alla necessità di dover effettuare una "decisione", ha la "virtù magica" di scegliere sempre la strada giusta! In termini equivalenti, è come se l'algoritmo, di fronte a più alternative da poter seguire, le seguisse tutte contemporaneamente, generando più "copie" di se stesso. Ciascuna copia dell'algoritmo procede la computazione, indipendentemente dalle altre, seguendo una e una sola delle alternative possibili.

Esempio 17.10 (Non determinismo). In un racconto dello scrittore francese Marcel Aymé, Sabina, la protagonista, ha il dono sovrannaturale di generare una nuova copia

di se stessa ogni volta che vuole sedurre un nuovo amante. Le varie Sabine hanno avventure indipendenti le une dalle altre e, al termine del racconto, il numero di Sabine è salito a qualche migliaio! □

Gli algoritmi non deterministici sono di interesse puramente teorico, ma non sono ovviamente utilizzabili in pratica. Per definirli in modo formale, si supponga di avere a disposizione le tre seguenti istruzioni elementari, ciascuna delle quali richieda $O(1)$ tempo di computazione:

choice(I), che sceglie arbitrariamente un elemento dell'insieme finito I;
failure, che blocca la computazione in uno stato di "fallimento";
success, che blocca la computazione in uno stato di "successo".

Un algoritmo non deterministico è un algoritmo che, per risolvere un problema decisionale, utilizza queste tre istruzioni. Quando viene eseguita una istruzione **choice**(I), l'algoritmo produce $|I|$ copie di se stesso, ed ogni copia procede indipendentemente dalle altre dopo aver "scelto" un elemento di I. L'algoritmo fornisce risposta SI se e solo se esiste una sequenza di scelte che portano allo stato di "successo", mentre fornisce risposta NO altrimenti. Si noti che, per definizione, è data risposta SI se una qualsiasi copia esegue un'istruzione **success**, mentre è data risposta NO se tutte le copie eseguono un'istruzione **failure**!

Esempio 17.11 (Algoritmo non deterministico per CRICCA). Sia S un vettore di n elementi booleani, con n uguale al numero di nodi del grafo $G = (N, A)$. Un algoritmo non deterministico per il problema CRICCA è il seguente:

```
procedure NDCRICCA({grafo G = (N, A)}; n, k: integer);
    var i, j, h: integer; S: vettore;
    begin
        for i := 1 to n do S[i] := choice({true, false});
        h := 0;
        for i := 1 to n do
            if S[i] then begin
                h := h + 1;
                for j := 1 to i − 1 do
                    if S[j] and [i, j] ∉ A then failure
            end;
        if h ≥ k then success else failure;
    end;
```

Nel primo ciclo **for** vengono effettuate le scelte ed il nodo i va a far parte della soluzione parziale se $S[i]$ = **true**. Col secondo **for**, invece, si verifica che la soluzione costruita sia una cricca. Se il nodo i fa parte della soluzione parziale, allora si verifica col ciclo **for** più interno che per ogni nodo $j < i$ che sta anch'esso nella soluzione esista l'arco $[i, j]$. Se non è stata trovata una cricca (perché $[i, j] \notin A$) allora si termina con fallimento. Se invece si è trovata una cricca e questa ha cardinalità h maggiore o uguale a k, allora si termina con successo, altrimenti con fallimento. La complessità della procedura è chiaramente $O(n^2)$, in quanto per ipotesi ogni istruzione **choice**, **success**, e **failure** richiede tempo $O(1)$. □

Esempio 17.12 (Algoritmo non deterministico per COMMESSO VIAGGIATORE). Sia S un vettore di n elementi interi, con n uguale al numero delle città. Sia I un insieme contenente ad ogni iterazione le città non ancora scelte ed inizializzato in modo che $I = \{1, 2, \ldots, n\}$. Un algoritmo non deterministico per il COMMESSO VIAGGIATORE è il seguente:

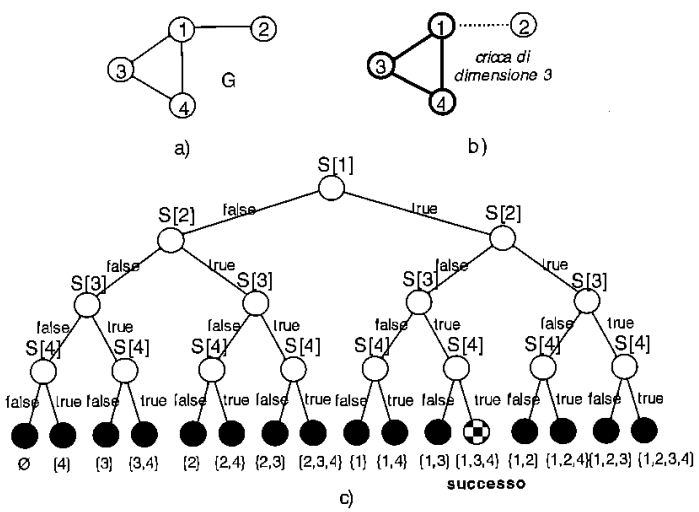


Figura 17.4: Procedura NDCRICCA. a) Il grafo G; b) Una cricca di dimensione 3; c) albero delle scelte: i nodi bianchi corrispondono ad esecuzioni di istruzioni **choice**, quelli neri ad esecuzioni di **failure**, e quello a scacchi a una **success**.

```
procedure NDTSP(n, k: integer; d: matrice);
    var i, c: integer; S: vettore; I: insieme;
    begin
        I := {1, 2, ... , n};
        for i := 1 to n do begin
            S[i] := choice(I);
            I := I − {S[i]}
        end;
        c := d[S[n], S[1]];
        for i := 2 to n do
            c := c + d[S[i − 1], S[i]];
        if c ≤ k then success else failure;
    end;
```

Nel primo ciclo **for** vengono effettuate le scelte, e la città scelta $S[i]$ è eliminata dall'insieme I in modo che non possa essere più scelta. Col secondo **for**, invece, si calcola la lunghezza c della soluzione costruita. Se tale lunghezza è minore o uguale a k, allora si termina con successo, altrimenti con fallimento. La complessità della procedura dipende da come è realizzato l'insieme I. Per esempio, se la cancellazione di $S[i]$ da I richiede $O(n)$ tempo, allora la complessità di NDTSP risulta essere $O(n^2)$. □

Un algoritmo non deterministico è caratterizzato da una computazione ad albero radicato, detto albero delle scelte. I nodi interni dell'albero rappresentano "soluzioni parziali", gli archi rappresentano "scelte", e le foglie rappresentano un "successo" o un "fallimento". Ogni percorso radice-foglia dell'albero corrisponde ad una sequenze di scelte, ovvero alla computazione di una "copia" dell'algoritmo non deterministico.

Esempio 17.13 (Albero delle scelte per NDCRICCA). La Fig. 17.4 mostra un grafo G e il corrispondente albero delle scelte ottenuto eseguendo NDCRICCA su G per $k = 3$. □

Gli algoritmi non deterministici operano una sequenza di scelte $S[1], S[2], \ldots$ per ottenere la soluzione. Tale sequenza di scelte è talmente correlata alla soluzione stessa da venire spesso identificata con essa. Ogni scelta $S[i]$ avviene su un sottoinsieme I_i che è funzione delle scelte precedenti $S[1], \ldots, S[i-1]$ effettuate dall'algoritmo. Supponendo per semplicità che il numero delle scelte sia limitato da n, lo "scheletro" generale di una procedura non deterministica è il seguente:

```
procedure NONDETERMINISTICA({opportuni parametri});
    {opportune dichiarazioni};
    begin
        {opportune inizializzazioni};
        for i := 1 to n do begin
            {determina I_i in funzione di S[1], ... , S[i − 1]};
            if I_i = ∅ then
                failure
            else begin
                S[i] := choice(I_i);
                if {S[1] , ... , S[i] è una soluzione} then success;
            end;
        end;
    end;
```

Si noti che la procedura è caratterizzata da due fasi distinte: nella prima fase viene costruita una ipotetica soluzione, mentre nella seconda viene certificata. Se entrambe queste fasi richiedono tempo polinomiale, allora la complessità della procedura è polinomiale. Ciò accade per tutti i problemi precedentemente elencati!

17.3 Enumerazione

Poiché il non determinismo non è realistico, occorre "simulare" il comportamento di una procedura non deterministica con una deterministica. Ma ciò corrisponde a generare ed esplorare deterministicamente lo spazio delle soluzioni, ovvero visitare sistematicamente l'albero delle scelte corrispondente alla computazione non deterministica, che ha un numero superpolinomiale di nodi!

Esempio 17.14 (Spazio delle soluzioni). È immediato constatare dalla Fig. 17.4 che l'albero delle scelte per NDCRICCA è un albero binario completo con tutte le foglie al livello n e pertanto ha $O(2^n)$ nodi. Inoltre, è facile convincersi che l'albero delle scelte per NDTSP ha invece $O(n!)$ nodi. Infatti, la prima scelta effettuata da NDTSP nella radice dà luogo ad n possibili alternative, la seconda scelta fatta nei figli della radice ad $n-1$ alternative, ed in generale l'i-esima scelta ad $n-i+1$ alternative, per $i = 1, 2, \ldots, n$. □

La procedura NONDETERMINISTICA si può così trasformare in una procedura deterministica ENUMERAZIONE equivalente. Assumendo ancora per semplicità che il numero delle scelte sia limitato da n, si ottiene il seguente "scheletro" di procedura, da chiamarsi con l'istruzione ENUMERAZIONE(1):

```
procedure ENUMERAZIONE(i: integer);
    {opportune dichiarazioni};
    begin
        if i ≤ n then begin
            {determina I_i in funzione di S[1], ... , S[i − 1]};
            if I_i ≠ ∅ then
                for j := 1 to |I_i| do
                    begin
                        S[i] := j-esimo elemento di I_i;
                        if {S[1] , ... , S[i] è una soluzione} then
                            {arresta la computazione con successo}
                        else
                            ENUMERAZIONE(i + 1);
                    end;
        end;
    end;
```

Se $|I_i| \leq m$ per $1 \leq i \leq n$, la complessità di ENUMERAZIONE è $O(m^n)$. Si noti che tale procedura si basa sulla tecnica di "backtrack" e la sua struttura è quella di una visita di un albero radicato avente $O(m^n)$ nodi. Tale procedura è stata illustrata in forma ricorsiva per ragioni di chiarezza e concisione, ma può essere trasformata per esercizio in una iterativa equivalente utilizzando le regole di trasformazione definite nel § 3.

Esempio 17.15 (Enumerazione per CRICCA). Si assuma per semplicità che il vettore booleano S, gli interi n e k, e il grafo $G = (N, A)$ siano parametri globali. Una specializzazione della procedura ENUMERAZIONE per la CRICCA è la seguente:

```
procedure ENCRICCA(i, h: integer);
    var cricca: boolean; j: integer;
    begin
        if i ≤n then begin
            S[i] := false;
            ENCRICCA(i + 1, h);
            S[i] := true;
            cricca := true;
            for j := 1 to i − 1 do
                if S[j] and [i, j] ∉ A
                    then cricca := false;
            if cricca then
                if h + 1 = k
                    then {arresta la computazione con successo}
                    else ENCRICCA(i + 1, h + 1);
        end
    end;
```

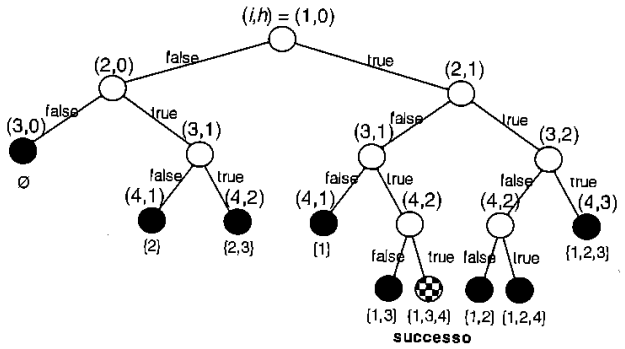



Figura 17.5: Albero della computazione della procedura ENCRICCA eseguita col confronto iniziale "$k - h \leq n - i + 1$" sul grafo G della Fig. 17.4 per $k = 3$.

La chiamata nel programma principale è ENCRICCA(1, 0). Nella generica chiamata ENCRICCA(i, h), il parametro i è il nodo di G che viene considerato per essere inserito oppure no nella soluzione parziale, mentre il parametro h è la dimensione della soluzione parziale costruita. Se i non è inserito nella soluzione ($S[i]$ = **false**), allora viene richiamata ENCRICCA($i + 1$, h), poiché la dimensione h della soluzione non è aumentata. Se invece i è inserito nella soluzione ($S[i]$ = **true**), si verifica che la soluzione parziale costruita sia una cricca oppure no, controllando che il nuovo nodo aggiunto sia connesso con tutti gli altri nodi $j < i$ che fanno parte della soluzione parziale (cioè tali che $S[j]$ = **true**). Se *cricca* = **true**, la nuova soluzione è una cricca di dimensione $h + 1$, poiché è stato aggiunto il nodo i. Se la dimensione $h + 1$ ha raggiunto k allora si arresta la computazione con successo, mentre se è minore di k si richiama ENCRICCA($i + 1$, $h + 1$), poiché deve essere effettuata almeno una ulteriore scelta. La complessità della procedura è chiaramente $O(2^n)$.

Per evitare del lavoro inutile, si può sostituire il confronto iniziale "$i \leq n$" col confronto "$k - h \leq n - i + 1$". Infatti, $n - i + 1$ è il numero di nodi del grafo ancora da considerare per l'eventuale inserimento nella soluzione, mentre $k - h$ è il numero di nodi che mancano per raggiungere la dimensione k richiesta della cricca. Se $k - h > n - i + 1$, non ci sono abbastanza nodi per raggiungere la dimensione voluta ed è inutile proseguire. L'albero della computazione generato da questa variante di ENCRICCA per $G = (N, A)$, con $N = \{1, 2, 3, 4\}$ e $A = \{[1, 2], [1, 3], [1, 4], [3, 4]\}$, e $k = 3$, è mostrato nella Fig. 17.5. Si noti che ENCRICCA genera tale albero in ordine anticipato. Benché la variante eviti di generare una parte dell'albero, la complessità della procedura resta comunque $O(2^n)$. □

Esempio 17.16 (Enumerazione per COMMESSO VIAGGIATORE). Si assuma ancora che il vettore S, gli interi n e k, e la matrice delle distanze d siano parametri globali. Si può rappresentare l'insieme I delle città non ancora scelte con un vettore inizializzato in modo che $I[i] = i$, per $i = 1, 2, \ldots, n$. La procedura ENUMERAZIONE può essere quindi specializzata per risolvere il COMMESSO VIAGGIATORE come segue:

```
procedure ENTSP(i, c: integer);
    var j: integer;
    begin
        if i ≤ n then
            for j := i to n do begin
                S[i] := I[j];
                I[j] := I[i];
                if i > 1 then c := c + d[S[i − 1], S[i]];
                ENTSP(i + 1, c);
                if i > 1 then c := c − d[S[i − 1], S[i]];
                I[i] := I[j];
                I[j] := S[i];
            end
        else begin
            c := c + d[S[n], S[1]];
            if c ≤ k then
                {arresta la computazione con successo}
        end
    end;
```

La chiamata nel programma principale è ENTSP(1,0). Nel ciclo **for** sono generate tutte le permutazioni di $I[i..n]$ (cfr. Esercizio 12.4). In accordo alla definizione ricorsiva stessa di permutazione, tutte le permutazioni di $I[i..n]$ iniziano a turno con la città scelta $S[i] = I[j]$, per $i \leq j \leq n$, e sono seguite dalle permutazioni delle restanti città non ancora scelte. Poiché la successiva chiamata ricorsiva è effettuata su $I[i+1..n]$, la città scelta $I[j]$ è sostituita prima della chiamata con la città non scelta $I[i]$, che altrimenti non sarebbe più considerata. Inoltre, la lunghezza c del percorso parziale è incrementata di $d[S[i-1], S[i]]$ se $i > 1$. La configurazione di $I[i..n]$ e il valore di c prima della scelta sono ripristinati al ritorno dalla chiamata interna, rimettendo $I[i]$ ed $S[i]$ al loro posto, e sottraendo $d[S[i-1], S[i]]$ a c. Quando $i > n$, è stato generato un percorso completo per il commesso viaggiatore, cioè una permutazione di tutte le città $1, 2, \ldots, n$, e viene calcolata la lunghezza della soluzione individuata sommando $d[S[n], S[1]]$ a c. La complessità della procedura ENTSP è $O(n!)$.

Per risparmiare tempo, si può evitare di generare n volte la stessa soluzione, corrispondente alle n permutazioni ottenute tramite traslazione ciclica destra (come ad esempio "1, 2, 3, 4", "4, 1, 2, 3", "3, 4, 1, 2" e "2, 3, 4, 1" per $n = 4$). A tal fine, si può fissare la prima scelta una volta per tutte nel programma principale, per esempio ponendo $S[1] = 1$, e quindi richiamare direttamente ENTSP(2). In questo modo, poiché risulta sempre $i > 1$, la chiamata ricorsiva interna diventa quindi ENTSP($i + 1$, $c + d[S[i-1], S[i]]$). La complessità della procedura resta comunque $O((n-1)!)$. □

17.4 Esercizi

Esercizio 17.1 (Tangentopoli). Dati un insieme M di n "mazzette" ed un insieme P di n "politici corrotti" appartenenti al Partito Democratico dei Lavoratori (PDL) oppure al Partito Popolare Cristiano (PPC), si vogliono assegnare tutte le mazzette a tutti i politici in modo che il PDL riceva complessivamente il doppio del PPC. Si scriva una procedura non deterministica di complessità polinomiale.

Si rappresentino le mazzette con un vettore M di n interi ed i politici con un vettore P di n booleani tale che $P[i]$ = **true** se il politico i è del PDL e $P[i]$ = **false** se è del PPC. Si ottiene la seguente procedura:


```
procedure MAZZETTE(var M, S: mazzette; var P: politici);
    var j, i, MPDL, MPPC: integer;
    begin
        for i := 1 to n do begin
            j := choice {i, i + 1, ..., n};
            S[i] := M[j];
            M[j] := M[i];
        end;
        MPDL := 0; MPPC := 0;
        for i := 1 to n do
            if P[i]
                then MPDL := MPDL + S[i]
                else MPPC := MPPC + S[i];
        if MPDL = 2 * MPPC then success
            else failure
    end;
```

Esercizio 17.2 (Zaino 0/1). Dati un insieme di n oggetti, con profitti $\{p_1, \ldots, p_n\}$ e volumi $\{v_1, \ldots, v_n\}$, tutti interi positivi, e due interi positivi c e k, il problema dello ZAINO 0/1 (cfr. Esercizi 13.1 e 13.2) consiste nel determinare un insieme $\{x_1, \ldots, x_n\}$ di interi uguali a 0 oppure 1 tali che $\sum_{1 \leq i \leq n} p_i x_i \geq k$ e $\sum_{1 \leq i \leq n} v_i x_i \leq c$. Si scriva una procedura non deterministica di complessità polinomiale per il problema dello ZAINO 0/1.

```
procedure NDZAINO(var p, v: vettore; c, k: integer);
    var S: vettore; volume, profitto, i: integer;
    begin
        for i := 1 to n do
            S[i] := choice({true, false});
        volume := 0; profitto := 0;
        for i := 1 to n do
            if S[i] then begin
                volume := volume + v[i];
                profitto := profitto + p[i]
            end;
        if volume ≤ c and profitto ≥ k
            then success
            else failure
    end;
```

Esercizio 17.3 (Colorazione). Si scriva una procedura non deterministica di complessità polinomiale che risolva il problema della COLORAZIONE.

Esercizio 17.4 (Tangentopoli). Si scriva una procedura deterministica di enumerazione per il problema dell'Esercizio 17.1.

Esercizio 17.5 (Zaino 0/1). Si scriva una procedura deterministica di enumerazione per il problema dello ZAINO 0/1 dell'Esercizio 17.2.

Esercizio 17.6 (Colorazione). Si scriva una procedura deterministica di enumerazione per il problema della COLORAZIONE.

Esercizio 17.7 (3-Copertura di insiemi). Si scriva una procedura deterministica di enumerazione per il problema della 3-COPERTURA DI INSIEMI: "Dati un insieme $X = \{1, 2, \ldots, m\}$ ed una famiglia $Y = \{Y_1, \ldots, Y_n\}$ di n sottoinsiemi di X, ciascuno avente cardinalità 3, esiste una sottofamiglia S di insiemi di Y che partiziona X?"

Si assuma che la famiglia Y sia rappresentata con una matrice $n \times 3$, e che la sottofamiglia S sia rappresentata con un vettore booleano. Per verificare che S partizioni X, si utilizza un vettore booleano $B[1..m]$. Se Y_j è stato selezionato ($S[j]$ = **true**), allora $B[Y[j, k]]$ è posto a **true** per $k = 1, 2, 3$. Se però $B[Y[j, k]]$ era già **true**, allora non si ha una partizione. Infine, si verifica che non sia rimasto alcun elemento di B a **false**, poiché anche in tal caso non si ha una partizione. La chiamata è ENCOPERTURA(i) e la complessità è $O((n + m)2^n)$.

```
procedure ENCOPERTURA(i: integer);
    var B: vettore; j, k: integer; partizione: boolean;
    begin
        if i ≤ n then begin
            S[i] := false;
            ENCOPERTURA(i + 1);
            S[i] := true;
            ENCOPERTURA(i + 1);
        end else begin
            partizione := true;
            for j := 1 to m do B[j] := false;
            for j := 1 to n do
                if S[j] then
                    for k := 1 to 3 do
                        if B[Y[j, k]]
                            then partizione := false
                            else B[Y[j, k]] := true;
            for j := 1 to m do
                if not B[j] then partizione := false;
            if partizione then
                {arresta la computazione con successo};
        end;
    end;
```

Capitolo 18

Problemi NP-completi

Abbiamo visto che la complessità di un algoritmo non deterministico è proporzionale alla profondità dell'albero delle scelte, cioè al massimo livello delle foglie, e che una simulazione deterministica richiede tempo esponenziale in tale profondità. A questo punto è lecito chiedersi: "è mai possibile una simulazione deterministica che richieda tempo polinomiale"? Allo stato attuale delle conoscenze non si può dire nulla, ma esiste un vasto insieme di problemi, che stanno nella "barca" insieme a DOMINO LIMITATO, COLORAZIONE, CRICCA, COMMESSO VIAGGIATORE, PROGRAMMAZIONE LINEARE 0/1, e SODDISFATTIBILITÀ, per i quali:

1. si conoscono solo algoritmi polinomiali non deterministici;
2. non si conosce alcun algoritmo polinomiale deterministico;
3. se si trovasse un solo algoritmo polinomiale deterministico per un problema qualsiasi, allora se ne potrebbero trovare per tutti.

Tali problemi, detti "NP-completi", sono oggetto di studio del presente capitolo.

18.1 Le classi $\mathbb{P}$ ed $\mathbb{NP}$

Per poter trattare in modo omogeneo problemi estremamente eterogenei tra loro (come problemi su grafi, algebrici, geometrici, numerici) facciamo alcune ipotesi *semplificative*:

a. i problemi siano di tipo decisionale;
b. il linguaggio di programmazione usato sia il Pascal, con l'aggiunta delle operazioni **choice**, **success** e **failure**;
c. la codifica dei numeri utilizzati nel calcolo sia concisa, ovvero si adotti una rappresentazione in base $b > 1$.

Vediamo di giustificare queste tre ipotesi e di mostrare come esse non siano affatto restrittive. Innanzitutto, un problema di tipo decisionale non può essere più difficile di uno di ricerca o di ottimizzazione. Pertanto, se si dimostra l'intrattabilità computazionale del problema in forma decisionale, allora si è dimostrata l'intrattabilità del problema formulato in uno degli altri due modi.

Esempio 18.1 (Formulazioni per CRICCA). Il problema della CRICCA è stato formulato nel § 17 sotto forma decisionale. Nella formulazione di ricerca, si richiede di trovare una cricca di dimensione maggiore o uguale a k, e in quella di ottimizzazione

di trovare una cricca di dimensione massima. Ovviamente, se si risolve il problema di ricerca, allora si risolve anche quello decisionale, perché l'esibizione di una cricca è una prova della sua esistenza. Inoltre, se si risolve il problema di ottimizzazione, allora si risolvono anche gli altri due. Infatti, una cricca di dimensione massima h permette sia di rispondere al problema decisionale per ogni k, in quanto la risposta è SI per ogni $k \leq h$ e NO per ogni $k > h$, che di esibire una soluzione del problema di ricerca per ogni $k \leq h$ e di affermare che non esiste una soluzione per $k > h$. □

In secondo luogo, come vedremo anche nel § 26, il Pascal è un linguaggio di programmazione "universale", cioè non ci sono problemi risolubili con algoritmi scritti in un altro linguaggio, ma non in Pascal. Non solo, ma il linguaggio Pascal è "polinomialmente correlato" agli altri linguaggi universali, cioè non ci sono problemi di complessità polinomiale che non siano risolubili con procedure Pascal anch'esse di complessità polinomiale (al più, può cambiare il grado del polinomio). Pertanto, non è restrittivo considerare il linguaggio Pascal, integrato dalle operazioni **choice**, **success** e **failure**.

Infine, è ben noto che, utilizzando i calcolatori elettronici, tutti i dati sono rappresentati alla fin fine con sequenze di bit. Comunque, rappresentare i dati di un problema in base 2 o in base 5 non influenza la risolubilità polinomiale del problema. Infatti, è facilmente verificabile come si possa passare in tempo polinomiale dalla rappresentazione in base 2 di un dato alla sua rappresentazione in base 5, e viceversa. In altri termini, finché si codificano numeri con basi maggiori di 1, la dimensione dell'input, rappresentata in una qualsiasi base $b > 1$, è polinomialmente correlata alla dimensione dello stesso input rappresentata con un'altra base qualsiasi $b' > 1$. Questa proprietà invece non è verificata rappresentando l'input con una base unaria, in cui il numero k è codificato con una sequenza di k uni. Evitando di usare la base $b = 1$, si impedisce che il problema diventi "difficile" solo a causa di una inopportuna codifica non concisa.

Esempio 18.2 (Codifica in base $b > 1$). Nell'ORDINAMENTO, abbiamo assunto la dimensione del problema pari al numero n di elementi da ordinare. A voler essere rigorosi, la dimensione dovrebbe essere $n \log_b a$, con $a = \max\{a_i\}$, poiché per codificare il numero a in base $b > 1$ occorrono $\log_b a$ caratteri.[1] Anche per a molto grande, comunque, la funzione $\log_b a$ cresce polinomialmente e le due dimensioni $d_1 = n$ e $d_2 = n \log_b a$ sono tra loro polinomialmente correlate. Per esempio, anche se il valore a del più grande elemento in ingresso cresce esponenzialmente col numero n di elementi da ordinare, cioè se $a = c^n$ con $c > 1$, allora $\log_b a$ è $O(n)$ e le due dimensioni d_1 e d_2 sono tali che d_2 è $O(d_1^2)$ mentre d_1 è $O(\sqrt{d_2})$. Usando la base $b = 1$, questo non è più vero. La codifica di a richiede a caratteri, e la dimensione $d_3 = na$ non è più polinomialmente correlata né a $d_1 = n$ né a $d_2 = n \log_b a$. Infatti, per $a = c^n$, si ha $d_3 = na = nc^n$, che è esponenziale in n, e quindi d_3 è $O(d_1 n^{d_1})$. □

Possiamo così definire le classi $\mathbb{P}$ ed $\mathbb{NP}$ come segue:

$\mathbb{P}$ è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi deterministici;

[1] Nel provare la limitazione inferiore $\Omega(n \log n)$ per l'ORDINAMENTO, si è implicitamente assunto che i numeri a_i fossero codificati con una quantità costante di caratteri. Questa ipotesi, benché teoricamente scorretta, è realistica. Infatti, una parola di memoria di un elaboratore elettronico ha 32 bit, che permettono di codificare numeri grandi fino a $2^{31} - 1$ (un bit è usato per il segno). Numeri più grandi di $2^{31} - 1$ assai raramente si incontrano in pratica.

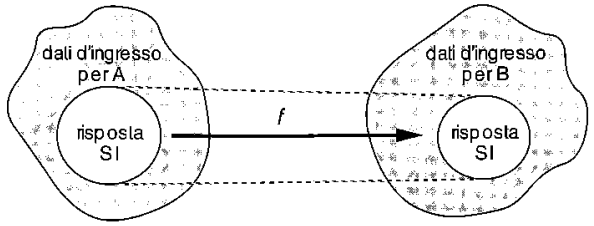


Figura 18.1: Riduzione da un problema A ad un problema B.

$\mathbb{NP}$ è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi non deterministici.

Ovviamente, la classe $\mathbb{P}$ è contenuta nella classe $\mathbb{NP}$, poiché ogni algoritmo deterministico è anche un algoritmo non deterministico (che semplicemente non usa mai **choice**), ma non si sa se $\mathbb{P}$ sia propriamente contenuta in $\mathbb{NP}$ oppure se le due classi coincidano. Tale questione resta a tuttoggi la più importante e famosa questione irrisolta dell'Informatica!

18.2 Riducibilità polinomiale

Tra problemi in $\mathbb{NP}$ è utile definire una relazione $\propto$ di riducibilità polinomiale. Siano A e B due problemi decisionali. Si dice che A si riduce in tempo polinomiale a B, e si scrive $A \propto B$, se esiste una funzione f di trasformazione

$$f\text{: (dati d'ingresso per } A) \rightarrow \text{(dati d'ingresso per } B)$$

tale che:

1. f è computabile in tempo polinomiale con un algoritmo deterministico;
2. x è un dato d'ingresso per cui A dà risposta SI se e solo se $f(x)$ è un dato d'ingresso per cui B dà risposta SI (si veda la Fig. 18.1).

In pratica, se si dimostra che $A \propto B$, si trasforma un qualsiasi dato d'ingresso per A in un particolare dato d'ingresso per B e quindi A risulta essere, in un certo senso, un sottoproblema di B! Si hanno le seguenti conseguenze:

I. Se $B \in \mathbb{NP}$, allora anche $A \in \mathbb{NP}$. Infatti, per risolvere il problema A con un algoritmo polinomiale non deterministico su un dato x, basta trasformare x in $f(x)$ e risolvere il problema B sul dato $f(x)$: l'algoritmo risultante è ovviamente polinomiale non deterministico.
II. Se $B \in \mathbb{P}$, allora anche $A \in \mathbb{P}$. Infatti, si opera come nel caso precedente e poiché la trasformazione f è computabile con un algoritmo polinomiale deterministico, l'algoritmo risultante è ovviamente polinomiale deterministico.
III. Se una limitazione inferiore alla complessità di A è $\Omega(p(n))$ e la trasformazione f richiede al più tempo $O(p(n))$, allora $\Omega(p(n))$ è anche una limitazione inferiore alla complessità del problema B.

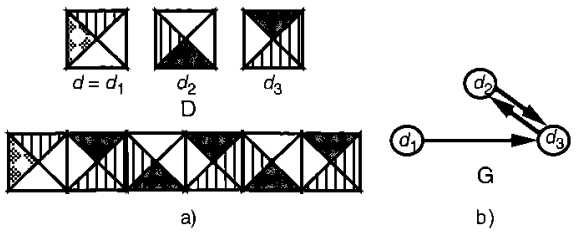


Figura 18.2: DOMINO LINEARE ($n = 6, m = 3$). a) Insieme D dei tipi di tessere con copertuta (risposta SI). b) Il grafo G corrispondente con un ciclo raggiungibile da $d = d_1$.

Vediamo un paio di esempi che, sebbene non siano per problemi decisionali, ma per problemi di ricerca e di ottimizzazione, si prestano bene ad illustrare la tecnica di riduzione polinomiale e le sue conseguenze.[2]

Esempio 18.3 (DOMINO LINEARE $\in \mathbb{P}$). Si consideri una semplificazione del DOMINO LIMITATO, nota come DOMINO LINEARE, in cui si chiede di coprire un rettangolo di altezza unitaria e larghezza uguale ad n. Questo problema può essere risolto in tempo polinomiale con un algoritmo deterministico riducendolo ad un problema di "visita" di un grafo orientato che, come abbiamo visto nel § 9, è in $\mathbb{P}$. Infatti, dati l'insieme D di m tipi di tessere, inclusa la tessera $d \in D$ da sistemare a sinistra, e l'intero n, si costruisca un grafo orientato $G = (N, A)$ il cui insieme di nodi N corrisponde all'insieme delle tessere D, ed esiste l'arco $(d_i, d_j) \in A$ se e solo se il lato destro della tessera d_i ha lo stesso colore del lato sinistro della tessera d_j. Chiaramente, il grafo G può essere costruito in $O(m^2)$ tempo, con $m = |D| = |N|$. È facile verificare che esiste una copertura "legale" per il DOMINO LINEARE se e solo se esiste un cammino in G a partire dal nodo corrispondente a d che include $n - 1$ archi e/o esiste un ciclo in G i cui nodi sono raggiungibili con un cammino che parte dal nodo corrispondente a d. Poiché queste due condizioni sono verificabili in tempo polinomiale applicando una BFS opportunamente modificata, il DOMINO LINEARE è risolubile con un algoritmo polinomiale deterministico. Un esempio di riduzione è fornito nella Fig. 18.2 per $m = 3$ tipi di tessere ed $n = 6$. □

Esempio 18.4 (INVILUPPO CONVESSO è $\Omega(n \log n)$). Dimostriamo che il problema dell'INVILUPPO CONVESSO è $\Omega(n \log n)$ fornendo una riduzione ORDINAMENTO $\propto$ INVILUPPO CONVESSO in cui la trasformazione f ha complessità $O(n)$. Siano dati n numeri $a_1, \ldots, a_n$ per l'ORDINAMENTO, che per semplicità assumiamo interi positivi e tra loro distinti. Si costruisca un insieme di n punti $p_1, \ldots, p_n$ del piano

[2] Il concetto di riduzione è già stato implicitamente utilizzato nei § 10 e § 15, dove si richiedeva di formulare alcuni problemi "incogniti" (l'INSIEME DOMINANTE per grafi di intervalli e l'ABBINAMENTO per grafi bipartiti [cfr. Esercizi 10.4, 10.5, 10.6 e 15.9]), come problemi di CAMMINI MINIMI o di FLUSSO MASSIMO. In effetti, la riduzione potrebbe essere considerata come una sesta tecnica di progetto di algoritmi, che però in generale non porta ad algoritmi molto efficienti (con l'eccezione notevole dell'ABBINAMENTO per grafi bipartiti). Infatti, riducendo un problema particolare ad uno più generale, in genere si perdono alcune proprietà specifiche del problema di partenza che possono essere sfruttate per progettare algoritmi specializzati molto più efficienti.

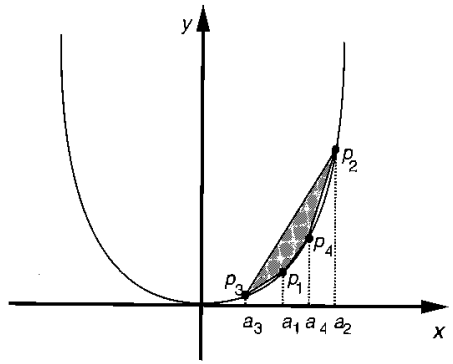


Figura 18.3: Esempio di riduzione ORDINAMENTO $\propto$ INVILUPPO CONVESSO ($n = 4$).

per l'INVILUPPO CONVESSO come segue. Si consideri la parabola $y = x^2$, e si ponga $p_i = (a_i, a_i^2)$, $1 \le i \le n$. La soluzione dell'INVILUPPO CONVESSO è data dai vertici del medesimo che, per i dati costruiti, sono tutti gli n punti ordinati per ascisse crescenti. Pertanto, ogni algoritmo che risolva l'INVILUPPO CONVESSO deve poter implicitamente ordinare le n ascisse $a_1, \ldots, a_n$, cioè risolvere l'ORDINAMENTO. Un esempio è illustrato nella Fig. 18.3 per $n = 4$. Poiché quest'ultimo problema è $\Omega(n \log n)$, come si è visto nel § 4, e la trasformazione f è $O(n)$, anche l'INVILUPPO CONVESSO è $\Omega(n \log n)$, e quindi l'algoritmo di Graham descritto nel § 12 è ottimo! □

18.3 Teorema di Cook-Levin

Agli inizi degli anni settanta, S. Cook e L. Levin si sono chiesti se esistesse un particolare problema decisionale in $\mathbb{NP}$ tale che, se si dimostrasse la sua appartenenza a $\mathbb{P}$, allora dovrebbe risultare sicuramente $\mathbb{P} = \mathbb{NP}$. La risposta a questo quesito è affermativa: il problema esiste ed è il DOMINO LIMITATO! Il risultato fondamentale, dimostrato indipendentemente dall'uno e dall'altro, recita:

Teorema di Cook-Levin. *Ogni problema in $\mathbb{NP}$ si riduce in tempo polinomiale al* DOMINO LIMITATO. □

La dimostrazione del teorema, benché concettualmente non molto difficile, è un po' macchinosa e sarà presentata dettagliatamente nel § 26. Per sommi capi, si assume di avere un calcolatore elettronico nella cui memoria sono caricati un programma Pascal non deterministico per risolvere il problema A ed un dato d'ingresso x per A di dimensione n. Poiché A è in $\mathbb{NP}$, il calcolatore risolverà A con dato x in tempo $p(n)$, con p funzione polinomiale in n. Poiché ogni istruzione del calcolatore manipola una quantità costante d'informazione, il calcolatore utilizzerà complessivamente un numero $cp(n)$ di bit di memoria, con c costante. Si costruisce allora un insieme opportuno D di $(cp(n))^k$ tessere, con k costante, tali che la regione da "coprire" corrisponda grossomodo a (numero di bit di memoria) $\times$ (numero di istruzioni eseguite). Le tessere sono definite "astutamente" in modo che la prima riga sia ricopribile se e

solo se l'input x è corretto per A, la riga i sia ricopribile dopo la riga $i-1$ se e solo se il calcolatore effettua una operazione corretta per risolvere A, e l'ultima riga sia ricopribile se e solo se il calcolatore si arresta su **success**.

Il teorema di Cook-Levin ha come conseguenza il seguente risultato:

Corollario. $\mathbb{P} = \mathbb{NP}$ *se e solo se* DOMINO LIMITATO $\in \mathbb{P}$.

Dimostrazione. Infatti, se DOMINO LIMITATO $\in \mathbb{P}$ ed un problema $A \in \mathbb{NP}$, allora anche $A \in \mathbb{P}$, poiché $A \propto$ DOMINO LIMITATO. Viceversa, se DOMINO LIMITATO $\notin \mathbb{P}$, allora $\mathbb{P} \neq \mathbb{NP}$, poiché DOMINO LIMITATO $\in \mathbb{NP}$. □

In pratica, questo corollario del teorema di Cook-Levin asserisce che il DOMINO LIMITATO è il più difficile problema in $\mathbb{NP}$. Formalmente, un problema siffatto viene detto "NP-completo".

Un problema A è detto "NP-arduo" se $B \propto A$ per ogni problema $B \in \mathbb{NP}$. A è detto "NP-completo" se, oltre ad essere NP-arduo, appartiene ad $\mathbb{NP}$. Il DOMINO LIMITATO è dunque NP-completo, ma non è il solo problema NP-completo. Da quando è stato dimostrato il teorema di Cook-Levin, sono stati scoperti centinaia di altri problemi NP-completi, inclusi CRICCA, COLORAZIONE, COMMESSO VIAGGIATORE, PROGRAMMAZIONE LINEARE 0/1 e SODDISFATTIBILITÀ. Poiché la relazione $\propto$ è transitiva (cioè se $P \propto Q$ e $Q \propto R$, allora anche $P \propto R$), tutti i problemi NP-completi sono computazionalmente equivalenti: o sono tutti risolubili in tempo polinomiale con algoritmi deterministici (e allora $\mathbb{P} = \mathbb{NP}$) oppure sono tutti intrinsecamente superpolinomiali (e quindi $\mathbb{P} \neq \mathbb{NP}$). Fino al giorno d'oggi, non è noto alcun algoritmo polinomiale deterministico per un problema NP-completo, e si pensa che non lo sarà mai! L'esistenza di un tale algoritmo è legata alla possibilità di "simulare" un algoritmo non deterministico che opera in tempo polinomiale con un algoritmo deterministico che operi anch'esso in tempo polinomiale, ma allo stato attuale non si conoscono che simulazioni superpolinomiali. La tesi più accreditata, comunque, è che tale simulazione sia intrinsecamente superpolinomiale e che quindi la classe $\mathbb{P}$ sia propriamente contenuta nella classe $\mathbb{NP}$.

18.4 Prove di NP-completezza

Per dimostrare che un problema A è NP-completo, non è necessario mostrare ogni volta che $B \propto A$ per ogni problema $B \in \mathbb{NP}$. Infatti, per la transitività della relazione $\propto$, "basta" mostrare che:

1. $A \in \mathbb{NP}$;
2. esiste un problema B, già notoriamente NP-completo, tale che $B \propto A$.

In pratica, poiché $B \propto A$ implica da sola che A è almeno tanto difficile quanto B, si fa a meno di mostrare la parte (1), e si dimostra direttamente, e in modo costruttivo, la parte (2). In linea di principio, un qualsiasi problema B che sia NP-arduo può essere un candidato per provare la riduzione, ma in pratica occorre "fiuto" per scegliere un problema B la cui "somiglianza" con A agevoli la dimostrazione. Un problema che si presta abbastanza bene, per generalità e flessibilità, a provare riduzioni polinomiali è comunque il DOMINO LIMITATO. Vediamo come questo problema possa essere ridotto a CRICCA, SODDISFATTIBILITÀ, e PROGRAMMAZIONE LINEARE 0/1.

Teorema 18.1. CRICCA *è NP-completa.*

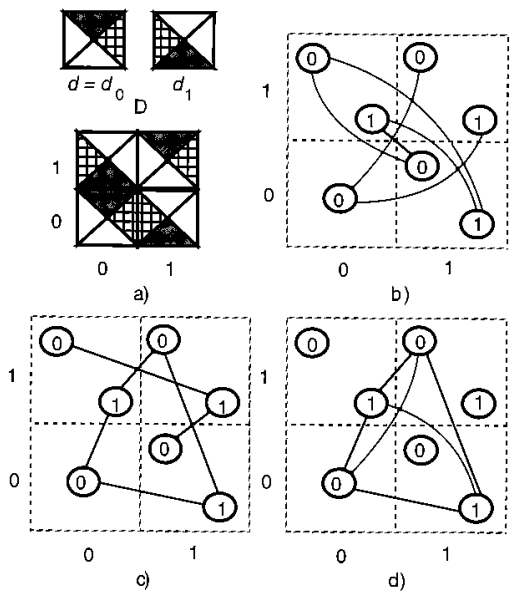


Figura 18.4: DOMINO LIMITATO $\propto$ CRICCA. a) Insieme D di $m = 2$ tipi di tessere e copertura di un quadrato di lato $n = 2$. b) Archi di G corrispondenti a caselle non adiacenti (condizione 3). c) Archi di G corrispondenti a caselle adiacenti (condizione 4). d) Una cricca di dimensione $k = n^2 = 4$ corrispondente alla copertura mostrata in a).

Dimostrazione. Mostriamo che DOMINO LIMITATO $\propto$ CRICCA. Siano dati $n > 0$, $D = \{d_0, d_1, \ldots, d_{m-1}\}$, con $d = d_0$, e indichiamo il quadrato di lato n da ricoprire con $[0, n-1] \times [0, n-1]$. Definiamo un grafo non orientato $G = (N, A)$ ed un intero k come segue:

1. Per ogni casella $(i,j) \neq (0,0)$ in $[0, n-1] \times [0, n-1]$ ed ogni $d_h \in D$, con $0 \leq h \leq m-1$, N contiene un nodo u_{hij} (l'appartenenza di u_{hij} ad una cricca "codifica" il fatto che la tessera d_h copre la casella (i,j));
2. Per $(i,j) = (0,0)$, N contiene il nodo u_{000} (che "codifica" la copertura di (0,0) con la tessera $d = d_0$);
3. Ciascun nodo u_{hij} è connesso con un arco $[u_{hij}, u_{h'i'j'}] \in A$ ad ogni nodo $u_{h'i'j'}$ tale che:

   3.1 $i' = i, j' \neq j$ e $j' \neq j \pm 1$, oppure
   3.2 $j' = j, i' \neq i$ ed $i' \neq i \pm 1$, oppure
   3.3 $j' \neq j$ ed $i' \neq i$;

   (cioè a nodi corrispondenti a caselle (i', j') distinte da (i, j) e che non sono adiacenti ad (i, j));

4. Ciascun nodo u_{hij} è connesso con un arco $[u_{hij}, u_{h'i'j'}] \in A$ ad ogni nodo $u_{h'i'j'}$ tale che:

   4.1 d_h e $d\,h'$ hanno i lati adiacenti dello stesso colore, ed inoltre
   4.2 $i' = i$ e $j' = j \pm 1$, oppure
   4.3 $j' = j$ e $i' = i \pm 1$,

   (cioè a nodi corrispondenti a caselle (i', j') che sono adiacenti alla casella (i, j) e che possono essere coperti in modo "legale" con due tessere);

5. $k = n^2$.

È facile verificare che la riduzione richiede tempo polinomiale in n ed m. Per costruzione, il problema DOMINO LIMITATO ha una copertura "legale" se e solo se il grafo G ha una cricca di dimensione k. Poiché CRICCA $\in \mathbb{NP}$, se n'è dimostrata l'NP-completezza. □

Esempio 18.5 (DOMINO LIMITATO $\propto$ CRICCA). La Fig. 18.4 mostra un esempio di riduzione da un insieme D contenente $m = 2$ tessere per coprire un quadrato di lato $n = 2$. Per semplicità, sono disegnati separatamente gli archi corrispondenti a caselle non adiacenti (condizione (3): Fig. 18.4b) e quelli corrispondenti a caselle adiacenti (condizione (4): Fig. 18.4c). La Fig. 18.4a) mostra una copertura legale e la Fig. 18.4d) la corrispondente cricca. □

Teorema 18.2. SODDISFATTIBILITÀ *è NP-completa.*

Dimostrazione. Forniamo una riduzione dal DOMINO LIMITATO. Siano dati $n > 0$ e $D = \{d_0, d_1, \ldots, d_{m-1}\}$, con $d = d_0$. Reinterpretiamo le condizioni di ricoprimento del DOMINO LIMITATO in termini di una formula booleana composta da "clausole" tra loro in "**and**". Si introducono le variabili logiche u_{hij}, $0 \le h \le m-1$, $0 \le i, j \le n-1$, con il seguente significato:

$$u_{hij} = \textbf{true} \text{ se e solo se la tessera } d_h \text{ copre la casella } (i, j).$$

Si definiscono quindi le seguenti clausole:

1. u_{000} (la clausola u_{000} "codifica" la copertura di (0,0) con la tessera d);
2. $\textbf{and}_{0 \le i,j \le n-1}$ $(\textbf{or}_{0 \le h \le m-1} u_{hij})$ (queste clausole asseriscono che almeno una tessera copre ciascuna casella del quadrato $[0, n-1] \times [0, n-1]$);
3. $\textbf{and}_{0 \le i,j \le n-1}$ $(\textbf{and}_{0 \le h < h' \le m-1}$ $(\textbf{not } u_{hij} \textbf{ or not } u_{h'ij}))$ (queste clausole asseriscono che al più una tessera copre ciascuna casella del quadrato $[0, n-1] \times [0, n-1]$);
4. $\textbf{and}_{0 \le i \le n-2, 0 \le j \le n-1}$ $(\textbf{and}_{h,h'}$ $(\textbf{not } u_{hij} \textbf{ or not } u_{h'i+1j}))$ per ogni coppia di tessere d_h e $d_{h'}$ che non possono essere adiacenti orizzontalmente, con d_h alla sinistra di $d_{h'}$;
5. $\textbf{and}_{0 \le i \le n-1, 0 \le j \le n-2}$ $(\textbf{and}_{h,h'}$ $(\textbf{not } u_{hij} \textbf{ or not } u_{h'ij+1}))$ per ogni coppia di tessere d_h e $d_{h'}$ che non possono essere adiacenti verticalmente, con d_h sotto $d_{h'}$.

La formula booleana F data dall'**and** di tutte le clausole definite in (1)-(5) è in forma normale congiuntiva e contiene un numero polinomiale, in n ed m, di variabili logiche, di clausole, e di variabili (affermate o negate) per clausola. Inoltre, il problema DOMINO LIMITATO ha una copertura "legale" per i dati n e D se e solo se la formula F è soddisfattibile. Poiché la riduzione richiede tempo polinomiale in n ed m e SODDISFATTIBILITÀ $\in \mathbb{NP}$, si è dimostrato che tale problema è NP-completo. □

Esempio 18.6 (DOMINO LIMITATO $\propto$ SODDISFATTIBILITÀ). Consideriamo un altro esempio di riduzione a partire dagli stessi dati di input del DOMINO LIMITATO mostrati nella Fig. 18.4a). Oltre alla clausola u_{000}, la condizione (2) determina l'**and** delle seguenti 4 clausole:

$$(u_{000} \textbf{ or } u_{100}) \textbf{ and } (u_{001} \textbf{ or } u_{101}) \textbf{ and } (u_{010} \textbf{ or } u_{110}) \textbf{ and } (u_{011} \textbf{ or } u_{111})$$

Per la condizione (3) si ottiene invece:

$$(\textbf{not } u_{000} \textbf{ or not } u_{100}) \textbf{ and } (\textbf{not } u_{001} \textbf{ or not } u_{101}) \textbf{ and}$$
$$(\textbf{not } u_{010} \textbf{ or not } u_{110}) \textbf{ and } (\textbf{not } u_{011} \textbf{ or not } u_{111})$$

Dalla condizione (4) deriva:

$$(\textbf{not } u_{000} \textbf{ or not } u_{010}) \textbf{ and } (\textbf{not } u_{001} \textbf{ or not } u_{011}) \textbf{ and}$$
$$(\textbf{not } u_{100} \textbf{ or not } u_{010}) \textbf{ and } (\textbf{not } u_{101} \textbf{ or not } u_{011}) \textbf{ and}$$
$$(\textbf{not } u_{100} \textbf{ or not } u_{110}) \textbf{ and } (\textbf{not } u_{101} \textbf{ or not } u_{111})$$

Infine, per la condizione (5) si ha:

$$(\textbf{not } u_{000} \textbf{ or not } u_{001}) \textbf{ and } (\textbf{not } u_{010} \textbf{ or not } u_{011}) \textbf{ and}$$
$$(\textbf{not } u_{100} \textbf{ or not } u_{101}) \textbf{ and } (\textbf{not } u_{110} \textbf{ or not } u_{111})$$

Mettendo tutti gli spezzoni in **and**, si ottiene la formula F che è soddisfatta per l'assegnamento dei valori di verità

$$u_{000} = u_{011} = u_{101} = u_{110} = \textbf{true}, \; u_{001} = u_{010} = u_{100} = u_{111} = \textbf{false},$$

che corrisponde alla copertura del DOMINO LIMITATO mostrata nella Fig. 18.4a). □

Teorema 18.3. PROGRAMMAZIONE LINEARE 0/1 *è NP-completa.*

Dimostrazione. La dimostrazione è analoga a quella del Teorema 18.2. Siano dati $n > 0$ e $D = \{d_0, d_1, \ldots, d_{m-1}\}$, con $d = d_0$, per il DOMINO LIMITATO. Reinterpretiamo le condizioni di ricoprimento del DOMINO LIMITATO in termini di un sistema di disequazioni, introducendo le variabili intere 0/1 x_{hij}, $0 \le h \le m-1$, $0 \le i, j \le n-1$, con il seguente significato:

$$x_{hij} = 1 \text{ se e solo se la tessera } d_h \text{ copre la casella } (i, j).$$

Si definiscono le seguenti disequazioni:

1. $x_{000} = 1$ (che "codifica" la copertura di (0,0) con la tessera d);
2. $\sum_{0 \le h \le m-1} x_{hij} = 1$, $0 \le i, j \le n-1$ (queste equazioni asseriscono che esattamente una tessera copre ciascuna casella di $[0, n-1] \times [0, n-1]$);
3. $x_{hij} + \sum_{h'} x_{h'i+1j} \le 1$, $0 \le h \le m-1$, $0 \le i \le n-2$, $0 \le j \le n-1$, per tutte le tessere $d_{h'}$ che non possono essere adiacenti orizzontalmente a d_h, con d_h alla sinistra di $d_{h'}$;
4. $x_{hij} + \sum_{h'} x_{h'ij} + 1 \le 1$, $0 \le h \le m-1$, $0 \le i \le n-1$, $0 \le j \le n-2$, per tutte le tessere $d_{h'}$ che non possono essere adiacenti verticalmente a d_h, con d_h sotto $d_{h'}$.

Poiché una equazione $a = b$ può essere espressa come coppia di disequazioni $a \le b$ e $a \ge b$, il sistema definito in (1)-(4) è un sistema di disequazioni. Tale sistema contiene un numero polinomiale, in n ed m, di variabili 0/1, di disequazioni, e di variabili per disequazione. È facile verificare che il DOMINO LIMITATO ha una copertura "legale" per n e D se e solo se il sistema di disequazioni ha soluzione. La riduzione richiede tempo polinomiale in n ed m. Essendo PROGRAMMAZIONE LINEARE 0/1 $\in \mathbb{NP}$, si è provato che questo problema è NP-completo. □

Esempio 18.7 (DOMINO LIMITATO $\propto$ PROGRAMMAZIONE LINEARE 0/1). Riconsiderando i dati di input del DOMINO LIMITATO mostrati nella Fig. 18.4a), si ottiene il sistema

$$
\begin{array}{rcll}
x_{000} & = & 1 & \text{per la condizione (1),} \\
x_{000} + x_{100} & = & 1 & \\
x_{001} + x_{101} & = & 1 & \\
x_{010} + x_{110} & = & 1 & \\
x_{011} + x_{111} & = & 1 & \text{per le condizioni (2),} \\
x_{000} + x_{010} & \leq & 1 & \\
x_{001} + x_{011} & \leq & 1 & \\
x_{100} + x_{010} & \leq & 1 & \\
x_{101} + x_{011} & \leq & 1 & \\
x_{100} + x_{110} & \leq & 1 & \\
x_{101} + x_{111} & \leq & 1 & \text{per le condizioni (3),} \\
x_{000} + x_{001} & \leq & 1 & \\
x_{010} + x_{011} & \leq & 1 & \\
x_{100} + x_{101} & \leq & 1 & \\
x_{110} + x_{111} & \leq & 1 & \text{per le condizioni (4),}
\end{array}
$$

che è verificato per

$$x_{000} = x_{011} = x_{101} = x_{110} = 1, x_{001} = x_{010} = x_{100} = x_{111} = 0,$$

che corrisponde ancora alla copertura mostrata nella Fig. 18.4a). □

Vediamo un breve elenco di nuovi problemi che, al pari dei precedenti, sono tutti NP-completi.

COPERTURA ESATTA DI INSIEMI: "Dato un insieme X e una famiglia di suoi sottoinsiemi $Y = \{Y_1, \ldots, Y_n\}$, esiste una sottofamiglia F di elementi di Y che partizioni X?"

ABBINAMENTO 3-DIMENSIONALE: "Dato un insieme M, sottoinsieme di $W \times X \times V$, con W, X, V insiemi disgiunti di eguale cardinalità q, esiste un sottoinsieme M' di M, con $|M'| = q$, tale che se (w, x, v) e (w', x', v') sono elementi distinti di M', allora $w \neq w'$, $x \neq x'$ e $v \neq v'$?"

PARTIZIONE: "Dato un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi, esiste un sottoinsieme S di $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?"

SOMMA DI SOTTOINSIEME: "Dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi ed un intero positivo k, esiste un sottoinsieme S di $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i = k$?"

ZAINO: "Dati un insieme $O = \{o_1, \ldots, o_n\}$ di "oggetti", tali che o_i è caratterizzato da un "profitto" p_i e da un "volume" v_i, entrambi interi positivi, e due interi positivi c (la "capacità" dello zaino) e k (l'"obiettivo"), esiste un sottoinsieme S di $\{1, \ldots, n\}$ tale che $\sum_{i \in S} v_i \leq c$ e $\sum_{i \in S} p_i \geq k$?"

SCHEDULING MULTIPROCESSORE: "Dati un insieme $P = \{p_1, \ldots, p_n\}$ di "programmi", tali che p_i richiede un "tempo (intero) di esecuzione" t_i, m "processori" identici, ed un intero d, è possibile eseguire tutti i programmi sui processori in al più d unità di tempo?"

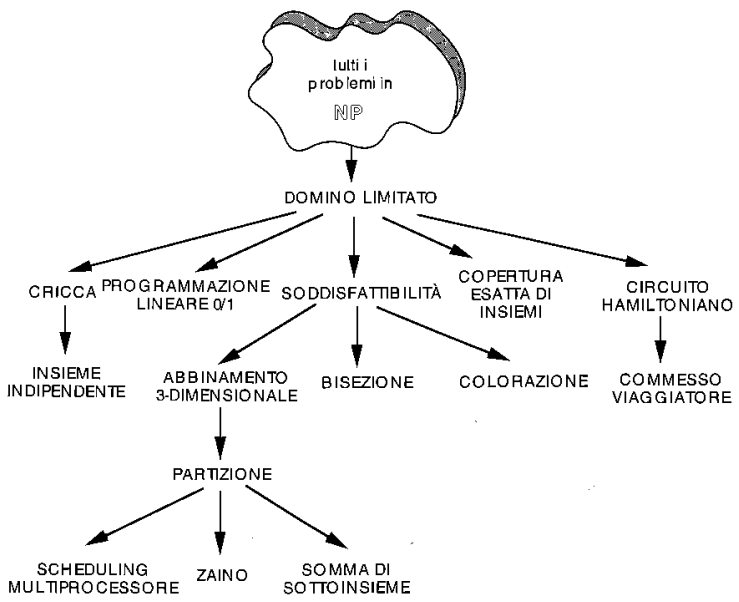


Figura 18.5: Riduzioni tra problemi in $\mathbb{NP}$.

CIRCUITO HAMILTONIANO: "Dato un grafo non orientato G, esiste un circuito che attraversi ogni nodo una e una sola volta?"

INSIEME INDIPENDENTE: "Dati un grafo non orientato G ed un intero k, esiste un sottoinsieme di almeno k nodi mutuamente non connessi da archi?"

BISEZIONE: "Dati un grafo non orientato $G = (N, A)$ ed un intero k, esiste una partizione di N in due sottoinsiemi di $|N|/2$ nodi tale che il numero di archi con un estremo in un sottoinsieme ed un estremo nell'altro non supera k?"

Tra questi problemi, si possono dimostrare tutte le riduzioni illustrate nella Fig. 18.5, dove una riduzione $A \propto B$ è mostrata con una freccia da A verso B. Le riduzioni DOMINO LIMITATO $\propto$ CRICCA, DOMINO LIMITATO $\propto$ SODDISFATTIBILITÀ e DOMINO LIMITATO $\propto$ PROGRAMMAZIONE LINEARE 0/1 sono state appena dimostrate. Vediamo di provarne altre molto più semplici.

Teorema 18.4. CRICCA $\propto$ INSIEME INDIPENDENTE.

Dimostrazione. Dato un grafo non orientato $G = (N, A)$ per il problema CRICCA, si definisca il grafo "complemento" $G^c = (N, A^c)$ di G avente gli stessi nodi, ma in cui sono scambiati archi e "non archi": $[i, j] \in A^c$ se e solo se $[i, j] \notin A$. È immediato verificare che un sottoinsieme S di nodi è una soluzione di CRICCA per il grafo G se e solo è anche una soluzione di INSIEME INDIPENDENTE per G^c. Un esempio è fornito nella Fig. 18.6. □

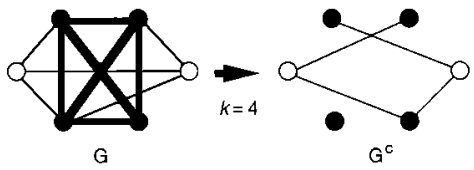


Figura 18.6: Riduzione CRICCA $\propto$ INSIEME INDIPENDENTE. I nodi neri formano una cricca di G ed un insieme indipendente di G^c di cardinalità maggiore o uguale a $k = 4$.

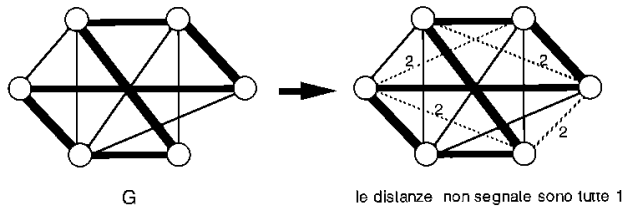


Figura 18.7: Riduzione CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE. Le distanze tra le città sono tutte 1, tranne quelle tratteggiate, che valgono 2 perché non corrispondono ad archi di G. Un circuito Hamiltoniano di G ed il percorso corrispondente per il commesso viaggiatore di lunghezza $k = n = 6$ sono mostrati in grassetto.

Teorema 18.5. CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE.
Dimostrazione. Dato un grafo non orientato $G = (N, A)$, si definisca un insieme di dati per il COMMESSO VIAGGIATORE come segue. Si introducano n città, con $n = |N|$, si ponga $k = n$, e si definisca così la matrice $D = [d_{ij}]$ delle distanze tra le città i e j, per $1 \leq i, j \leq n$:

$$d_{ij} = \begin{cases} 1, & \text{se } [i,j] \in A, \\ 2, & \text{se } [i,j] \notin A. \end{cases}$$

È immediato verificare che esiste un circuito Hamiltoniano in G se e solo se esiste un percorso del COMMESSO VIAGGIATORE con distanza complessiva k. Un esempio è fornito nella Fig. 18.7. □

Teorema 18.6. PARTIZIONE $\propto$ SCHEDULING MULTIPROCESSORE.
Dimostrazione. Dato un insieme $A = \{a_1, \ldots, a_n\}$ per PARTIZIONE, si definisca un insieme $P = \{p_1, \ldots, p_n\}$ di "programmi", tali che p_i richiede tempo $t_i = a_i$, per $1 \leq i \leq n$. Si considerino $m = 2$ "processori" e si ponga $d = (1/2)\sum_{1 \leq i \leq n} a_i$. È

facile vedere che è possibile eseguire tutti i programmi sui due processori in al più d unità di tempo se e solo se esiste una soluzione per PARTIZIONE. □

Teorema 18.7. PARTIZIONE $\propto$ SOMMA DI SOTTOINSIEME.
Dimostrazione. Dato un insieme $A = \{a_1, \ldots, a_n\}$ per PARTIZIONE, si ponga $k = (1/2)\sum_{1 \leq i \leq n} a_i$. □

Teorema 18.8. PARTIZIONE $\propto$ ZAINO.
Dimostrazione. Dato un insieme $A = \{a_1, \ldots, a_n\}$ per PARTIZIONE, si pongano $p_i = v_i = a_i$, $1 \leq i \leq n$, e $k = c = (1/2)\sum_{1 \leq i \leq n} a_i$. □

Lo scopo delle precedenti riduzioni è mostrare come molti problemi, che nascono da contesti diversi, siano in realtà riformulazioni dello stesso problema o sue semplici varianti. In questo caso, è facile dimostrare l'NP-completezza di un problema fornendo una riduzione polinomiale da uno "che gli somigli il più possibile". La semplicità di queste riduzioni non deve però trarre in inganno, poiché molte riduzioni sono, al contrario, piuttosto complicate come, per esempio, SODDISFATTIBILITÀ $\propto$ ABBINAMENTO 3-DIMENSIONALE e DOMINO LIMITATO $\propto$ CIRCUITO HAMILTONIANO, che qui non stiamo a dimostrare.

18.5 Esercizi

Esercizio 18.1 (Dimensione). Si stabilisca se le complessità delle funzioni PIPPO e FOLLIA dell'Esercizio 15.4 siano polinomiali nella dimensione dell'input oppure no.

Le funzioni PIPPO e FOLLIA ricevono in ingresso un intero n. Poiché per codificare n in modo conciso occorrono $O(\log n)$ caratteri, la dimensione d dell'input è $O(\log n)$. La funzione PIPPO richiede $O(\log n) = O(d)$ tempo, e quindi ha complessità polinomiale nella dimensione dell'input. La funzione FOLLIA invece richiede $O(n) = O(2^d)$ tempo, e quindi ha complessità esponenziale nella dimensione dell'input.

Esercizio 18.2 (Counting sort). Si stabilisca se la complessità della procedura COUNTING-SORT dell'Esercizio 11.3 sia polinomiale nella dimensione dell'input oppure no.

Esercizio 18.3 (Coefficiente binomiale). Si stabilisca se la complessità della procedura TARTAGLIA dell'Esempio 14.1 sia polinomiale nella dimensione dell'input oppure no.

Esercizio 18.4 (Copertura con nodi). Dati un grafo non orientato G ed un intero k, il problema della COPERTURA CON NODI chiede se esiste un sottoinsieme S di al più k nodi tale che ciascun arco abbia almeno un estremo in S. Si dimostri che INSIEME INDIPENDENTE $\propto$ COPERTURA CON NODI.

Esercizio 18.5 (Ciclo Hamiltoniano). Dato un grafo orientato G, il problema del CICLO HAMILTONIANO chiede se esiste un ciclo che include ogni nodo di G esattamente una volta. Si dimostri che CIRCUITO HAMILTONIANO $\propto$ CICLO HAMILTONIANO.

Esercizio 18.6 (Cammino Hamiltoniano). Dato un grafo orientato G, il problema del CAMMINO HAMILTONIANO chiede se esiste un cammino che include ogni nodo di G esattamente una volta. Si dimostri che CICLO HAMILTONIANO $\propto$ CAMMINO HAMILTONIANO. Si mostri che la riduzione vale anche se sono dati il nodo di partenza r ed il nodo di arrivo t per il CAMMINO HAMILTONIANO.

Esercizio 18.7 (Catena Hamiltoniana). Dato un grafo non orientato G, il problema della CATENA HAMILTONIANA chiede se esiste una catena che include ogni nodo di G esattamente una volta. Si dimostri che CICLO HAMILTONIANO $\propto$ CATENA HAMILTONIANA (Suggerimento: si modifichi la riduzione dell'Esercizio 18.6, sostituendo ogni nodo del grafo orientato D di partenza con una catena di tre nodi del grafo non orientato G di arrivo, e sistemando opportunamente in G le connessioni relative agli archi di D).

Esercizio 18.8 (Cammino massimo). Dati un grafo orientato G, due nodi r e t, ed un intero k, il problema del CAMMINO MASSIMO chiede se esiste un cammino semplice da r a t che include almeno k archi. Si dimostri che il problema è NP-completo. Si dimostri inoltre che il problema diventa polinomiale qualora G sia aciclico.

Esercizio 18.9 (Albero di copertura limitato). Dati un grafo non orientato G ed un intero k, il problema dell'ALBERO DI COPERTURA LIMITATO chiede se esiste un albero di copertura T per G tale che in ogni nodo di T siano incidenti al più k archi. Si dimostri che il problema è NP-completo.

Capitolo 19

Gerarchia di complessità

I problemi NP-completi sono in un certo senso i più difficili problemi in $\mathbb{NP}$. Benché non sia stata dimostrata alcuna limitazione inferiore superpolinomiale alla complessità di problemi NP-completi, il fatto che per decenni nessuno sia riuscito a trovare neanche un algoritmo polinomiale deterministico per un solo problema NP-completo fa supporre che questi problemi siano inerentemente intrattabili. Non si deve però pensare che i problemi NP-completi siano i problemi più difficili. Sono noti molti altri problemi ben più difficili di quelli NP-completi, per alcuni dei quali sono state provate limitazioni inferiori superpolinomiali o addirittura maggiori!

Per trattare e classificare in modo omogeneo i problemi più disparati, assumeremo in questo capitolo che i problemi siano formulati in termini di "riconoscimento di linguaggi". Sia V l'insieme di caratteri (o alfabeto) utilizzato per codificare i dati d'ingresso di un problema decisionale A. Il problema stesso può essere messo in corrispondenza con il sottoinsieme L (linguaggio) di tutte le sequenze finite di simboli (stringhe) su V tale che:

$$L = \{x\colon x \text{ è una stringa su } V \text{ e codifica i dati di } A \text{ che danno risposta SI}\}.$$

La dimensione dell'input è individuata dal numero di caratteri che appaiono nella stringa x.

Esempio 19.1 (Linguaggio per CRICCA). Sia V l'alfabeto utilizzato per codificare il grafo G e l'intero k. Il problema della CRICCA corrisponde al linguaggio $L_{\text{CRICCA}} = \{x\colon x$ è una stringa su V che codifica un grafo G avente una cricca di almeno k nodi$\}$. Per esempio, si assuma di utilizzare $V = \{|, 1, 0, *\}$, e si considerino i dati di ingresso per CRICCA: $G = (N, A)$, $N = \{1, 2, 3, 4\}$, $A = \{[1, 2], [1, 3], [1, 4], [3, 4]\}$ e $k = 3$ (si veda la Fig 17.4). Questi dati possono essere codificati con la stringa $x = 0111|1000|1001|1010 * 11$. Questa stringa rappresenta le righe della matrice di adiacenza nodi-nodi di G, separate tra loro da "|", seguite a loro volta da "$*$" che le separa dalla successiva codifica binaria di k. La dimensione dell'input è $O(n^2)$. □

I vantaggi che derivano dal considerare il riconoscimento di linguaggi sono molteplici. In primo luogo, si possono applicare tecniche di dimostrazione ben consolidate derivanti dalla teoria della computabilità e dei linguaggi formali. Inoltre, è più agevole provare risultati di complessità che non riguardino esclusivamente il tempo, ma anche altre "misure", come per esempio lo spazio di memoria. In particolare, considerando soltanto risposte SI o NO, il costo (sia in termini di spazio che di tempo) per produrre il risultato viene ad essere costante e ci si può concentrare maggiormente sul costo dovuto alla computazione, anziché sull'ingresso e l'uscita dei dati. Questa

proprietà è molto importante, perché per molti problemi accade spesso che il costo per produrre la soluzione sia dominante rispetto a quello per calcolare la soluzione.

19.1 Problemi NP-intermedi

La prima domanda che viene spontanea consiste nel chiedersi, qualora $\mathbb{P}$ sia propriamente contenuta in $\mathbb{NP}$, se nell'insieme differenza $\mathbb{NP} - \mathbb{P}$ ci siano tutti e soli i problemi NP-completi. Enunciamo senza dimostrazione un risultato fondamentale provato da Ladner (1975):

Teorema 19.1. *Se $\mathbb{P} \neq \mathbb{NP}$, allora esiste un problema (linguaggio) $A \in \mathbb{NP} - \mathbb{P}$ tale che A non è NP-completo.* □

Detto in altri termini, se $\mathbb{P} \neq \mathbb{NP}$, allora esistono problemi in $\mathbb{NP}$ che né stanno in $\mathbb{P}$ né sono NP-completi. Tali problemi sono detti NP-intermedi. In realtà Ladner ha dimostrato proprietà ancora più forti. Infatti, se $\mathbb{P} \neq \mathbb{NP}$, allora devono esistere coppie di problemi NP-intermedi A e B tali che né $A \propto B$ né $B \propto A$. In altri termini, ci devono essere problemi in $\mathbb{NP}$ non confrontabili secondo la relazione $\propto$ qualora $\mathbb{P} \neq \mathbb{NP}$. Un problema candidato per essere NP-intermedio è il seguente:

ISOMORFISMO TRA GRAFI: "Dati due grafi non orientati $G = (N, A)$ ed $H = (N, E)$ con lo stesso insieme di nodi, sono isomorfi, ovvero esiste una corrispondenza biunivoca f: $N \rightarrow N$ tale che $[u, v] \in A$ se e solo se $[f(u), f(v)] \in E$?"

Coloro che hanno invano cercato di dimostrare che l'ISOMORFISMO TRA GRAFI è NP-completo hanno notato come questo problema sia molto più "rigido" rispetto ai tipici problemi NP-completi. I problemi NP-completi, richiedono un certo grado di "ridondanza", in quanto se la struttura desiderata (sottoinsieme, permutazione, ecc.) esiste, allora di solito esiste anche apportando lievi modifiche all'input. Invece, nel caso dell'ISOMORFISMO TRA GRAFI, un qualsiasi cambiamento nel grafo G si deve riflettere in un corrispondente cambiamento in H, altrimenti f non può essere più un isomorfismo. Questa rigidità per ora non è servita a dimostrare che il problema sia in $\mathbb{P}$, benché molti suoi casi speciali siano risolvibili con algoritmi polinomiali deterministici. Pertanto, l'ISOMORFISMO TRA GRAFI potrebbe essere un problema NP-intermedio.

19.2 La classe co-$\mathbb{NP}$

Vediamo ora cosa accade passando da un problema decisionale A al suo "complemento" A^c, cioè a quel problema decisionale che ha risposta SI dove A ha risposta NO, e risposta NO dove A ha risposta SI. Se $A \in \mathbb{P}$, allora anche $A^c \in \mathbb{P}$ e l'algoritmo risolutivo avrà scambiate le istruzioni **success** e **failure**. Se invece $A \notin \mathbb{P}$ e $A \in \mathbb{NP}$, allora in generale non si può dire niente. Non basta infatti scambiare tra loro le istruzioni **success** e **failure**, perché l'algoritmo non deterministico risultante per A^c darebbe risposta SI se quello per A eseguisse almeno una **failure**. Invece, A^c deve dare risposta SI se e solo se A ha risposta NO, ovvero se tutte le possibili computazioni dell'algoritmo non deterministico per A terminano su **failure**. A^c è quindi non più facile di A (anzi, presumibilmente A^c è più difficile di A, assumendo $\mathbb{P} \neq \mathbb{NP}$).

Ad esempio, non si sa se il complemento di SODDISFATTIBILITÀ sia in $\mathbb{NP}$ oppure no:

FALSITÀ: "Data una formula booleana in forma normale congiuntiva, è tale formula falsa per tutte le possibili assegnazioni di valori di verità vero/falso alle variabili booleane?"

In generale, tutti i linguaggi (problemi) in $\mathbb{NP}$ si possono esprimere come segue:

$$L = \{x : \exists y\, P(x, y) = \mathbf{true}\},$$

dove $\exists$ ("esiste") è il quantificatore esistenziale, x appartiene allo spazio degli input del problema in esame, y appartiene invece allo spazio delle soluzioni del problema, e P è una funzione booleana (detta "predicato") calcolabile in tempo polinomiale con un algoritmo deterministico (il "certificato").

I linguaggi complementati, invece, possono essere espressi nel modo seguente:

$$L^c = \{x : \forall y\, P(x, y) = \mathbf{false}\},$$

dove $\forall$ ("per ogni") è il quantificatore universale. Infatti, "**not** $\exists\, y\, P(x, y) = \mathbf{true}$" $\equiv$ "$\forall\, y$ **not** $P(x, y) = \mathbf{true}$" $\equiv$ "$\forall\, y\, P(x, y) = \mathbf{false}$". È intuitivo immaginare come il passaggio da un quantificatore esistenziale ad uno universale complichi notevolmente il problema (qualora sia $\mathbb{P} \neq \mathbb{NP}$).

La classe ottenuta complementando i problemi in $\mathbb{NP}$ è detta co-$\mathbb{NP}$. Vale il seguente risultato.

Teorema 19.2. *Se* co-$\mathbb{NP} \neq \mathbb{NP}$, *allora* $\mathbb{P} \neq \mathbb{NP}$.

Dimostrazione. Siano per assurdo co-$\mathbb{NP} \neq \mathbb{NP}$ e $\mathbb{P} = \mathbb{NP}$. Essendo $\mathbb{P} =$ co-$\mathbb{P}$, si ottiene la contraddizione che $\mathbb{NP} = \mathbb{P} =$ co-$\mathbb{P} =$ co-$\mathbb{NP}$. □

Poiché il Teorema 19.2 dà una condizione solo sufficiente, ne segue che se $\mathbb{P} = \mathbb{NP}$ allora anche co-$\mathbb{NP} = \mathbb{P}$, ma se $\mathbb{P} \neq \mathbb{NP}$ allora potrebbe essere co-$\mathbb{NP} = \mathbb{NP}$ (anche se quest'ultimo caso pare poco plausibile).

In modo analogo alla definizione di problema NP-completo, si può definire un problema co-NP-completo come un problema tale che ogni problema in co-$\mathbb{NP}$ sia riducibile ad esso con una trasformazione polinomiale deterministica. È facile verificare che se A è NP-completo, allora il suo complemento A^c è co-NP-completo. Il seguente risultato mostra come un singolo controesempio possa smantellare l'ipotesi che co-$\mathbb{NP} \neq \mathbb{NP}$:

Teorema 19.3. *Se esiste un problema NP-completo A tale che il suo complemento A^c sta in $\mathbb{NP}$, allora* co-$\mathbb{NP} = \mathbb{NP}$.

Dimostrazione. Poiché A è NP-completo, A^c è co-NP-completo. Pertanto, ogni B^c in co-$\mathbb{NP}$ è tale che $B^c \propto A^c$. Ma essendo A^c in $\mathbb{NP}$, allora anche B^c è in $\mathbb{NP}$, e quindi $\mathbb{NP} \supseteq$ co-$\mathbb{NP}$. Inoltre, ogni B che è in $\mathbb{NP}$ è tale che $B \propto A$, perché A è NP-completo. Essendo A^c in $\mathbb{NP}$, si ha che $A = (A^c)^c$ è in co-$\mathbb{NP}$. Ma allora anche B è in co-$\mathbb{NP}$ e quindi co-$\mathbb{NP} \supseteq \mathbb{NP}$. Combinando i due risultati, segue che co-$\mathbb{NP} = \mathbb{NP}$. □

Una conseguenza immediata del Teorema 19.3 è che se esiste un problema NP-completo A che appartiene sia ad $\mathbb{NP}$ che a co-$\mathbb{NP}$, allora co-$\mathbb{NP} = \mathbb{NP}$. Allo stato attuale delle conoscenze, non è noto alcun problema NP-completo $A \in \mathbb{NP} \cap$ co-$\mathbb{NP}$. Sono noti invece problemi che appartengono ad $\mathbb{NP} \cap$ co-$\mathbb{NP}$ per i quali non si conoscono né la loro NP-completezza, né la loro appartenenza a $\mathbb{P}$, che sono quindi candidati ad essere NP-intermedi. Un esempio illustre è il seguente:

PRIMALITÀ: "Dato un numero n, è tale numero primo, cioè non esiste alcun numero diverso da 1 e da n stesso che lo divida esattamente?"

Il problema può essere risolto con la seguente procedura deterministica:

```
function PRIMO(n: integer): boolean;
    var j: integer; b: boolean;
    begin
        b := true; j := 2;
        while (b) and (j < n) do
            if n mod j = 0
                then b := false
                else j := j + 1;
        PRIMO := b
    end;
```

La funzione PRIMO prova a dividere n per ciascun numero j compreso tra 2 ed $n-1$, e restituisce come risultato "vero" se non è stato trovato alcun j tale che n **mod** $j = 0$, cioè tale che il resto della divisione di n per j sia zero. La sua complessità è $O(n)$, che non è polinomiale nella dimensione del problema. Infatti, poiché un numero intero n codificato in modo "conciso" (p.e. in base $b = 2$) necessita di $d = \log n + 1$ bit, la dimensione del problema è d. La complessità della funzione PRIMO è pertanto $O(2^d)$, cioè esponenziale nella dimensione del problema (cfr. Esercizio 18.1).

Utilizzando il teorema di Fermat, è possibile però dimostrare che PRIMALITÀ $\in \mathbb{NP}$ (cfr. Esercizio 18.1). D'altronde, il complemento della PRIMALITÀ è il seguente:

COMPOSTEZZA: "Dato un numero n, è tale numero composto, cioè esiste un numero diverso da 1 e da n stesso che lo divida esattamente?"

È immediato verificare che COMPOSTEZZA $\in \mathbb{NP}$:

```
procedure COMPOSTO(n: integer);
    var h, k: integer;
    begin
        h := choice({2, 3, ... , n − 1});
        k := choice({2, 3, ... , n − 1});
        if n = h * k then success else failure
    end;
```

Per il Teorema 19.3, pare poco plausibile che PRIMALITÀ sia NP-completa. In effetti, è stato fornito da Miller (1976) un algoritmo deterministico per decidere se un numero è primo oppure composto, che ha complessità polinomiale nel caso che valga l'"ipotesi estesa di Riemann". Inoltre, esistono algoritmi polinomiali "probabilistici" che risolvono la PRIMALITÀ con probabilità arbitrariamente vicina ad 1 (cfr. § 20).

In pratica, molti ritengono che mostrare che $A \in \mathbb{NP}$ ed $A^c \in$ co-$\mathbb{NP}$ sia una indicazione che A possa essere risolvibile con un algoritmo polinomiale deterministico, anche se ancora non si conosce se $\mathbb{P} = \mathbb{NP} \cap$ co-$\mathbb{NP}$ oppure no.

Infine, sono noti alcuni problemi decisionali correlati a problemi di ottimizzazione che, pur essendo formulabili come l'intersezione di un problema in $\mathbb{NP}$ ed uno in co-$\mathbb{NP}$, non possono stare né in $\mathbb{NP}$ né in co-$\mathbb{NP}$, a meno che non risulti $\mathbb{NP} =$ co-$\mathbb{NP}$. Un esempio è dato da

DIMENSIONE CRICCA MASSIMA: "Dati un grafo non orientato G ed un intero k, la dimensione della cricca massima è uguale a k?",

che è dato dall'intersezione di CRICCA (dove si chiede se esiste una cricca con almeno k nodi) e del suo stesso complemento (ove si chiede se tutte le cricche hanno al più k nodi).

19.3 Gerarchia polinomiale

È possibile definire una gerarchia infinita di classi di complessità, dove ogni nuova classe è definita per mezzo delle altre classi già definite ed apparentemente contiene problemi di difficoltà sempre crescente. Consideriamo ad esempio il seguente problema:

FORMULA BOOLEANA k-QUANTIFICATA (k-QBF): "Data una formula booleana preceduta da una sequenza di k quantificatori esistenziali ed universali alternati, tale che il primo quantificatore è esistenziale e k è costante, è tale formula vera?"

Evidentemente, per $k = 1$ il problema 1-QBF altro non è che la SODDISFATTIBILITÀ, ma all'aumentare di k pare aumentare anche la difficoltà per risolvere k-QBF.

In generale, per $k \geq 0$ si può definire la classe Σ_k^p come la classe contenente tutti i linguaggi seguenti

$$L = \{x : \exists y_1\ \forall y_2\ \exists y_3\ \forall y_4\ \cdots Q y_k\ P(x, y_1, \ldots, y_k) = \mathbf{true}\},$$

dove $Q = \forall$ se k è pari, mentre $Q = \exists$ se k è dispari, x appartiene allo spazio degli input del problema, $y_1, \ldots, y_k$ appartengono allo spazio delle soluzioni del problema, e P è un predicato calcolabile con un algoritmo deterministico polinomiale (cioè $P \in \mathbb{P}$). Analogamente, si può definire la classe Π_k^p come la classe co-Σ_k^p, cioè quella contenente i linguaggi seguenti

$$L = \{x : \forall y_1\ \exists y_2\ \forall y_3\ \exists y_4\ \cdots Q y_k\ P(x, y_1, \ldots, y_k) = \mathbf{false}\},$$

dove $Q = \exists$ se k è pari, mentre $Q = \forall$ se k è dispari, e x, $y_1, \ldots, y_k$, e P hanno le stesse proprietà enunciate precedentemente.

Chiaramente, $\Sigma_0^p = \Pi_0^p = \mathbb{P}$, mentre $\Sigma_1^p = \mathbb{NP}$ e $\Pi_1^p =$ co-$\mathbb{NP}$. Le classi Σ_k^p e Π_k^p formano la cosiddetta gerarchia polinomiale, nella quale valgono le seguenti proprietà di inclusione.

Teorema 19.4. *Per ogni $k \geq 0$, $\Sigma_k^p \cup \Pi_k^p \subseteq \Sigma_{k+1}^p \cap \Pi_{k+1}^p$.* □

Allo stato attuale delle conoscenze non si sa né se $\Sigma_k^p \neq \Sigma_{k+1}^p$ per ogni $k \geq 0$, né se $\Sigma_k^p \neq \Pi_k^p$ per ogni $k \geq 1$. Inoltre, è stato dimostrato che k-QBF è Σ_k^p-completo per ogni $k \geq 1$, e quindi k-QBF è il più "difficile" problema in Σ_k^p. Una proprietà importante è che se una classe nella gerarchia è propriamente inclusa dentro un'altra, allora risulta $\mathbb{P} \neq \mathbb{NP}$. Tale proprietà deriva dal seguente teorema, che enunciamo senza dimostrazione.

Teorema 19.5. *Se esiste un qualsiasi $k \geq 1$ tale che $\Sigma_k^p = \Pi_k^p$, allora $\Sigma_h^p = \Pi_h^p = \Sigma_k^p$ per ogni $h \geq k$.* □

In altri termini, la gerarchia polinomiale potrebbe "collassare". In tal caso, sarebbe interessante conoscere quale sia il minimo k per il quale risulti $\Sigma_k^p = \Pi_k^p$. Infatti, potrebbe anche accadere che la gerarchia collassi in Σ_2^p pur essendo $\mathbb{P} \neq \mathbb{NP}$, anche se tale eventualità sembra poco plausibile.

19.4 La classe $\mathbb{P}$-SPAZIO

Finora, la complessità di un algoritmo è stata valutata considerandone esclusivamente il tempo di computazione e non lo spazio di memoria. Ciò è in parte dovuto sia al fatto che la maggior parte degli algoritmi per problemi "facili" (cioè in $\mathbb{P}$) non utilizza quantità enormi di memoria, sia al fatto che le attuali memorie in commercio sono molto più capienti e molto meno costose rispetto ad alcuni anni fa. Ciò non ostante, è interessante vedere come siano caratterizzate anche le classi di complessità spaziale, e come esse siano correlate alle classi di complessità temporale. Innanzitutto, è immediato osservare che se il tempo è limitato da una certa quantità $T(n)$, allora anche lo spazio è limitato dalla stessa quantità, perché ogni istruzione elementare sia di un algoritmo deterministico sia di uno non deterministico utilizza un numero costante di bit di memoria. Viceversa, se lo spazio è limitato da una quantità $S(n)$, allora è possibile dimostrare che il tempo è limitato da $2^{cS(n)}$, dove c è una costante opportuna, poiché altrimenti l'algoritmo ciclerebbe all'infinito.

La classe dei problemi risolubili con algoritmi deterministici che utilizzano una quantità di memoria polinomiale nella dimensione dell'input è detta $\mathbb{P}$-SPAZIO. Analogamente, la classe dei problemi risolubili con algoritmi non deterministici che utilizzano una quantità di memoria polinomiale nella dimensione dell'input è detta $\mathbb{NP}$-SPAZIO. La proprietà più curiosa è che $\mathbb{P}$-SPAZIO $=$ $\mathbb{NP}$-SPAZIO, cioè il non determinismo non è più potente del determinismo rispetto alla memoria polinomiale, al contrario di ciò che pare avvenire rispetto al tempo polinomiale (a meno che $\mathbb{P} = \mathbb{NP}$). Per quanto detto precedentemente, $\mathbb{P} \subseteq \mathbb{P}$-SPAZIO, perché lo spazio utilizzato (cioè il numero di celle di memoria impiegate) non può superare il tempo di computazione (cioè il numero di istruzioni elementari eseguite). Finora non è noto se $\mathbb{P}$ sia propriamente contenuta in $\mathbb{P}$-SPAZIO oppure no, ma sono noti problemi risolubili con memoria polinomiale ma non in tempo polinomiale. Un illustre rappresentante della classe $\mathbb{P}$-SPAZIO è:

FORMULA BOOLEANA QUANTIFICATA (QBF): "Data una formula booleana preceduta da una sequenza arbitrariamente lunga di quantificatori esistenziali ed universali alternati, è tale formula vera?"

Si noti che, al contrario di ciò che accade per i problemi k-QBF della gerarchia polinomiale, il numero k di alternanze di QBF non è limitato da una costante, bensì può crescere arbitrariamente. Che QBF appartenga a $\mathbb{P}$-SPAZIO deriva dal fatto che si può verificare se la formula booleana è vera provando tutti gli assegnamenti di valori di verità per tutte le n variabili booleane e valutando la formula per ognuno di essi. Tale procedimento richiede spazio di memoria polinomiale in n per memorizzare l'attuale assegnamento e valutare la formula, anche se è necessario un tempo esponenziale per esaminare tutti i 2^n assegnamenti. Come nel caso della gerarchia polinomiale, però, QBF è il più difficile problema in $\mathbb{P}$-SPAZIO. Infatti è stato dimostrato che QBF è P-SPAZIO-completo con una versione analoga del Teorema di Cook-Levin, dove ogni problema in $\mathbb{P}$-SPAZIO si riduce a QBF con una trasformazione che richiede spazio polinomiale anziché tempo polinomiale.

Per molti giochi, come la dama e gli scacchi, giocati su scacchiera di dimensione arbitraria $n \times n$, con n dato di ingresso, il problema di decidere se esiste una strategia vincente per il giocatore che muove per primo è in $\mathbb{P}$-SPAZIO, ma non in $\mathbb{NP}$ (assumendo $\mathbb{P}$-SPAZIO $\neq \mathbb{NP}$). Questo problema è invero strettamente correlato a QBF. Infatti, detto A il giocatore che muove per primo, e B l'altro giocatore, si ha:

$\exists$ una mossa per A tale che

$\forall$ mossa di B
$\exists$ una mossa per A tale che

$\vdots$

$\forall$ mossa di B
$\exists$ una mossa per A tale che A vince?

Questa rappresentazione è comune a tutti i linguaggi in $\mathbb{P}$-SPAZIO, che possono essere definiti come segue:

$$L = \{x : \exists\, k\; Q_1 y_1\; Q_2 y_2\; \cdots Q_k y_k\; P(x, y_1, \ldots, y_k) = \mathbf{true}\},$$

dove i Q_i, $1 \leq i \leq k$, sono quantificatori esistenziali e universali alternati (cioè se $Q_i = \exists$ allora $Q_{i+1} = \forall$ mentre se $Q_i = \forall$ allora $Q_{i+1} = \exists$), x appartiene allo spazio degli input del problema, $y_1, \ldots, y_k$ appartengono allo spazio delle soluzioni del problema, e P è al solito un predicato calcolabile con un algoritmo deterministico polinomiale.

È possibile inoltre dimostrare che $\Sigma_k^p \cup \Pi_k^p \subseteq \mathbb{P}$-SPAZIO, che ha come conseguenza che se $\mathbb{P} \neq \mathbb{NP}$ allora anche $\mathbb{P} \neq \mathbb{P}$-SPAZIO (comunque, potrebbero anche valere contemporaneamente $\mathbb{P} = \mathbb{NP}$ e $\mathbb{P} \neq \mathbb{P}$-SPAZIO).

19.5 Le classi $\mathbb{EXP}$ e $\mathbb{NEXP}$

La classe $\mathbb{EXP}$ consiste di quei problemi risolubili con algoritmi deterministici in tempo esponenziale $O(2^{n^k})$, con k costante, mentre la classe $\mathbb{NEXP}$ comprende quei problemi risolubili con algoritmi non deterministici in tempo esponenziale. Sono state provate le seguenti relazioni:

Teorema 19.6. *$\mathbb{P}$ è propriamente contenuta in $\mathbb{EXP}$, ed $\mathbb{NP}$ è propriamente contenuta in $\mathbb{NEXP}$.* □

$\mathbb{P}$-SPAZIO $\subseteq \mathbb{EXP}$, perché lo spazio utilizzato da una computazione non può superare il tempo della computazione stessa, ma si ignora se $\mathbb{P}$-SPAZIO $\neq \mathbb{EXP}$. Ovviamente, $\mathbb{EXP} \subseteq \mathbb{NEXP}$ per definizione. Purtroppo, come si ignora se $\mathbb{P} \neq \mathbb{NP}$, allo stesso modo si ignora anche se $\mathbb{EXP} \neq \mathbb{NEXP}$. Comunque, è stato dimostrato il seguente risultato.

Teorema 19.7. *Se $\mathbb{EXP} \neq \mathbb{NEXP}$, allora $\mathbb{P} \neq \mathbb{NP}$.* □

Il rompicapo delle TORRI DI HANOI [cfr. § 3] è un problema in $\mathbb{EXP}$, ma il più famoso problema della classe è senza dubbio il gioco seguente:

BLOCCO STRADALE: "Siano dati due giocatori (Ferrari e McLaren) ed un grafo non orientato i cui archi rappresentano strade e i cui nodi rappresentano incroci tra strade. Ogni arco è colorato con uno tra tre colori possibili. Alcuni nodi sono marcati "vince Ferrari" ed altri nodi sono marcati "vince McLaren". Ogni giocatore ha una "scuderia" di auto, ciascuna delle quali inizialmente occupa un nodo. Ciascun giocatore sposta a turno un'auto della sua scuderia da un nodo ad un altro percorrendo una catena di archi tutti colorati con lo stesso colore, purché non incontri nessun'altra auto lungo i nodi del percorso. Vince il primo giocatore che raggiunge un nodo marcato con la propria vittoria. Esiste una strategia vincente per il giocatore che muove per primo?"

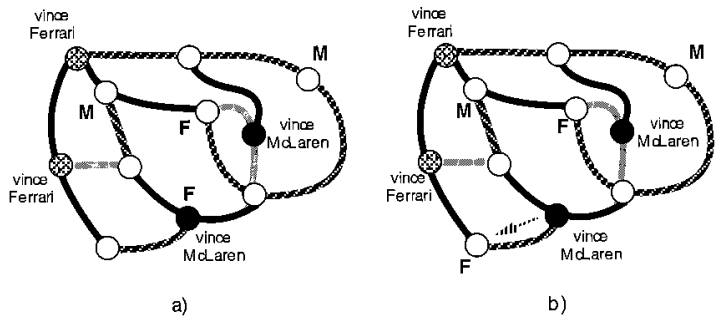


Figura 19.1: BLOCCO STRADALE. a) Situazione iniziale. b) Mossa iniziale della Ferrari: comunque muova la McLaren, la Ferrari può vincere!

Esempio 19.2 (BLOCCO STRADALE). La Fig. 19.1a) illustra un particolare grafo per il BLOCCO STRADALE avente 11 nodi (incroci), 16 archi (strade), 2 incroci marcati "vince Ferrari", 2 marcati "vince McLaren", e 2 auto per giocatore, indicate con F e M. È possibile verificare che la Ferrari, muovendo per prima, ha una strategia vincente, qualsiasi siano le successive mosse della McLaren! Infatti è possibile spostare la macchina F che si trova più in basso percorrendo l'arco tratteggiato, come mostrato nella Fig. 19.1b). Si nota subito che a questo punto la McLaren non può vincere in una sola mossa, perché non c'è una catena formata da tutti archi dello stesso colore che unisca un nodo occupato da una macchina M ed un nodo "vince McLaren". Pertanto, la McLaren deve tentare di sbarrare la strada alle Ferrari. Se la McLaren muove la macchina M più a sinistra, può occupare o il nodo raggiungibile con l'arco tratteggiato oppure uno dei due nodi "vince Ferrari" raggiungibili con uno o due archi della catena nera, mentre negli altri due nodi della catena nera non ci può andare perché occupati da macchine F. Ma allora la Ferrari può muovere una delle due macchine F lungo la catena nera e raggiungere l'altro nodo "vince Ferrari" rimasto scoperto. Se invece la McLaren muove la macchina M più a destra, può percorrere solo la catena tratteggiata e occupare uno dei tre nodi raggiungibili con essa, mentre nel quarto nodo non ci può andare perché occupato da una macchina F. Anche se M occupa il nodo "vince Ferrari" più in alto, lascia comunque la strada sgombra alla macchina F in basso. Pertanto, la Ferrari ha una strategia vincente. □

Per i problemi delle TORRI DI HANOI e del BLOCCO STRADALE sono state dimostrate limitazioni inferiori della complessità temporale che sono esponenziali nel numero n di dischi e di nodi, rispettivamente. Pertanto, per il BLOCCO STRADALE, non solo è difficile determinare l'esistenza di una strategia vincente, ma è anche difficile convincersi che una data strategia sia effettivamente vincente. Infatti, i problemi in $\mathbb{EXP}$ e $\mathbb{NEXP}$ non ammettono certificati polinomiali, bensì esponenziali! Inoltre, è stato dimostrato che ogni problema in $\mathbb{EXP}$ si riduce in tempo esponenziale al BLOCCO STRADALE, che quindi è EXP-completo.

19.6 Le classi $\mathbb{EXP}$-$\mathbb{SPAZIO}$ ed $\mathbb{E}$

$\mathbb{EXP}$-$\mathbb{SPAZIO}$ è la classe dei problemi risolubili con algoritmi deterministici con una quantità di memoria esponenziale $O(2^{n^k})$, con k costante. Invece, $\mathbb{E}$ è la classe dei problemi risolubili con algoritmi deterministici in tempo esponenziale, esponenziale doppio, triplo, ... , k-uplo, ... per $k = 0, 1, 2, \ldots$, cioè in tempo

$$\bigcup_{k \geq 0} O\left(2^{2^{\cdot^{\cdot^{2^n}}}}\right)$$

dove l'esponenziale è ripetuto k volte. Valgono le seguenti inclusioni:

Teorema 19.8. *$\mathbb{P}$-$\mathbb{SPAZIO}$ è propriamente contenuta in $\mathbb{EXP}$-$\mathbb{SPAZIO}$, ed $\mathbb{EXP}$-$\mathbb{SPAZIO}$ è propriamente contenuta in $\mathbb{E}$.* □

È stato dimostrato che la classe $\mathbb{E}$ coincide con la classe delle funzioni "elementari", cioè quelle ottenibili per composizione di somma, sottrazione, prodotto, divisione, esponenziale e logaritmo. Tale classe comprende tutte le funzioni che si usano in pratica, incluse le funzioni trigonometriche (qualora se ne consideri l'estensione al campo complesso).

Un esempio di problema per cui è stata dimostrata una limitazione inferiore della complessità in tempo che è un esponenziale *doppio* è il seguente:

FORMULA ARITMETICA DI PRESBURGER: "Data una formula con quantificatori universali ed esistenziali, costanti e variabili che possono assumere valori interi positivi, operatori logici **not**, **and**, **or**, l'operatore "+" di somma tra interi, e l'operatore "=" di uguaglianza, è l'espressione vera?"

Neanche i problemi in $\mathbb{E}$ sono i più difficili, giacché ci sono problemi che non sono neanche elementari! Per esempio, un problema che non ammette alcun algoritmo di complessità esponenziale k-upla, qualsiasi sia k, è quello di determinare la verità di una FORMULA WS1S. Tale problema è simile alla FORMULA ARITMETICA DI PRESBURGER, in cui le variabili possono indicare anche insiemi di interi e compare l'operatore "$\in$" di appartenenza ad un insieme. Per esempio, la FORMULA WS1S comprende enunciati del tipo: "In ogni insieme non vuoto di numeri c'è un numero minore di tutti gli altri". Tali enunciati non si risolvono semplicemente sostituendo tutti i valori possibili alle variabili, perché tali valori sono in quantità infinita, ma utilizzando manipolazioni algebriche oppure l'induzione matematica. Benché un noto teorema provato da Büchi negli anni '60 garantisca in linea di principio l'esistenza di un algoritmo per risolvere la FORMULA WS1S, il summenzionato risultato di complessità dimostrato da Meyer nel 1972 ne stabilisce in pratica l'insolubilità.

Per concludere, il diagramma di contenimento tra alcune delle classi di complessità trattate in questo capitolo è illustrato nella Fig. 19.2. Allo stato attuale delle conoscenze si ignora se tutte le inclusioni illustrate siano proprie oppure no. Le uniche dimostrate finora sono $\mathbb{EXP} \supset \mathbb{P}$, $\mathbb{NEXP} \supset \mathbb{NP}$ ed $\mathbb{E} \supset \mathbb{EXP}$-$\mathbb{SPAZIO} \supset \mathbb{P}$-$\mathbb{SPAZIO}$.

19.7 Esercizi

Esercizio 19.1 (PRIMALITÀ). Si dimostri che PRIMALITÀ $\in \mathbb{NP}$, utilizzando il teorema di Fermat: "Un numero $n > 2$ è primo se e solo se esiste un intero b compreso

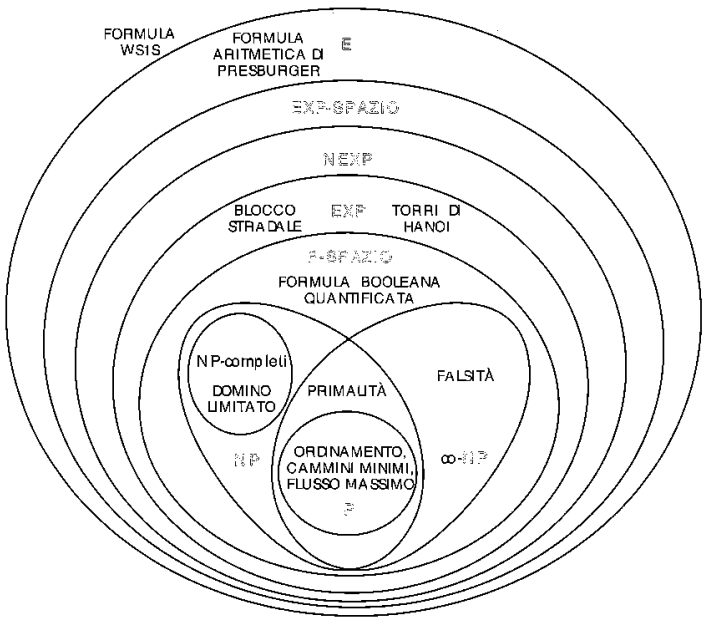


Figura 19.2: Presunte relazioni di contenimento fra classi di complessità.

tra 2 ed $n-1$ tale che b^{n-1} **mod** $n = 1$ e b^i **mod** $n \neq 1$ per ogni i compreso tra 1 ed $n-2$".

Esercizio 19.2. Si dimostri che se $\Sigma_k^p \subseteq \Pi_k^p$ allora $\Sigma_k^p = \Pi_k^p$.

Esercizio 19.3. Si dimostri che ciascuna classe della gerarchia polinomiale è chiusa rispetto all'unione ed all'intersezione, dove il linguaggio unione di due linguaggi L' ed L'' è formato da tutte le stringhe che appartengono ad L' oppure L'', mentre il linguaggio intersezione da tutte quelle che appartengono ad L' ed L''.

Esercizio 19.4. Si dimostri che, per ogni k, Σ_k^p è chiusa rispetto alla riducibilità polinomiale.

Capitolo 20

Algoritmi probabilistici

Nel progetto e nell'analisi di algoritmi, sono stati considerati finora algoritmi deterministici e non deterministici. Gli algoritmi deterministici sono quelli usualmente eseguibili con un calcolatore elettronico e servono a risolvere in pratica i problemi di tutti i giorni. Gli algoritmi non deterministici, invece, risultano utili per studiare la complessità teorica di problemi computazionali, ma devono poi essere ricondotti ad algoritmi enumerativi deterministici per essere eseguiti su un calcolatore reale. D'altro canto, nell'analisi di algoritmi deterministici, è stata talvolta considerata la complessità nel caso medio, ove il tempo di calcolo è stato valutato mediando su tutti i possibili dati di ingresso, dopo aver individuato (o per lo più assunto arbitrariamente) una distribuzione di probabilità in accordo alla quale i dati di ingresso possono presentarsi. Esiste tuttavia un altro genere di algoritmi, che sono in un certo senso intermedi tra quelli deterministici e quelli non deterministici, e per i quali il calcolo delle probabilità è applicato non tanto ai dati di ingresso, quanto piuttosto ai dati di uscita: gli algoritmi probabilistici (detti anche algoritmi randomizzati).

In un algoritmo probabilistico, la computazione dipende dai valori prodotti da un generatore di numeri casuali. Si assuma di avere a disposizione una funzione Pascal RANDOM(h, k), che quando è richiamata restituisce con la stessa probabilità un qualsiasi intero generato casualmente tra h e k.[1]

Esempio 20.1 (Testa o croce?). RANDOM(0, 1) restituisce 0 con probabilità $1/2$ oppure 1 con probabilità $1/2$, come se fosse effettuato il lancio di una moneta e restituita testa oppure croce. □

Il più semplice utilizzo della funzione RANDOM consiste nel permutare casualmente i dati di ingresso di un problema. In questo modo una distribuzione casuale sui dati d'ingresso viene imposta preventivamente, anziché assunta successivamente nell'analisi di complessità. Gli effetti sfavorevoli dovuti al caso pessimo sono pertanto ridotti e l'analisi della complessità nel caso medio risulta così più affidabile.

Esempio 20.2 (Quicksort). Nella procedura PERNO del QUICKSORT (cfr. § 11), l'elemento $A[primo]$ può essere preventivamente scambiato con $A[i]$, dove i è un intero generato da RANDOM(*primo*, *ultimo*). □

Un utilizzo più sofisticato, invece, permette di progettare algoritmi che nel risolvere un problema possono fornire un risultato scorretto (o addirittura non fornirne

[1] In pratica la funzione RANDOM è realizzata con un generatore di numeri pseudo-casuali, cioè da un algoritmo deterministico che fornisce numeri che dal punto di vista statistico si comportano come se fossero veramente casuali (si veda l'Appendice A).

alcuno), ma che risultano tuttavia utilissimi qualora la probabilità che tali eventi si verifichino possa essere resa arbitrariamente piccola.

Esempio 20.3 (Espressione polinomiale nulla). Data un'espressione algebrica polinomiale $p(x_1, \ldots, x_n)$ in n variabili, il problema ESPRESSIONE POLINOMIALE NULLA richiede se l'espressione è identicamente nulla oppure no. Si osservi che se p fosse dato già in forma semplificata, cioè come somma di monomi, allora il problema si risolverebbe banalmente verificando se esiste oppure no un coefficiente diverso da zero. Se invece p è dato come espressione algebrica arbitraria, allora il migliore algoritmo deterministico noto, basato sul metodo di semplificazione, richiede un tempo di calcolo molto elevato. È possibile definire un algoritmo probabilistico che dapprima genera casualmente una n-pla di valori $v_1, \ldots, v_n$ e poi valuta $p(v_1, \ldots, v_n)$. Se $p(v_1, \ldots, v_n) \neq 0$, allora l'espressione polinomiale è sicuramente non nulla. Altrimenti, se $p(v_1, \ldots, v_n) = 0$, c'è la possibilità che l'espressione sia nulla. Indicato con d il grado massimo delle variabili in p, è stato dimostrato che generando ciascun valore v_i tramite RANDOM$(1, 2d)$, $1 \leq i \leq n$, la probabilità di errore non supera $1/2$. Iterando la valutazione di p più volte e assumendo l'espressione nulla se $p(v_1, \ldots, v_n) = 0$ per tutte le n-ple generate, la probabilità di errore può essere resa piccola a piacere. □

20.1 Primalità

L'algoritmo probabilistico sicuramente più famoso è quello di Rabin (1980) per il problema della PRIMALITÀ (cfr. § 19). Tale algoritmo si basa sul concetto di "testimone di compostezza". Sia $n > 2$ il numero intero (dispari) in ingresso del quale si vuole verificare la primalità. Un numero intero b compreso tra 1 ed $n - 1$ è un testimone che n è composto (cioè non primo) se è vera una delle seguenti due condizioni:

I. b^{n-1} **mod** $n \neq 1$;
II. esiste un intero $i > 0$ tale che $m = (n-1)/2^i$ è anch'esso un intero ed inoltre $2 \leq MCD(b^m - 1, n) \leq n - 1$, dove MCD è il Massimo Comun Divisore.

Se b è un testimone che n è composto, allora n è effettivamente un numero composto. Infatti se n è primo, allora deve essere b^{n-1} **mod** $n = 1$ per il teorema di Fermat (cfr. Esercizio 19.1). Pertanto, se vale la condizione (i) allora n è composto. Inoltre, se vale la condizione (ii) allora n ammette un divisore proprio e quindi anche in questo caso n è composto. Se invece b non testimonia che n è composto, allora n può essere primo oppure composto. La verifica che b sia un testimone di compostezza per n può essere opportunamente realizzata in Pascal usando $O(\log n)$ operazioni aritmetiche su numeri interi di $O(\log n)$ bit.

Rabin ha dimostrato che se n è un numero composto, allora ci sono almeno $3(n-1)/4$ numeri compresi tra 1 ed $n - 1$ che ne testimoniano la compostezza. Ciò implica che scegliendo a caso un numero b compreso tra 1 ed $n - 1$ e verificando se b testimonia la compostezza di n oppure no, c'è una probabilità inferiore ad $1/4$ di rispondere erroneamente che n è primo quando invece è composto. Questo porta al progetto del seguente algoritmo probabilistico.


```
function PRIMO(n: integer): boolean;
    var j, b: integer; composto: boolean;
    begin
        composto := false;
        for j := 1 to 50 do begin
            b := RANDOM(1, n − 1);
            if b testimonia che n è composto
                then composto := true;
        end;
        PRIMO := not composto
    end;
```

Il precedente algoritmo ripete la verifica di compostezza su 50 valori indipendenti e uniformemente distribuiti di b e assume che n sia primo se e solo se nessuno di tali 50 valori è un testimone della compostezza. Mentre la risposta **false** (cioè che n è composto) è sempre corretta, la risposta **true** (cioè che n è primo) è corretta con probabilità maggiore o uguale a $(3/4)^{50}$. In altri termini, la probabilità di errore, cioè di assumere che n sia primo quando invece è composto, non supera $(1/4)^{50}$. Questa probabilità è talmente piccola da essere di gran lunga inferiore alla probabilità che la risposta sbagliata sia dovuta a guasti occorsi nei circuiti hardware del calcolatore! Inoltre, poiché 50 è una costante, il numero di operazioni aritmetiche della funzione PRIMO resta $O(\log n)$.

L'algoritmo di Rabin mostra come la probabilità di errore possa essere resa arbitrariamente piccola iterando il procedimento probabilistico per un opportuno numero di volte (50, nella funzione PRIMO), permettendo così la risoluzione probabilistica in tempo polinomiale di molti problemi decisionali. Si noti che questa tecnica di riduzione dell'errore non richiede che la probabilità iniziale di errore sia piccola, ma solo che sia minore di $1 - \epsilon$ per qualche $\epsilon > 0$. Se invece di $1/4$ la probabilità di errore fosse 0.99, basterebbe aumentare opportunamente il numero di iterazioni per ottenere la stessa probabilità $(1/4)^{50}$ di errore pur mantenendo la stessa complessità di $O(\log n)$ operazioni aritmetiche!

Ovviamente, la situazione ideale sarebbe quella di poter risolvere in tempo polinomiale, e con probabilità di errore arbitrariamente piccola, problemi intrattabili come quelli NP-completi. Purtroppo per i problemi NP-completi è possibile dimostrare che se un solo problema NP-completo ammettesse un algoritmo probabilistico polinomiale con probabilità di errore arbitrariamente piccola allora lo stesso varrebbe anche per tutti gli altri problemi NP-completi. Pertanto, il progetto di algoritmi probabilistici risulta conveniente per risolvere problemi polinomiali con complessità inferiore a quella di qualsiasi algoritmo deterministico polinomiale già conosciuto, oppure problemi come la PRIMALITÀ per i quali non si conosce ancora l'NP-completezza ma che sono risolvibili con algoritmi deterministici di complessità superpolinomiale.

20.2 Alberi randomizzati di ricerca

Consideriamo adesso un algoritmo per un problema già di complessità polinomiale, che non fornisce mai un risultato sbagliato ma il cui tempo di esecuzione (anziché la sua correttezza) dipende da distribuzioni di probabilità.

Gli alberi randomizzati di ricerca sono una struttura di dati proposta da Seidel e Aragon (1996) per realizzare un dizionario tramite alberi binari di ricerca. Pur non essendo mantenuto un bilanciamento esplicito come nei B-alberi binari, le operazioni

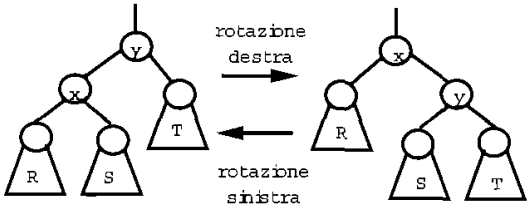


Figura 20.1: Rotazioni in un treap.

dei dizionari sono realizzabili in modo sorprendentemente semplice ed efficiente. Le *n* chiavi (distinte) del dizionario sono memorizzate negli *n* nodi di un albero binario *T* detto treap (termine ottenuto dalla contrazione di tree e heap). Ad ogni chiave è associata in modo casuale una priorità distinta. I nodi di *T* sono arrangiati secondo un doppio ordinamento:

1. le chiavi rispettano l'ordinamento (simmetrico) degli alberi binari di ricerca, cioè per ogni nodo *u* le chiavi contenute nel sottoalbero radicato nel figlio sinistro (destro) di *u* contengono chiavi minori (maggiori) della chiave memorizzata in *u* [cfr. § 8];
2. le priorità rispettano l'ordinamento degli heap (col massimo nella radice), cioè per ogni nodo *u* le priorità dei figli di *u* sono minori della priorità di *u* [cfr. § 7].

È facile verificare che per ogni insieme di *n* chiavi distinte (con associate *n* priorità distinte) il treap *T* esiste ed è unico: la chiave con priorità più alta sta nella radice, mentre le altre chiavi sono ripartite in due parti, da memorizzare nei sottoalberi radicati nei figli della radice, in accordo al loro valore rispetto a quello della chiave memorizzata nella radice. In altri termini, il treap *T* risultante è esattamente l'albero binario di ricerca che si ottiene inserendo le chiavi per priorità decrescenti ma con l'usuale algoritmo di inserzione degli alberi binari di ricerca.

L'operazione APPARTIENE segue l'usuale algoritmo degli alberi binari di ricerca, e la sua complessità è $O(h)$ dove h è il livello massimo delle foglie. Per la INSERISCI, si inserisce come al solito la nuova chiave x in una nuova foglia rispettando l'ordinamento sulle chiavi. Poiché tale inserimento può violare l'ordinamento sulle priorità, la chiave x viene fatta risalire lungo il percorso foglia-radice effettuando rotazioni destre o sinistre finché non ha un padre con priorità maggiore (oppure finché non arriva alla radice). Nella CANCELLA si effettua la stessa procedura della INSERISCI, ma invertendo l'ordine: prima si trova x, poi si fa scendere lungo il percorso radice-foglia effettuando rotazioni destre o sinistre finché x non giunge in una foglia, e a tal punto si elimina la foglia che contiene x.

Esempio 20.4 (Inserzione/cancellazione). La Figura 20.1 mostra l'effetto delle operazioni di rotazione usate. La Figura 20.2 illustra invece l'effetto delle operazioni INSERISCI e CANCELLA. □

Oltre alle usuali operazioni APPARTIENE, INSERISCI, CANCELLA, MIN, MAX, CANCELLAMIN e CANCELLAMAX tipiche degli alberi binari di ricerca, col treap si possono realizzare facilmente anche le nuove operazioni SEPARA e CONGIUNGI. Dato un valore x, SEPARA(x, D) divide il dizionario D in due dizionari D' e D'' contenenti, rispettivamente le chiavi minori di x e le chiavi maggiori di x. Viceversa,

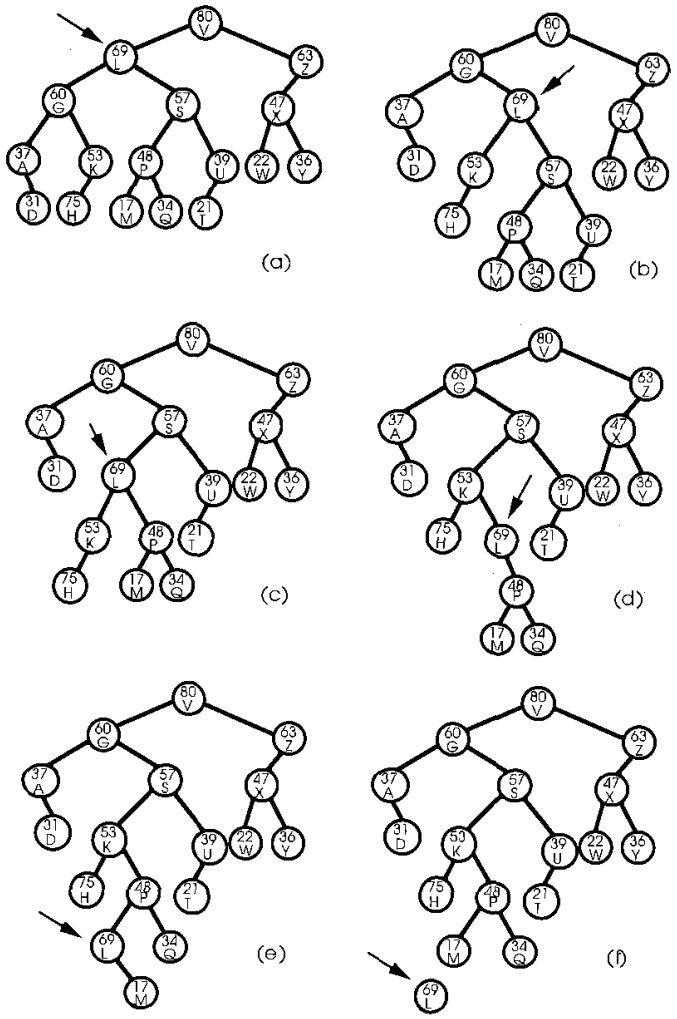


Figura 20.2: Cancellazione/inserzione in un treap della chiave *L* con priorità 69.

dati due dizionari D' e D'' tali che tutte le chiavi di D' sono minori di tutte le chiavi di D'', CONGIUNGI(D', D'') unisce i due dizionari in un unico dizionario D. Utilizzando un treap, SEPARA e CONGIUNGI si riducono ad opportune operazioni INSERISCI e CANCELLA. Infatti, per realizzare SEPARA, basta associare una priorità "infinita" alla chiave x ed effettuarne una INSERISCI. Poiché dopo l'inserzione x si troverà nella radice, per l'ordinamento simmetrico sulle chiavi D' (D'') sarà individuato dal sottoalbero radicato nel figlio sinistro (destro) della radice. Viceversa, per realizzare CONGIUNGI, basta aggiungere una nuova radice avente come figlio sinistro (destro) il treap che rappresenta D' (D'') e poi effettuare una CANCELLA della radice.

Un ulteriore vantaggio del treap è rappresentato dalla possibilità di effettuare operazioni partendo, anziché dalla radice, da un nodo qualsiasi del treap accessibile direttamente tramite un puntatore. Per esempio, per cancellare un nodo u accessibile con un puntatore, non è necessaria alcuna ricerca, ma basta farlo scendere tramite rotazioni destre o sinistre fino a renderlo una foglia e poi eliminarlo. In modo analogo, un'inserzione di una chiave x può essere accelerata se è disponibile un puntatore al nodo u che contiene la chiave che segue (precede) immediatamente x, poiché la ricerca per l'inserzione può partire da u anziché dalla radice. Per effettuare tali inserzioni e cancellazioni è necessario che ogni nodo del treap contenga il puntatore al padre.

Infine, è possibile effettuare una finger search, dove si vuole individuare il nodo u del treap contenente la chiave x partendo da un altro nodo v (possibilmente "vicino" ad u) contenente una chiave y e accessibile tramite puntatore. Infatti basta percorrere l'unico cammino tra v ed u salendo da v fino al nodo w che è l'antenato comune di u e v di livello massimo e poi scendendo verso u. Per effettuare tali ricerche però, oltre ai puntatori ai padri, sono necessarie altre informazioni addizionali nei nodi. Una possibilità consiste nel mantenere per ogni nodo u i valori $\min(u)$ e $\max(u)$ delle chiavi minima e massima che si trovano nel sottoalbero radicato in u. In questo modo, il nodo w è il primo che si incontra tale che $\min(w) \leq x \leq \max(u)$. Le informazioni min() e max() possono essere mantenute senza aumentare la complessità mentre sono effettuate rotazioni, discese e risalite. A tal fine, occorre mantenere due nodi sentinella che non sono mai cancellati con valori delle chiavi uguali a $-\infty$ e $+\infty$ e con priorità scelte in modo casuale. Invece di memorizzare due chiavi aggiuntive, si possono memorizzare in ogni nodo due puntatori aggiuntivi a due nodi opportuni.

Nel realizzare un treap, non è necessario memorizzare esplicitamente le priorità nei nodi. Per esempio, si può invece usare una opportuna funzione hash H che associa una priorità $H(x)$ ad ogni chiave x e che può essere ricalcolata in tempo $O(1)$ ogni volta che serve. Alcune "buone" funzioni hash sono state proposte da Seidel e Aragon (1996) che suggeriscono anche la possibilità di usare come priorità, anziché i valori forniti da una funzione hash, gli indirizzi stessi delle celle di memoria che contengono i nodi del treap.

Le complessità attese per le operazioni sui treap sono riassunte nella seguente tabella e valgono per priorità casuali, indipendenti ed identicamente distribuite (per esempio, nell'intervallo $1..maxint$).

APPARTIENE, INSERISCI, CANCELLA	$O(\log n)$
SEPARA, CONGIUNGI	$O(\log \max\{n, m\})$
INSERISCI con puntatore al nodo successore o predecessore	$O(1)$
CANCELLA con puntatore al nodo	$O(1)$
FINGER SEARCH (su un cammino con d archi)	$O(d)$

20.3 Classi probabilistiche di complessità

La nozione di algoritmo probabilistico può essere formalizzata in termini dell'usuale nozione di algoritmo non deterministico, ma interpretata in modo diverso. Ricordiamo che la computazione di un algoritmo non deterministico è caratterizzata da un albero delle scelte, dove ogni percorso radice-foglia corrisponde ad una sequenza di scelte, ovvero alla computazione di una copia dell'algoritmo. Una scelta è effettuata tramite l'istruzione **choice**(I), che sceglie arbitrariamente un elemento dall'insieme I, e quindi induce nell'albero delle scelte un nodo con $|I|$ figli. La computazione restituisce risposta SI se e solo se esiste una sequenza di scelte che porta allo stato di successo, ovvero se almeno una foglia dell'albero esegue una **success**, mentre dà risposta NO se tutte le scelte portano allo stato di fallimento, cioè se tutte le foglie dell'albero eseguono **failure**.

Per passare dal non determinismo al probabilismo, occorre soltanto reinterpretare la "modalità" di ogni scelta e ridefinire il "risultato" di una computazione nell'albero delle scelte. Una scelta non è più effettuata non deterministicamente tramite la funzione **choice**, bensì probabilisticamente tramite la funzione RANDOM(h, k). In altri termini, i $k - h + 1 = |I|$ figli di un nodo nell'albero delle scelte sono visti come possibili alternative equiprobabili anziché come scelte non deterministiche. La computazione probabilistica restituisce risposta SI se più della metà delle foglie dell'albero portano ad uno stato di successo (cioè eseguono **success**), mentre dà risposta NO se più della metà delle foglie portano allo stato di fallimento (cioè eseguono **failure**).

In base alle definizioni precedenti, Gill (1977) ha introdotto una sequenza di classi probabilistiche di complessità, dove ogni classe è ottenuta dalla precedente imponendo sempre ulteriori vincoli da rispettare.

La classe più ampia è quella ottenuta richiedendo semplicemente che l'albero delle scelte abbia un numero polinomiale di livelli (con tutte le foglie all'ultimo livello) e che sia sempre fornita una risposta (SI oppure NO). La classe risultante è detta $\mathbb{PP}$ (per "Polinomiale Probabilistica") e corrisponde agli algoritmi probabilistici meno utili in pratica, perché la probabilità di errore può essere $1/2 - (1/2)^n$, dove n è la dimensione dell'input. In tal caso, la probabilità di errore non può essere resa arbitrariamente piccola a meno di non usare un numero esponenziale di iterazioni. È facile verificare che valgono le seguenti inclusioni.

Teorema 20.1. $\mathbb{NP} \cup \text{co-}\mathbb{NP} \subseteq \mathbb{PP} \subseteq \mathbb{P}\text{-}\mathbb{SPAZIO}$. □

Imponendo che, per qualche $\epsilon > 0$, il rapporto delle foglie dell'albero delle scelte che danno risposta corretta sia almeno $1/2 + \epsilon$, si ottiene la classe $\mathbb{BPP}$ (dove la "B" sta per "bounded", cioè "limitata") che corrisponde agli algoritmi probabilistici di maggior interesse pratico. Infatti con questa classe si può ottenere una probabilità di errore arbitrariamente piccola con un numero di iterazioni limitato da un polinomio. La classe $\mathbb{BPP}$ è inconfrontabile con la classe $\mathbb{NP}$, nel senso che nessuna delle due classi contiene l'altra.

Con l'ulteriore restrizione che l'albero delle scelte non abbia alcuna foglia che esegue **success** quando la risposta è NO si ottiene la classe $\mathbb{RP}$ (per "Random Polinomiale") talvolta indicata anche con $\mathbb{R}$. In altri termini, gli algoritmi in $\mathbb{RP}$ danno risposta NO con probabilità di errore nulla (e ovviamente una probabilità di errore piccola a piacere può essere ottenuta con un numero polinomiale di iterazioni). I problemi decisionali visti precedentemente (ESPRESSIONE POLINOMIALE NULLA e PRIMALITÀ) appartengono a questa classe o al suo complemento co-$\mathbb{RP}$. È facile

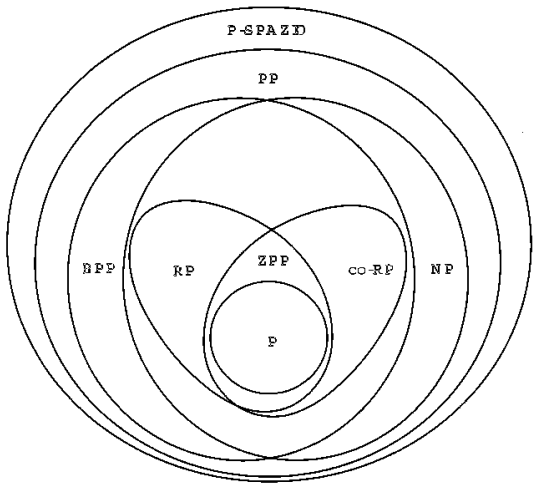


Figura 20.3: Presunte relazioni di contenimento tra classi probabilistiche di complessità.

vedere che mentre $\mathbb{PP}$ e $\mathbb{BPP}$ sono chiuse rispetto al complemento, $\mathbb{RP}$ invece non lo è. Per esempio, l'algoritmo di Rabin mostra che il complemento di PRIMALITÀ (cioè COMPOSTEZZA, cfr. § 19) è in $\mathbb{RP}$, ma nessuno finora ha dimostrato se PRIMALITÀ è in $\mathbb{RP}$. Pertanto, l'intersezione $\mathbb{RP} \cap$ co-$\mathbb{RP}$ potrebbe anche essere un sottoinsieme proprio di $\mathbb{RP}$.

Teorema 20.2. $\mathbb{RP} \cap$ co-$\mathbb{RP} \subseteq \mathbb{RP} \subseteq \mathbb{BPP}$ ed $\mathbb{RP} \subseteq \mathbb{NP}$. □

Infine, imponendo che la probabilità di errore sia sempre nulla, si ottiene la classe $\mathbb{ZPP}$ (cioè i problemi risolvibili in tempo atteso polinomiale con probabilità zero di errore, come nel caso degli alberi randomizzati di ricerca). Un algoritmo in $\mathbb{ZPP}$ è caratterizzato da tre stati finali: uno di successo, uno di fallimento, ed uno di incertezza. Algoritmi di questo tipo sono detti Las Vegas e non danno mai una risposta scorretta, ma possono in taluni casi non dare alcuna risposta con probabilità minore o uguale a $1/2$. Essi sono contrapposti agli algoritmi visti precedentemente che invece possono "mentire", cioè dare una risposta scorretta, e che sono invece chiamati Monte Carlo. È possibile dimostrare che la classe $\mathbb{ZPP}$ coincide proprio con $\mathbb{RP} \cap$ co-$\mathbb{RP}$:

Teorema 20.3. $\mathbb{P} \subseteq \mathbb{ZPP} = \mathbb{RP} \cap$ co-$\mathbb{RP}$. □

Riassumendo, le presunte relazioni di contenimento fra le classi probabilistiche di complessità sono mostrate nella Figura 20.3. Nessuna di tali inclusioni è stata finora dimostrata essere propria, anche se questa congettura rappresenta l'ipotesi più accreditata. Infatti, sono stati dimostrati i seguenti risultati.

Teorema 20.4. Nessun problema NP-completo può essere in $\mathbb{RP}$, a meno che $\mathbb{RP} = \mathbb{NP}$, né può essere in $\mathbb{ZPP}$, a meno che $\mathbb{ZPP} = \mathbb{NP}$. □

20.4 Esercizi

Esercizio 20.1 (Testimone di compostezza). Si dimostri che la verifica che b è un testimone della compostezza di n può essere fatta in tempo polinomiale.

Esercizio 20.2 (Alberi randomizzati di ricerca). Si scrivano le procedure Pascal per le operazioni sugli alberi randomizzati di ricerca realizzati con puntatori.

Esercizio 20.3 (Prodotto di matrici). Date tre matrici $n \times n$ A, B, e C, si vuole decidere se $AB = C$. Si fornisca un algoritmo probabilistico di complessità $O(n^2)$ per verificare se $AB = C$ con probabilità di errore al più $1/2$ (Suggerimento: si generi un vettore x di n elementi uguali a -1 oppure $+1$ e si verifichi se $A(Bx) \neq Cx$).

Esercizio 20.4 ($\frac{1}{2}$-Soddisfattibilità). Data una formula booleana, si vuole verificare se la formula è soddisfatta da più della metà dei possibili assegnamenti dei valori di verità alle variabili booleane. Si fornisca un algoritmo appartenente alla classe $\mathbb{PP}$.

Capitolo 21

Algoritmi pseudo-polinomiali

Si consideri il problema SOMMA DI SOTTOINSIEME, del quale è stata dimostrata l'NP-completezza nel § 18:

SOMMA DI SOTTOINSIEME: "Dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi ed un intero positivo k, esiste un sottoinsieme S di indici in $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i = k$?"

Si supponga di memorizzare l'insieme A in un vettore $A[1..n]$ di interi e si introduca un vettore booleano $M[0..k]$ di $k + 1$ elementi. Il problema può essere facilmente risolto con la seguente funzione Pascal, basata sulla tecnica della "programmazione dinamica":

```
function SOMMASOTTINSIEME(var A: vettore; n, k: integer): boolean;
    var i, j: integer;
        M: packed array[0..k] of boolean;
    begin
        for j := 1 to k do M[j] := false;
        M[0] := true;
        for i := 1 to n do
            for j := k downto A[i] do
                M[j] := M[j - A[i]] or M[j];
        SOMMASOTTINSIEME := M[k]
    end;
```

In accordo alla tecnica della "programmazione dinamica", il problema viene risolto via via per $i = 1, 2, \ldots, n$. All'iterazione i-esima, il valore booleano di $M[j]$ indica se esiste un sottoinsieme di $\{a_1, \ldots, a_i\}$ la cui somma dà esattamente j. Inizialmente $M[0]$ è posto a **true** perché l'insieme vuoto dà somma 0. All'iterazione i-esima, viene considerata la possibilità di aggiungere a_i. L'assegnamento "$M[j] := M[j - A[i]]$ **or** $M[j]$" prevede i due casi possibili in cui a_i sia aggiunto ad un sottoinsieme di $\{a_1, \ldots, a_{i-1}\}$ che dà somma $j - a_i$, oppure non sia aggiunto ad un sottoinsieme di $\{a_1, \ldots, a_{i-1}\}$ che dà somma j. Nel primo caso, $M[j]$ è uguale ad $M[j - A[i]]$, mentre nel secondo caso $M[j]$ mantiene il suo vecchio valore. Effettuando l'operazione **or**, $M[j]$ indica correttamente se c'è un sottoinsieme di $\{a_1, \ldots, a_i\}$ con somma j. Si noti che il ciclo **for** più interno procede per valori decrescenti di j, onde evitare di considerare più occorrenze di a_i nel medesimo sottoinsieme, come invece potrebbe accadere procedendo per valori crescenti di j. La funzione restituisce

il valore **true** se e solo se $M[k] =$ **true**, ovvero se esiste un sottoinsieme di $\{a_1, \ldots, a_n\}$ la cui somma dà k.

Esempio 21.1 (Somma di sottoinsieme). Siano $n = 3$, $k = 24$, $a_1 = 7$, $a_2 = 11$, $a_3 = 13$. La seguente tabella illustra il contenuto del vettore M per $i = 0, 1, 2, 3$:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
		1																							
$i = 1$		1						1																	
$i = 2$		1						1				1							1						
$i = 3$		1						1				1		1					1		1				1

□

La complessità della funzione SOMMASOTTINSIEME è ovviamente $O(nk)$, che non è polinomiale nella dimensione del problema. Infatti, assumendo $a_i \leq k$ per $1 \leq i \leq n$, poiché altrimenti a_i può essere eliminato, la dimensione d dei dati di ingresso è $O(n \log k)$, ma non $O(nk)$. Se k è limitata da un polinomio in n, cioè k è $O(n^c)$ con c costante, allora la funzione SOMMASOTTINSIEME ha complessità polinomiale, poiché nk risulta essere $O(n^{c+1})$. Se invece k è superpolinomiale in n, allora la complessità della funzione risulta essere anch'essa superpolinomiale. Infatti, se k è $O(2^n)$, allora nk è $O(n2^n)$.

In altri termini, la complessità di SOMMASOTTINSIEME non dipende soltanto dalla cardinalità n del sottoinsieme in ingresso, ma anche dai valori degli elementi del sottoinsieme in ingresso. Tale complessità risulta polinomiale oppure no a seconda che i valori degli elementi in ingresso siano "piccoli" oppure "grandi" rispetto al loro numero. Una situazione analoga si è incontrata per la procedura COUNTING-SORT (cfr. Esercizi 11.3 e 18.2), che permette di ordinare n elementi in tempo $O(n)$ qualora i valori di tali elementi siano "piccoli", cioè limitati anch'essi da $O(n)$.

In questo capitolo, vedremo sotto quali condizioni un problema NP-completo può essere risolto con un algoritmo polinomiale per valori "piccoli" dei dati in ingresso. Inoltre, saranno illustrati altri due algoritmi con queste caratteristiche per risolvere il problema dello ZAINO in forma di ottimizzazione e un problema di "scheduling" di processi periodici di rilevante interesse pratico.

21.1 NP-completezza "forte"

Dato un problema decisionale P, con insieme di dati di ingresso I, distinguiamo tra la "dimensione" d di I, cioè la lunghezza della stringa che codifica I, ed il "più grande numero" intero # che appare in I.

Esempio 21.2 (dimensione d e numero #).

1. Se $P =$ SOMMA DI SOTTOINSIEME, allora $I = \{n, k, a_1, \ldots, a_n\}$, $\# = \max\{n, k, \max_i\{a_i\}\}$, mentre d è $O(n \log \#)$;
2. Se $P =$ COMMESSO VIAGGIATORE, allora $I = \{n, k, [d_{ij}]\}$, $\# = \max\{n, k, \max_{ij}\{d_{ij}\}\}$, mentre d è $O(n^2 \log \#)$;
3. Se $P =$ CRICCA, allora $I = \{n, m, k, G = (N, A)\}$, $\# = \max\{n, m, k\}$, mentre d è $O(n + m + \log \#)$. □

Sia P_p il problema P ristretto a quei dati d'ingresso per i quali # è limitato superiormente da $p(d)$, con p funzione polinomiale in d. P è "fortemente NP-completo" se P_p è NP-completo.

Esempio 21.3 (NP-completezza forte).

1. La SOMMA DI SOTTOINSIEME non è fortemente NP-completa; infatti, se $\# = n$ il problema è risolubile in tempo polinomiale, mentre se $\# = k$, allora # non può in generale essere limitato superiormente da alcun polinomio nella dimensione $d = O(n \log k)$ (ovviamente se $\# = a_i > k$ allora a_i non può far parte di alcuna soluzione e può essere eliminato);
2. Il COMMESSO VIAGGIATORE è fortemente NP-completo, perché resta NP-completo anche quando $d_{ij} \in \{1, 2\}$, $1 \leq i, j \leq n$, e $k = n$ [si veda la dimostrazione del Teorema 18.5]; in tal caso, $\# = n$, che è ovviamente limitato superiormente da un polinomio nella dimensione $d = O(n^2 \log n)$;
3. Anche la CRICCA è fortemente NP-completa, poiché k può essere assunto minore o uguale ad n (se $k > n$, la risposta del problema è sempre NO) e quindi # è limitato superiormente da un polinomio in $d = O(n + m + \log(m + n))$. □

In pratica, sono "fortemente" NP-completi tutti quei problemi che possono essere dimostrati NP-completi con riduzioni che non usano numeri interi esponenzialmente grandi, mentre sono "debolmente" NP-completi quei problemi la cui dimostrazione di intrattabilità dipende pesantemente da riduzioni che usano numeri interi esponenzialmente grandi. Per esempio, le riduzioni illustrate per il DOMINO LIMITATO, la CRICCA, la SODDISFATTIBILITÀ, la PROGRAMMAZIONE LINEARE 0/1, l'INSIEME INDIPENDENTE, il COMMESSO VIAGGIATORE richiedono numeri interi che sono polinomialmente limitati dalla dimensione d dell'input (in alcuni casi sono addirittura limitati da costanti). Pertanto, tutti questi problemi sono fortemente NP-completi. Viceversa, ciò non è vero per la SOMMA DI SOTTOINSIEME, la PARTIZIONE e lo ZAINO, che sono invece debolmente NP-completi.

Formalmente, un algoritmo che risolve un certo problema P, per qualsiasi dato I d'ingresso, in tempo $p(\#, d)$, con p funzione polinomiale in # e d, ha complessità pseudo-polinomiale. L'algoritmo visto per risolvere il problema SOMMA DI SOTTOINSIEME ha complessità pseudo-polinomiale. Vale il seguente risultato:

Teorema 21.1. *Nessun problema fortemente NP-completo può essere risolto da un algoritmo pseudo-polinomiale, a meno che non sia* $\mathbb{P} = \mathbb{NP}$.

Dimostrazione. Se P è fortemente NP-completo, allora il problema ristretto P_p è NP-completo per qualche polinomio p. Se esistesse un algoritmo pseudo-polinomiale A per P, allora A risolverebbe P_p in tempo polinomiale, che è assurdo, a meno che non sia $\mathbb{P} = \mathbb{NP}$. □

Il più famoso problema fortemente NP-completo, in cui numeri interi hanno un ruolo rilevante, è la cosiddetta 3-PARTIZIONE, che generalizza appunto il problema della PARTIZIONE:

3-PARTIZIONE: "Dati $3n$ interi $\{a_1, \ldots, a_{3n}\}$, esiste una partizione in n triple $T_1, \ldots, T_n$, tale che la somma dei tre elementi di ogni T_j è la stessa, per $1 \leq j \leq n$?"

Al contrario della PARTIZIONE, della SOMMA DI SOTTOINSIEMI e dello ZAINO, che sono tutti risolubili in tempo pseudo-polinomiale, la 3-PARTIZIONE non lo è.

Per dimostrare l'NP-completezza forte di un problema P, non è sempre necessario dimostrare l'NP-completezza di un sottoproblema P_p. Infatti, è sufficiente provare l'esistenza di un altro problema Q, di cui si conosce già l'NP-completezza forte, che si può ridurre a P con una trasformazione che richiede tempo pseudo-polinomiale. Tale trasformazione deve inoltre garantire che $\#_P$, il più grande intero introdotto nei dati di P, sia limitato superiormente da una funzione $p(d_Q, \#_Q)$ polinomiale nella dimensione d_Q dell'input di Q e nel più grande intero $\#_Q$ che appare in tale input.

21.2 Zaino

Mostriamo una semplice riduzione pseudo-polinomiale dal problema dello ZAINO, in forma di ottimizzazione, a quello dei CAMMINI MINIMI, che permette di risolvere lo ZAINO in tempo pseudo-polinomiale.

ZAINO: "Dati un insieme $O = \{o_1, \ldots, o_n\}$ di "oggetti", tali che o_i è caratterizzato da un "profitto" p_i e da un "volume" v_i, entrambi interi positivi, e un intero positivo c (la "capacità" dello zaino), trovare un sottoinsieme S di $\{1, \ldots, n\}$ tale che $\sum_{i \in S} v_i \leq c$ e $\sum_{i \in S} p_i$ sia massima."

È possibile definire un grafo orientato pesato $G = (N, A)$ avente $|N| = n(c + 1) + 2$ nodi, indicati con i^k, per $1 \leq i \leq n$ e $0 \leq k \leq c$, s, e t, ed i seguenti archi. L'arco i^k ha due archi entranti, uno proveniente dal nodo $(i-1)^k$, la cui lunghezza è zero, e l'altro proveniente dal nodo $(i-1)^{k-v_i}$, la cui lunghezza è $-p_i$, purché questi nodi esistano. Dal nodo s escono due archi, diretti al nodo 1^0, la cui lunghezza è zero, e al nodo 1^{v_i}, la cui lunghezza è $-p_1$. Infine, nel nodo t entrano $c + 1$ archi, provenienti dai nodi n_k, $0 \leq k \leq c$, tutti di lunghezza zero. È facile verificare che ogni cammino da s a i^k corrisponde ad un sottoinsieme dei primi i oggetti il cui volume è esattamente uguale a k e il cui profitto è uguale alla lunghezza del cammino, cambiata di segno. Ogni cammino minimo tra s e t identifica pertanto una soluzione ottima dello ZAINO. Essendo G aciclico, un cammino minimo tra s e t può essere trovato in tempo $O(|N| + |A|)$ con l'algoritmo di Bellman-Ford-Moore [cfr. § 10]. Poiché $|A|$ è $O(|N|)$, $|N|$ è $O(n^c)$, e la costruzione di G richiede $O(n^c)$ tempo, il problema dello ZAINO, nella sua versione più generale di ottimizzazione, può essere risolto in tempo pseudo-polinomiale $O(n^c)$.

Esempio 21.4 (Riduzione da ZAINO a CAMMINI MINIMI). Il grafo G, costruito per il problema dello ZAINO con $n = 3$, $c = 4$, volumi $v_1 = 2$, $v_2 = 3$, $v_3 = 1$, e profitti $p_1 = 5$, $p_2 = 1$, $p_3 = 3$, è mostrato nella Fig. 21.1. La soluzione ottima dello ZAINO comprende gli oggetti o_1 e o_3 e corrisponde al cammino minimo $s, 1^2, 2^2, 3^3, t$. □

21.3 Real-time scheduling

In un problema per l'elaborazione in tempo reale ("real-time"), la correttezza del risultato non dipende soltanto dal valore del risultato, ma anche da quando tale risultato è prodotto. In altri termini, ci sono delle costrizioni temporali che il risultato deve assolutamente rispettare, in modo da evitare conseguenze che possono essere catastrofiche (come lo scoppio di una centrale nucleare o la perdita del controllo di un aereo). I problemi real-time sono talvolta erroneamente scambiati con i problemi on line (che abbiamo incontrato nel § 16). Naturalmente, vi sono molti problemi real-time che sono anche on line, ma le costrizioni temporali non sono necessariamente critiche in tutti i problemi on line, così come la necessità di elaborare i dati in ingresso via via che si presentano non sempre è richiesta in tutti i problemi real-time.

In particolare, in un problema di scheduling real-time, è dato un insieme di processi periodici che devono essere eseguiti su un processore. Ciascun processo ricorre infinite volte, in accordo ad un tempo di interarrivo regolare e costante nel tempo (il suo periodo), e deve rispettare delle scadenze temporali (o deadlines). Infatti, ciascuna occorrenza periodica del processo deve essere interamente eseguita prima

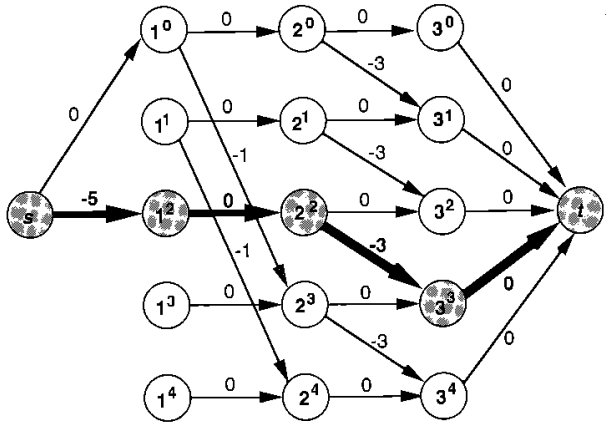


Figura 21.1: Esempio di riduzione pseudo-polinomiale da ZAINO a CAMMINI MINIMI.

della fine del suo periodo, cioè prima che si presenti la prossima occorrenza dello stesso processo periodico. Pertanto, si richiede di trovare un ordine in cui eseguire i processi (per infinite volte) in modo da rispettare le scadenze temporali di tutte le occorrenze periodiche dei processi. Gli algoritmi di scheduling che sono usati in pratica dai sistemi operativi real-time usano "prelazione" (preemption) dei processi e sono "guidati da priorità" (priority-driven). Le priorità sono assegnate una volta per tutte a ciascun processo in accordo ad un prefissato criterio. Ad ogni istante di tempo, viene eseguito il processo pronto per l'esecuzione che ha massima priorità, interrompendo se necessario un processo in esecuzione con priorità minore. Il processo interrotto sarà ripreso più tardi dal punto di interruzione, quando non ci sarà alcun processo pronto con priorità maggiore di esso. Pertanto, se un processo richiede C unità di tempo di computazione, la sua esecuzione può essere frazionata in più spezzoni di lunghezza qualsiasi, la somma dei quali deve ovviamente dare C. In sintesi, il problema REAL-TIME SCHEDULING può essere enunciato come segue:

REAL-TIME SCHEDULING: "Dato un insieme $\{p_1, \ldots, p_n\}$ di processi periodici, dove ciascun p_i è caratterizzato dal suo tempo di computazione C_i ed il suo periodo T_i (entrambi interi con $0 < C_i \leq T_i$), è possibile eseguire tutte le occorrenze periodiche dei processi per mezzo di un algoritmo con prelazione guidato da priorità in modo che ciascuna occorrenza periodica di ogni processo sia completamente eseguita prima della successiva occorrenza dello stesso processo, cioè entro la fine del suo periodo?"

In altri termini, la prima occorrenza del processo p_i dovrà essere eseguita nell'intervallo $[0, T_i]$, la seconda nell'intervallo $[T_i, 2T_i]$, la terza in $[2T_i, 3T_i]$, e così via. Il problema è NP-completo, ma può essere risolto in tempo pseudo-polinomiale. Un algoritmo guidato da priorità che è molto usato in ambito industriale è l'algoritmo Rate-Monotonic (RM), ideato da Liu e Layland (1972), dove a ciascun processo è assegnata una priorità in accordo al suo tasso di richieste (cioè all'inverso del periodo). In altri termini, l'algoritmo RM assegna priorità più alta al processo con periodo più breve. Pertanto, assumiamo che i processi $p_1, \ldots, p_n$ siano stati ordinati in accordo alla priorità RM, cioè in modo che $T_1 \leq T_2 \leq \cdots \leq T_n$.

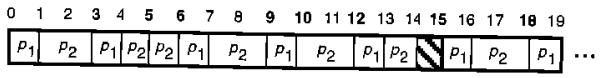


Figura 21.2: Algoritmo Rate-Monotonic. Esecuzione dei processi periodici p_1 e p_2 con $C_1 = 1$, $T_1 = 3$, $C_2 = 3$, e $T_2 = 5$.

Esempio 21.5 (Rate-Monotonic). Consideriamo due processi p_1 e p_2 con $C_1 = 1$, $T_1 = 3$, $C_2 = 3$, e $T_2 = 5$. Il processo p_1 ha priorità maggiore di p_2, e la prima richiesta di p_1 è eseguita nell'intervallo $[0, 1]$. Successivamente, la prima richiesta di p_2 è eseguita in $[1, 3]$. All'istante 3, giunge la seconda richiesta di p_1, l'esecuzione di p_2 è interrotta, e p_1 è eseguito in $[3, 4]$. All'istante 4 è ripresa l'esecuzione di p_2 interrotta precedentemente, ed è completata nell'intervallo $[4, 5]$, e così via. Lo schedule (infinito) risultante è periodico con periodo 15, il minimo comune multiplo di $T_1 = 3$ e $T_2 = 5$, ed è illustrato nella Figura 21.2. □

È possibile dimostrare che un insieme di processi periodici può essere eseguito dall'algoritmo RM se è rispettata la scadenza della prima richiesta di ciascun processo a partire da un istante nel quale avvengono contemporaneamente tutte le richieste dei processi, poiché è questo il caso peggiore nel quale il processore è "sovraccaricato" al massimo. Per esempio, un tale istante è l'istante 0 (o qualsiasi istante che sia un multiplo del minimo comune multiplo dei periodi dei processi). Come conseguenza, per verificare se un qualsiasi processo p_i potrà essere eseguito per un numero indefinito di volte, è sufficiente verificare che p_i possa essere eseguito una sola volta nel suo primo periodo $[0, T_i]$ insieme a tutti i processi che hanno priorità più alta di p_i, cioè insieme a $p_1, \ldots, p_{i-1}$. Grazie a questa proprietà, è facile dimostrare che tutte le occorrenze periodiche di p_i possono essere eseguite se e solo se:

$$\min_{0<t\leq T_i} \left\{ \sum_{1\leq k\leq i} C_k \left\lceil \frac{t}{T_k} \right\rceil \right\} \leq t.$$

Intuitivamente, $\lceil t/T_k \rceil$ fornisce il numero di occorrenze di p_k nell'intervallo $[0, t]$, e quindi $C_k \lceil t/T_k \rceil$ rappresenta il tempo di computazione richiesto per eseguire tali occorrenze in $[0, t]$. Pertanto, p_i può essere completato in $[0, t]$ se e solo se il tempo di computazione richiesto dalle occorrenze dei processi $p_1, \ldots, p_i$ non supera t. Se il minimo valore di t per il quale ciò accade è minore o uguale a T_i, allora p_i può essere eseguito una volta in $[0, T_i]$ insieme a tutti i processi che hanno priorità più alta di p_i. Inoltre, l'insieme $\{p_1, \ldots, p_n\}$ di tutti gli n processi può essere eseguito (un numero indefinito di volte) se e solo se:

$$\max_{1\leq i\leq n} \min_{0<t\leq T_i} \left\{ \sum_{1\leq k\leq i} C_k \left\lceil \frac{t}{T_k} \right\rceil \right\} \leq t.$$

La minimizzazione non è difficile da calcolare. Infatti la condizione su t deve essere verificata per un numero finito di volte. Consideriamo l'insieme $\{p_1, \ldots, p_i\}$ dei primi i processi (dove vale $T_1 \leq \cdots \leq T_i$). Il carico cumulativo $W_i(t)$ del

processore richiesto per eseguire i processi in $\{p_1, \ldots, p_i\}$ durante $[0, t]$ è:

$$W_i(t) = \sum_{1 \le k \le i} C_k \left\lceil \frac{t}{T_k} \right\rceil .$$

Consideriamo una sequenza di istanti $S_{i,0}, S_{i,1}, S_{i,2}, \ldots$ tali che:

$$\begin{aligned} S_{i,0} &= \sum_{1 \le k \le i} C_k, \\ S_{i,l+1} &= W_i(S_l). \end{aligned}$$

Se per qualche l si ha che $S_{i,l} = S_{i,l+1} \le T_i$, allora il processo p_i può essere eseguito in $[0, T_i]$. Altrimenti, se $T_i < S_{i,l}$ per qualche l, allora p_i non può essere eseguito in $[0, T_i]$. Si noti che $S_{i,l}$ è proprio uguale al minimo t, $0 < t \le T_i$, per il quale $\sum_{1 \le k \le i} C_k \lceil t/T_k \rceil \le t$, come richiesto. Questo algoritmo di verifica è chiamato CTT (per "Completion Time Test") ed è ripetuto per $i = 1, 2, \ldots, n$. CTT è stato ideato da Joseph e Pandya (1986) e richiede tempo pseudo-polinomiale, poiché la sua complessità dipende dai valori dei periodi dei processi e, in particolare, da quello del periodo più grande T_n. Infatti, per ciascun i, CTT richiede $O(nT_i)$ tempo. Poiché CTT è ripetuto n volte e $T_1 \le T_2 \le \ldots \le T_n$, in totale è richiesto $O(n^2 T_n)$ tempo.

Esempio 21.6 (CTT). Consideriamo ancora i processi p_1 e p_2 dell'Esempio 21.5, e verifichiamo se p_2 è eseguibile. Si ottiene $S_{2,0} = 1 + 3 = 4$, $S_{2,1} = W_2(4) = 1\lceil 4/3 \rceil + 3\lceil 4/5 \rceil = 5$, e $S_{2,2} = W_2(5) = 1\lceil 5/3 \rceil + 3\lceil 5/5 \rceil = 5$. Poiché $S_{2,1} = S_{2,2} \le T_2 = 5$, tutte le occorrenze periodiche di p_2 rispetteranno le loro scadenze temporali. □

Nel caso in cui, anziché un singolo processore, siano disponibili $m > 1$ processori, si partizionano preventivamente i processi tra gli m processori, e si eseguono i processi assegnati a ciascun processore come nel caso di processore singolo, usando l'algoritmo RM. Per partizionare i processi tra i processori, viene usato l'algoritmo RMFF (per "Rate-Monotonic First-Fit"). Siano $T_1 \le T_2 \le \cdots \le T_n$. Per $i = 1, 2, \ldots, n$, il generico processo p_i è assegnato al primo processore (cioè quello con indice minimo) tale che p_i e tutti gli altri processi già assegnati al medesimo processore superano la verifica dell'algoritmo CTT.

Poiché lo scopo degli algoritmi real-time è quello di fornire un servizio di controllo critico, le scadenze temporali dei processi devono essere garantite anche in presenza di guasti ai processori. Pertanto, gli algoritmi CTT e RMFF sono stati estesi per tollerare guasti dei processori che sono rilevati on line durante l'esecuzione dei processi periodici. Una strategia, proposta da Bertossi, Mancini e Rossini (1999), consiste nel costruire preventivamente più schedule diversi per $m - 1$ processori, prevedendo in anticipo il possibile guasto di ciascuno degli m processori. Quando il guasto di un processore è rilevato on line, lo schedule originale per m processori è sostituito repentinamente con quello opportuno per $m - 1$ processori (escludendo così il processore guasto). Nel far questo, vengono completate o addirittura rieseguite quelle occorrenze dei processi le cui computazioni non erano andate a buon fine a causa del guasto al processore. La complessità degli algoritmi CTT e RMFF così estesi resta comunque pseudo-polinomiale.

21.4 Esercizi

Esercizio 21.1 (Partizione). Si progetti un algoritmo di "programmazione dinamica" con complessità pseudo-polinomiale per la PARTIZIONE.

Esercizio 21.2 (Zaino). Fornire un algoritmo di "programmazione dinamica" con complessità pseudo-polinomiale per il problema dello ZAINO in forma di ottimizzazione.

Esercizio 21.3 (Scheduling multiprocessore). Fornire un algoritmo di "programmazione dinamica" con complessità pseudo-polinomiale per il problema dello SCHEDULING MULTIPROCESSORE definito nel § 18. Si noti che non è possibile frazionare l'esecuzione di un programma in più porzioni, a differenza del problema REAL-TIME SCHEDULING.

Esercizio 21.4 (Algoritmo di McNaughton). Si dimostri che SCHEDULING MULTIPROCESSORE in forma di ottimizzazione è risolubile in tempo $O(n)$ se è possibile frazionare l'esecuzione di un programma in più porzioni (aventi lunghezze reali), evitando ovviamente che due porzioni dello stesso programma siano eseguite nello stesso istante da più di un processore (Suggerimento: si consideri $d = \max\{\max_i\{t_i\}, (1/m)\sum_i t_i\}$).

Esercizio 21.5 (Real-time scheduling). Si dimostri che REAL-TIME SCHEDULING, in forma decisionale, è risolubile in tempo polinomiale anche per $m \ge 1$ processori, qualora non sia richiesto di usare un algoritmo guidato da priorità (Suggerimento: ci si riconduca all'Esercizio 21.4, ponendo $d = 1$ e $t_i = C_i/T_i$).

Esercizio 21.6 (I "soliti ignoti"). La scorsa notte, i "soliti ignoti" hanno svuotato l'oreficeria "Bella Gioia", usando la tecnica del "buco". Dopo il fattaccio, i tre componenti della banda si sono riuniti per spartire il bottino in parti uguali. Il valore del bottino è stimato in m Euro, con m multiplo di 3, sommando i prezzi riportati sui cartellini degli n monili. Dopo lunghe discussioni, il capo della banda ha scritto una procedura Pascal per effettuare la spartizione usando la tecnica della "programmazione dinamica".

La seguente procedura BOTTINO utilizza una matrice booleana a tre dimensioni SPA[1..n, -m..m, -m..m]. SPA[i, j, h] = **true** se i primi i monili sono spartiti dando valore j al primo ladro, h al secondo, ed il rimanente al terzo. La spartizione è possibile se e solo se SPA[n, $m/3$, $m/3$] = **true**, e la soluzione si può ricavare a ritroso a partire da SPA[n, $m/3$, $m/3$]. La procedura ha complessità pseudo-polinomiale $O(nm^2)$.

```
procedure BOTTINO(m, n: integer; var prezzo: vettore; var SPA: matrice);
    var i, j, h: integer;
    begin
        for i := 1 to n do
            for j := −m to (m div 3) do
                for h := −m to (m div 3) do SPA[i, j, h] := false;
        SPA[1, prezzo[1], 0] := true; SPA[1, 0, prezzo[1]] := true;
        SPA[1, 0, 0] := true;
        for i := 2 to n do
            for j := 0 to (m div 3) do
                for h := 0 to (m div 3) do
                    SPA[i, j, h] := SPA[i − 1, j − prezzo[i], h] or
                        SPA[i − 1, j, h − prezzo[i]] or SPA[i − 1, j, h]
    end;
```

Capitolo 22

Algoritmi di approssimazione

Molti problemi di ottimizzazione di rilevante interesse pratico sono NP-ardui, poiché gli stessi problemi in forma decisionale sono NP-completi. Per tali problemi, c'è poca speranza di progettare un algoritmo che trovi la soluzione ottima in tempo polinomiale. In molti casi, però, il fatto di non poter ottenere la soluzione ottima non è poi così drammatico, e ci si può accontentare di algoritmi che, in tempo polinomiale, trovino una soluzione ammissibile del problema che sia abbastanza "vicina" a quella ottima. Se è possibile dimostrare che la soluzione ammissibile fornita dall'algoritmo si discosta, nel caso pessimo o in quello medio, dalla soluzione ottima per uno scarto (o errore) relativo limitato per tutti i possibili dati di ingresso del problema, allora si ha un algoritmo di "approssimazione".

In questo capitolo, vedremo algoritmi di approssimazione assoluta per il COMMESSO VIAGGIATORE CON DISEGUAGLIANZE TRIANGOLARI e lo SCHEDULING MULTIPROCESSORE, dove lo scarto relativo è limitato da una costante che dipende solo dall'algoritmo. Vedremo inoltre un algoritmo di ϵ-approssimazione per la SOMMA DI SOTTOINSIEME, dove lo scarto relativo ϵ viene fornito in ingresso all'algoritmo stesso e può essere reso piccolo a piacere, al prezzo di un sempre maggiore sforzo computazionale. Vedremo infine come non tutti i problemi NP-ardui ammettano algoritmi di approssimazione.

22.1 Approssimazione assoluta

Dato un problema di ottimizzazione P, con funzione costo non negativa c, un algoritmo di approssimazione assoluta fornisce una soluzione ammissibile x' il cui costo si discosta da quello della soluzione ottima x^* per uno scarto relativo garantito e limitato da una costante. Formalmente, occorre che

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq e$$

per una qualche costante e non negativa che dipende dall'algoritmo e per tutti i possibili dati d'ingresso. In altri termini, l'algoritmo garantisce che la percentuale di scostamento dalla soluzione ottima non superi una certa costante. Tale scostamento vale per la medesima costante e per tutti gli input! Ad esempio, se $e = 1$, allora lo scostamento dall'ottimo non supera il 100%, cioè $c(x')$ è al più il doppio di $c(x^*)$, qualora il problema sia di minimizzazione, oppure almeno la metà, nel caso di massimizzazione.

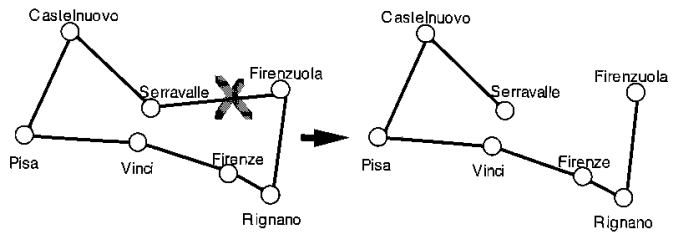


Figura 22.1: Cancellando un arco da un circuito Hamiltoniano si ottiene una catena Hamiltoniana che è anche un albero di copertura.

22.1.1 Commesso viaggiatore con diseguaglianze triangolari

Vediamo come sia possibile progettare un algoritmo di approssimazione assoluta per una variante semplificata del problema del COMMESSO VIAGGIATORE:

COMMESSO VIAGGIATORE CON DISEGUAGLIANZE TRIANGOLARI (ΔTSP): "Date n città e le distanze d_{ij} tra esse, tali che $d_{ij} \leq d_{ik} + d_{kj}$ per $1 \leq i, j, k \leq n$, trovare un percorso che, partendo da una qualsiasi città, attraversi ogni città esattamente una volta e ritorni alla città di partenza, in modo che la distanza complessiva percorsa sia minima."

La versione decisionale di ΔTSP è NP-completa. Infatti, nella dimostrazione di NP-completezza dell'usuale COMMESSO VIAGGIATORE (Teorema 19.5) sono state usate distanze $d_{ij} \in \{1, 2\}$, per $1 \leq i, j \leq n$, che ovviamente verificano la proprietà di triangolarità.

Per semplicità, ΔTSP può essere reinterpretato come il problema di trovare un circuito Hamiltoniano di costo minimo in un grafo K_n di n nodi, non orientato, pesato e completo, cioè formato da una cricca di n nodi.

Come mostrato nella Fig. 22.1, un qualsiasi circuito Hamiltoniano di K_n in cui si cancelli un arco è un particolare albero di copertura per esso e quindi un minimo albero di copertura per K_n non può avere un costo superiore alla soluzione ottima di ΔTSP.

Se si individua un minimo albero di copertura e se ne percorrono gli archi due volte, la prima volta in un senso e l'altra nel senso opposto, si può visitare ciascuna città almeno una volta e la distanza complessiva percorsa non supera il doppio della distanza ottima di ΔTSP. Visitando le città in modo da "saltare" una città già visitata, si ottiene, per la triangolarità delle distanze, un circuito Hamiltoniano x' il cui costo $c(x')$ non supera il doppio del costo $c(x^*)$ del circuito Hamiltoniano minimo x^* per ΔTSP. Un esempio è fornito nella Fig. 22.2.

Questo algoritmo di approssimazione richiede $O(n^2 \log n)$ tempo, poiché il minimo albero di copertura può essere trovato con l'algoritmo di Kruskal (cfr. § 13) e la soluzione x' si può ottenere considerando l'albero ordinato in modo arbitrario e poi visitandone i nodi in ordine anticipato (cfr. § 4).

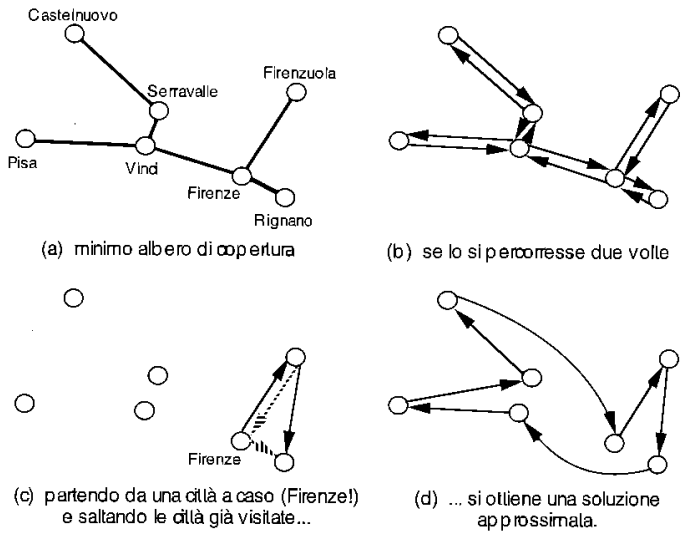


Figura 22.2: Algoritmo di approssimazione assoluta per il COMMESSO VIAGGIATORE CON DISEGUAGLIANZE TRIANGOLARI (ΔTSP).

L'algoritmo fornisce una soluzione in cui la distanza complessiva percorsa non supera del 100% quella minima. Infatti, si ha:

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq \frac{|2c(x^*) - c(x^*)|}{|c(x^*)|} \leq 1.$$

È possibile individuare dati di ingresso particolarmente "perversi" per i quali l'algoritmo produce soluzioni che raggiungono lo scarto pessimo del 100%. Per esempio, si considerino $2n$ città che si trovino sui vertici di due poligoni regolari concentrici, ciascuno di n lati, con distanze pari alle distanze Euclidee tra tali vertici. Un esempio per $n = 6$ è mostrato nella Fig. 22.3a), dove r ed $r + d$ sono i raggi dei cerchi circoscritti, rispettivamente, al poligono interno e a quello esterno. Il minimo albero di copertura è strutturato come mostrato nella Fig. 22.3b) e percorrendo tale albero due volte, si ottiene il percorso illustrato nella Fig. 22.3c). L'algoritmo di approssimazione può trovare, nel caso pessimo, la soluzione x' mostrata nella Fig. 22.3d), che è assai diversa da quella ottima x^* della Fig. 22.3e). Se L ed L' sono i lati, rispettivamente, del poligono interno e di quello esterno, allora:

$$c(x') = (n-1)(L + L') + 2d \quad \text{e} \quad c(x^*) = n(L + L')/2 + nd.$$

Ricordiamo che, se R è il raggio del cerchio circoscritto ad un poligono regolare di n lati, allora la lunghezza di ogni lato è $2R\,\mathrm{sen}(\pi/n)$. Assumendo che il poligono

interno abbia raggio $r = 1$ ed il poligono esterno abbia raggio $r + d = 1 + 1/n^2$, si ottiene:

$$\begin{aligned} L &= 2\,\mathrm{sen}(\pi/n), \\ L' &= 2(1 + 1/n^2)\,\mathrm{sen}(\pi/n), \\ c(x') &= 2(n-1)(2 + 1/n^2)\,\mathrm{sen}(\pi/n) + 2/n^2, \\ c(x^*) &= n(2 + 1/n^2)\,\mathrm{sen}(\pi/n) + 1/n. \end{aligned}$$

Essendo:

$$\lim_{n\to\infty} n\,\mathrm{sen}(\pi/n) = \pi$$

si ottengono:

$$\lim_{n\to\infty} c(x') = 4\pi$$

$$\lim_{n\to\infty} c(x^*) = 2\pi$$

e pertanto, al limite, $c(x')$ tende ad essere il doppio di $c(x^*)$.

I particolari dati di ingresso visti precedentemente, mostrano come lo scarto pessimo $|c(x') - c(x^*)| \,/\, |c(x^*)| \leq 1$ sia raggiungibile e quindi non migliorabile per il particolare algoritmo di approssimazione esposto. Questo però non deve fare erroneamente credere che tale algoritmo sia il migliore algoritmo di approssimazione possibile per il ΔTSP, poiché si potrebbe dimostrare uno scarto inferiore ad 1 per un altro algoritmo. In effetti, è noto un algoritmo di approssimazione più astuto, dovuto a Christofides (1976), che fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \leq 3c(x^*)/2$, cioè che non si scosta più del 50% da quella minima.

22.1.2 Scheduling multiprocessore

Si consideri adesso la versione di ottimizzazione del problema SCHEDULING MULTIPROCESSORE, la cui versione decisionale è stata dimostrata essere NP-completa nel Teorema 18.6:

SCHEDULING MULTIPROCESSORE: "Dati un insieme $P = \{p_1, \ldots, p_n\}$ di "programmi", tali che p_i richiede un "tempo (intero) di esecuzione" t_i, ed m "processori" identici, eseguire tutti i programmi sui processori nel più breve tempo possibile."

In questo problema l'esecuzione dei programmi non può essere interrotta (cioè frazionata in più porzioni come in REAL-TIME SCHEDULING). Un algoritmo di approssimazione per lo SCHEDULING MULTIPROCESSORE, noto come algoritmo LPT (per "Largest Processing Time") si basa sulla tecnica greedy e consiste nell'ordinare i programmi per tempi di esecuzione decrescenti, considerare sequenzialmente ciascun programma, ed assegnarlo al primo processore "libero".

Esempio 22.1 (Algoritmo LPT). Siano $m = 2$, $n = 3$, $t_1 = 5$, $t_2 = 4$, $t_3 = 3$. I programmi sono già ordinati per tempi di esecuzione decrescenti, e sono quindi considerati nell'ordine: p_1, p_2, p_3. Si assuma che se entrambi i processori sono liberi

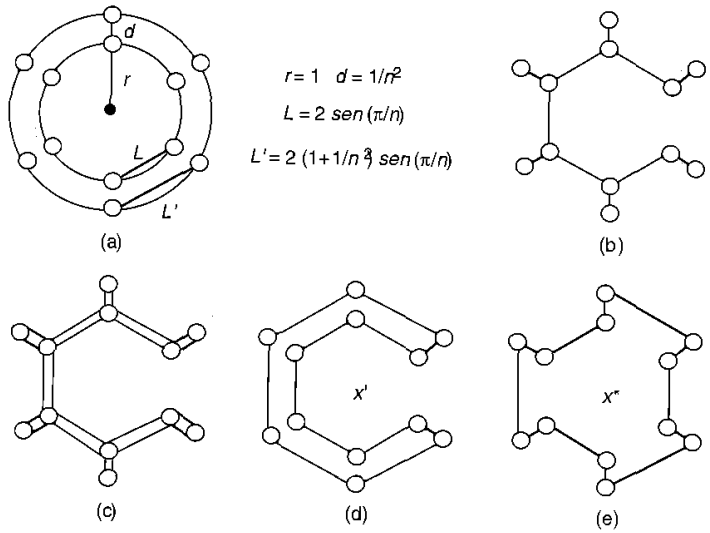


Figura 22.3: Caso pessimo per ΔTSP. (a) Le $2n$ città. (b) Il minimo albero di copertura. (c) L'albero percorso due volte. (d) Soluzione approssimata x', avente costo $c(x') = (n-1)(L+L') + 2d$. (e) Soluzione ottima x^*, avente costo $c(x^*) = n(L+L')/2 + nd$.

allo stesso istante (come accade per esempio all'istante iniziale), allora il programma sia assegnato al processore di indice minore. Il programma p_1 è così assegnato al processore P_1, indi p_2 è assegnato a P_2, e infine p_3 è assegnato a P_1 (perché P_2 si libera prima di P_1). La soluzione x' fornita dall'algoritmo LPT, la cui lunghezza è $c(x') = 7$, è mostrata nella Fig. 22.4. □

L'algoritmo LPT richiede tempo $O(n \log n)$ e produce una soluzione approssimata x' la cui lunghezza $c(x')$ si discosta dalla lunghezza minima $c(x^*)$ secondo la seguente formula:

$$\frac{c(x')}{c(x^*)} \leq \frac{4}{3} - \frac{1}{3m}.$$

Per dimostrare questo risultato, si proceda per assurdo. Si supponga l'esistenza di un insieme P di n programmi, con $t_1 \geq t_2 \geq \cdots \geq t_n$, tali che $c(x')/c(x^*) > 4/3 - 1/(3m)$. Senza perdere in generalità, si assuma che ci sia un solo programma che termina all'istante $c(x')$ e che questo sia proprio p_n. Infatti, se si avesse $p_i = c(x')$ per $i < n$, allora, considerando l'insieme $Q = \{p_1, \ldots, p_i\}$, si avrebbe una nuova soluzione approssimata y' con $c(y') = c(x')$, mentre la soluzione ottima y^* per Q sicuramente verificherebbe $c(y^*) \leq c(x^*)$, e quindi risulterebbe $c(y')/c(y^*) \geq$

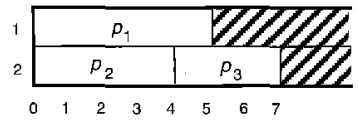


Figura 22.4: Algoritmo LPT.

$c(x')/c(x^*) > 4/3 - 1/(3m)$. Chiaramente

$$c(x^*) \geq \frac{1}{m} \sum_{i=1}^{n} t_i.$$

Detto s_n l'istante in cui inizia l'esecuzione di p_n nella soluzione x', si ha:

$$\sum_{i=1}^{n-1} t_i \geq m s_n, \quad \text{e} \quad c(x') = s_n + t_n.$$

Pertanto, si ottiene:

$$\begin{aligned} \frac{c(x')}{c(x^*)} &= \frac{s_n + t_n}{c(x^*)} \leq \\ &\leq \frac{t_n}{c(x^*)} + \frac{1}{mc(x^*)} \sum_{i=1}^{n-1} t_i = \\ &= \frac{(m-1)t_n}{mc(x^*)} + \frac{1}{mc(x^*)} \sum_{i=1}^{n} t_i \leq \\ &\leq \frac{(m-1)t_n}{mc(x^*)} + 1. \end{aligned}$$

Poiché $c(x')/c(x^*) > 4/3 - 1/(3m)$ per ipotesi, ricavando t_n dalla relazione

$$\frac{(m-1)t_n}{mc(x^*)} + 1 > \frac{4}{3} - \frac{1}{3m}$$

si giunge alla conclusione che $t_n > c(x^*)/3$. Essendo $t_n \leq t_i$ per $1 \leq i \leq n$, se $c(x')/c(x^*) > 4/3 - 1/(3m)$, allora in una soluzione ottima x^* nessun processore può eseguire più di due programmi. Considerando una soluzione x^* siffatta, è possibile dimostrare, effettuando gli scambi tra coppie di programmi mostrati nella Fig. 22.5, che da x^* è ottenibile una nuova soluzione ottima y^* con $c(y^*) = c(x^*)$ in cui i programmi sono eseguiti secondo la regola dell'algoritmo LPT. Ma ciò implica

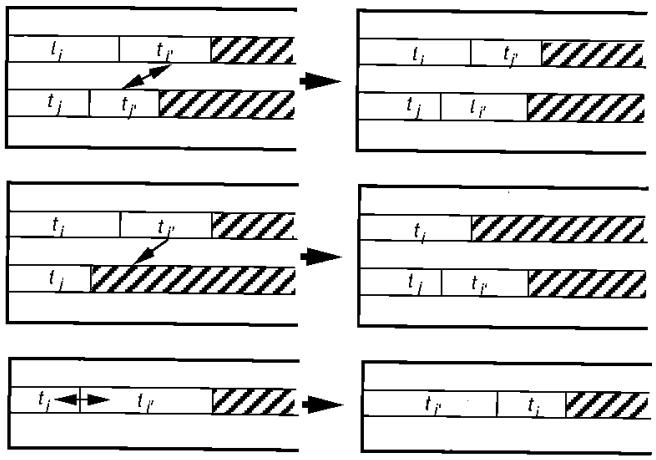


Figura 22.5: Scambi per trasformare x^* in y^*.

che $c(x') = c(y^*)$, che è una contraddizione poiché da $c(x')/c(x^*) > 4/3 - 1/(3m)$ discende che $c(x') > c(x^*) = c(y^*)$.

Dato m, lo scarto pessimo può essere raggiunto dal seguente insieme P di $n = 2m+1$ programmi: $t_1 = t_2 = 2m-1, t_3 = t_4 = 2m-2, t_5 = t_6 = 2m-3, \ldots, t_n-4 = t_n - 3 = m + 1$, $t_n - 2 = t_n - 3 = t_n = m$. Come esemplificato nella Fig. 22.6, $c(x^*) = 3m$, mentre $c(x') = 4m - 1$.

22.1.3 Non approssimabilità

Purtroppo, per molti problemi di ottimizzazione, trovare una soluzione approssimata è tanto difficile quanto trovarne una ottima! Questo è vero, per esempio, per le versioni di ottimizzazione di CRICCA, COLORAZIONE e COMMESSO VIAGGIATORE (senza diseguaglianze triangolari). A titolo di esempio, vediamo il seguente risultato.

Teorema 22.1. *Non esiste alcun algoritmo di approssimazione assoluta per il* COMMESSO VIAGGIATORE *tale che* $c(x') \leq sc(x^*)$, *con* s *intero positivo, a meno che non sia* $\mathbb{P} = \mathbb{NP}$.

Dimostrazione. Per assurdo, si assuma l'esistenza di un algoritmo polinomiale A che fornisca una soluzione x' con $c(x') \leq sc(x^*)$, per un s intero positivo. Si consideri la riduzione CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE utilizzata nel Teorema 18.5 e si ponga:

$$d_{ij} = \begin{cases} 1, & \text{se } [i,j] \in A, \\ sn, & \text{se } [i,j] \notin A. \end{cases}$$

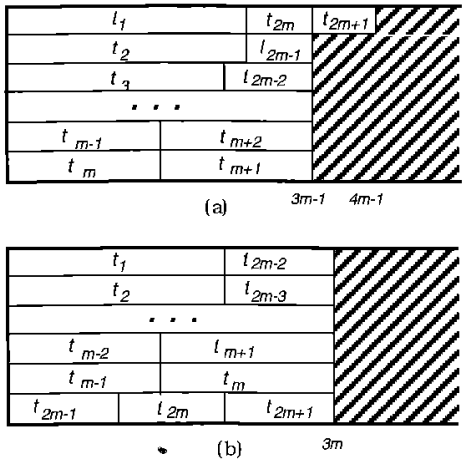


Figura 22.6: (a) Soluzione prodotta da LPT. (b) Soluzione ottima.

Per costruzione, la soluzione ottima x^* del COMMESSO VIAGGIATORE è tale che $c(x^*) \leq sn$ se e solo se il grafo G ha un circuito Hamiltoniano.

Si possono presentare due casi. Se G ha un circuito Hamiltoniano, allora $c(x^*) = n$, e quindi A fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \leq sc(x^*) = sn$. Se invece G non ha alcun circuito Hamiltoniano, allora $c(x^*) > sn$, ed A fornisce, in tempo polinomiale, una soluzione x' per cui $c(x') \geq c(x^*) > sn$.

Pertanto, A restituisce una soluzione x' con $c(x') \leq sn$ se e solo se G ha un circuito Hamiltoniano. Questa è chiaramente una contraddizione, perché A risolverebbe in tempo polinomiale un problema NP-completo, il che è impossibile, a meno che $\mathbb{P}$ non coincida con $\mathbb{NP}$. □

22.2 ϵ-approssimazione

In taluni casi, è possibile progettare algoritmi di approssimazione che producono soluzioni approssimate il cui scarto dall'ottimo non è una costante che dipende dall'algoritmo (come nell'approssimazione assoluta), bensì un valore fornito come dato di input che può essere reso arbitrariamente piccolo. In altri termini, comunque si scelga un ϵ piccolo a piacere, con $0 < \epsilon \leq 1$, tali algoritmi producono in tempo polinomiale nella dimensione del problema, ma in genere esponenziale in $1/\epsilon$, una soluzione ammissibile x' tale che:

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq \epsilon$$

Algoritmi di questo tipo sono detti algoritmi di ϵ-approssimazione. Si consideri ad esempio la versione di ottimizzazione della SOMMA DI SOTTOINSIEME:

SOMMA DI SOTTOINSIEME: "Dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi ed un intero positivo k, trovare un sottoinsieme S di $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i \leq k$ e $\sum_{i \in S} a_i$ sia massima."

Sia $\epsilon = 1/(h+1)$, con h intero maggiore o uguale ad uno (o, equivalentemente, $h = (1-\epsilon)/\epsilon$), e si supponga di memorizzare l'insieme A in un vettore di interi. Il problema può essere risolto con il seguente algoritmo ϵ-approssimato, basato sulla tecnica greedy, che genera dapprima nel ciclo (1) tutti i sottoinsiemi S di A di cardinalità al più h, e successivamente considera nel ciclo (2) ciascun elemento a_j di A, $1 \leq j \leq n$, aggiungendolo ad S qualora non sia già presente in S e non venga superata la soglia k.

```
procedure APPRSOTTINSIEME(var A: vettore; n, k, h: integer);
    var S, SOL: insieme; j, somma, z: integer;
    begin
        {ordina A in modo che A[1] ≥ A[2] ≥ ··· ≥ A[n]};
        SOL := ∅; somma := 0;
(1)     for ciascun sottoinsieme S di {1, ..., n} tale che |S| ≤ h do begin
            z := Σ_{i∈S} A[i];
            if z ≤ k then begin
(2)             for j := 1 to n do
                    if (j ∉ S) and (z + A[j] ≤ k) then begin
                        S := S ∪ {j};
                        z := z + A[j]
                    end;
                if z > somma then begin
                    SOL := S;
                    somma := z
                end
            end
        end
    end;
```

Poiché, fissato un $m>0$, ci sono $n(n-1)\cdots(n-m+1)/m! \leq n^m$ sottoinsiemi di $\{1, \ldots, n\}$ di cardinalità m, il ciclo (1) è ripetuto $\sum_{0 \leq m \leq h} n^m \leq hn^h + 1$ volte. Il ciclo (2), invece, è ripetuto n volte. Pertanto, la complessità dell'algoritmo APPRSOTTINSIEME è $O(hn^{h+1})$, che è polinomiale in n, ma esponenziale in h (e quindi in $1/\epsilon$). Fissato un h, comunque, la complessità diviene polinomiale.

APPRSOTTINSIEME fornisce una soluzione $x' =$ SOL di costo $c(x') = somma$ tale che $|c(x') - c(x^*)| \,/\, |c(x^*)| \leq \epsilon$. Per dimostrare questo risultato, sia $x^* = B$ il sottoinsieme ottimo di A con

$$|B| = m, B = \{b_1, b_2, \ldots, b_m\}, b_1 \geq b_2 \geq \cdots \geq b_m, \text{ e } c(x^*) = \sum_{1 \leq i \leq m} b_i.$$

Se $m \leq h$, allora l'insieme B è stato considerato nel ciclo (1) e APPRSOTTINSIEME ha fornito la soluzione ottima. Se invece $m > h$, essendo $c(x^*) = \sum_{1 \leq i \leq m} b_i$, si ha che $b_i \leq c(x^*)/i$ per $1 \leq i \leq h$, e quindi $b_i \leq c(x^*)/(h+1)$ per $h < i \leq m$.

In tal caso, la procedura APPRSOTTINSIEME avrà considerato il sottoinsieme $B' = \{b_1, b_2, \ldots, b_h\}$ nel ciclo (1). Sia

$b_i = \max\{b \in B$: APPRSOTTINSIEME non ha aggiunto b a B' nel ciclo (2)$\}$.

Se tale b_i non esiste, APPRSOTTINSIEME ha trovato la soluzione ottima. Altrimenti,

$$c(x^*) - c(x') \leq b_i \leq \frac{c(x^*)}{h+1}$$

e quindi, dividendo per $c(x^*)$, si ottiene

$$\frac{|c(x^*) - c(x')|}{|c(x^*)|} \leq \frac{1}{h+1} = \epsilon.$$

Esempio 22.2 (APPRSOTTINSIEME). Siano $n = 8$, $k = 45$, $A = \{23, 19, 13, 12, 11, 8, 7, 5\}$, $h = 1$ (e quindi $\epsilon = 1/2$). La seguente tabella illustra la computazione di APPRSOTTINSIEME:

$\|S\|$	insieme costruito in (1)	insieme costruito in (2)	z
0	$\emptyset$	$\{23, 19\}$	42
1	$\{23\}$	$\{23, 19\}$	42
1	$\{19\}$	$\{23, 19\}$	42
1	$\{13\}$	$\{23, 13, 8\}$	44
1	$\{12\}$	$\{23, 12, 8\}$	43
1	$\{11\}$	$\{23, 11, 8\}$	42
1	$\{8\}$	$\{23, 13, 8\}$	44
1	$\{7\}$	$\{23, 13, 7\}$	43
1	$\{5\}$	$\{23, 13, 5\}$	41

APPRSOTTINSIEME dà come risultato SOL $= x' = \{23, 13, 8\}$ con $somma = c(x') = 44$. Si noti che la soluzione ottima $B = x^* = \{19, 13, 8, 5\}$, con $c(x^*) = k = 45$, sarebbe stata trovata dall'algoritmo scegliendo $h = 2$. □

Ovviamente, la situazione ideale sarebbe quella di poter sempre progettare un algoritmo A di ϵ-approssimazione per un problema P che producesse una soluzione ϵ-approssimata in tempo polinomiale sia nella dimensione del problema sia in $1/\epsilon$. Purtroppo, tali algoritmi sono noti soltanto per rari problemi di ottimizzazione che coinvolgono numeri, come ad esempio la SOMMA DI SOTTOINSIEME, per i quali esistono già algoritmi che trovano la soluzione ottima in tempo pseudo-polinomiale. Un forte risultato negativo, a questo riguardo, è dato dal seguente teorema, che diamo senza dimostrazione.

Teorema 22.2. *Sia P un problema di ottimizzazione tale che:*

1. *la versione decisionale di P è fortemente NP-completa;*
2. *per ogni insieme I di dati di input per P il valore $c(x^*)$ della soluzione ottima è limitato superiormente da $p(\#)$, dove p è un polinomio e $\#$ è il più grande numero che compare in I.*

Allora, a meno che $\mathbb{P}$ non coincida con $\mathbb{NP}$, non esiste alcun algoritmo di ϵ-approssimazione per P che richieda tempo polinomiale sia nella dimensione del problema che in $1/\epsilon$. □

22.3 Esercizi

Esercizio 22.1 (Minima copertura con nodi). Dato un grafo non orientato G, il problema della MINIMA COPERTURA CON NODI chiede di trovare un sottoinsieme S di nodi avente minima cardinalità tale che ciascun arco abbia almeno un estremo in S. Si progetti un algoritmo di approssimazione assoluta tale che $|c(x') - c(x^*)|/|c(x^*)| \leq 1$ (Suggerimento: si scandiscano gli archi di G).

Esercizio 22.2 (Bin packing). Dato un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi (i "volumi" di n "oggetti") ed un intero positivo k (la "capacità" di una "scatoletta") trovare una partizione di $\{1, \ldots, n\}$ nel minimo numero di sottoinsiemi disgiunti (le "scatolette") tali che $\sum_{i \in S} a_i \leq k$ per ogni insieme S della partizione. Si consideri l'algoritmo FF ("First-Fit") in cui gli oggetti sono considerati in un ordine qualsiasi e ciascun oggetto è assegnato alla prima scatoletta che lo può contenere. Si dimostri che FF è un algoritmo di approssimazione assoluta tale che $|c(x') - c(x^*)| \,/\, |c(x^*)| < 1$.

Esercizio 22.3 (ΔTSP Euclideo). Si assuma che le n città del ΔTSP siano punti nel piano e che le distanze tra le città siano le distanze Euclidee tra i punti. Si dimostri che un percorso ottimo del commesso viaggiatore non incrocia mai se stesso.

Esercizio 22.4 (Zaino)). Si fornisca un algoritmo ϵ-approssimato per il problema dello ZAINO in forma di ottimizzazione.

Capitolo 23

Algoritmi branch-&-bound

Per risolvere un problema di ottimizzazione NP-arduo, si può modificare la procedura ENUMERAZIONE, vista nel § 16, in modo da evitare di effettuare certe sequenze di scelte che si rivelino incapaci di generare la soluzione ottima.

Si assuma che il problema da risolvere sia di minimo, che ogni sequenza di scelte abbia costo non negativo, e che ogni scelta, aggiunta alle scelte già effettuate, non faccia diminuire il costo della soluzione parziale così costruita. Si supponga di avere a disposizione una opportuna funzione "lower bound" LB(S), che dipenda dalla sequenza di scelte fatte $S = S[1], \ldots, S[i]$ e garantisca che tutte le soluzioni ammissibili generabili aggiungendo nuove scelte a tale sequenza abbiano costo maggiore o uguale a LB(S). Allora, è possibile mantenersi la migliore soluzione ammissibile MINSOL tra quelle finora esaminate, insieme al suo costo MINCOST, e valutare LB(S) ogniqualvolta è effettuata una nuova scelta $S[i]$ in un nodo dell'albero delle scelte. Se LB(S) è maggiore o uguale a MINCOST, allora si può evitare di generare ed esplorare il sottoalbero delle scelte radicato in tal nodo (si veda la Fig. 23.1).

Questo metodo, detto branch-&-bound, non migliora la complessità della procedura ENUMERAZIONE, che resta superpolinomiale nel caso pessimo, ma ne abbassa drasticamente il tempo di esecuzione in pratica. Ciò vale in special modo se si riesce ad individuare una "buona" funzione LB, che sia cioè il più vicino possibile al costo della soluzione ottima.

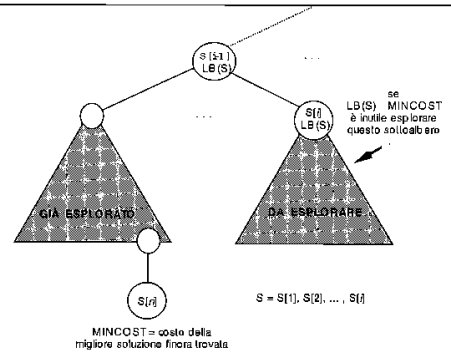


Figura 23.1: Esplorazione "intelligente" dell'albero delle scelte.

Supponiamo per semplicità che ogni scelta $S[i]$ avvenga su un sottoinsieme I_i e che il numero di scelte per individuare una soluzione sia limitato da n. Sia $c(S)$ il costo della sequenza di scelte $S = S[1], \ldots, S[i]$. Inoltre, siano MINSOL e MINCOST parametri globali inizializzati a $\emptyset$ e $+\infty$. Allora, una possibile struttura generale di un algoritmo BRANCH&BOUND, da chiamarsi con parametro $i = 1$, è la seguente:

```
procedure BRANCH&BOUND(i: integer);
    {opportune dichiarazioni};
    begin
        {determina I_i in funzione di S[1], ... , S[i − 1]};
        if I_i ≠ ∅ then
            for j := 1 to |I_i| do begin
                S[i] := j-esimo elemento di I_i;
                {calcola il "lower bound" LB(S)};
                if LB(S) < MINCOST then
                    if i < n then
                        BRANCH&BOUND(i + 1)
                    else
                        if c(S) < MINCOST then begin
                            MINSOL := S;
                            MINCOST := c(S)
                        end;
            end;
    end;
```

Ovviamente, se $|I_i| \leq m$, per $1 \leq i \leq n$, ed il calcolo di LB(S) richiede $O(f(n))$ tempo, allora la complessità di BRANCH&BOUND è $O(m^n f(n))$. Tale procedura è stata illustrata in forma ricorsiva per ragioni di chiarezza e concisione, ma può essere trasformata in una iterativa equivalente utilizzando le regole di trasformazione definite nel § 3. In pratica, il tempo di computazione di una qualsiasi procedura di tipo "branch-&-bound" è influenzato da svariati fattori:

1. *Soluzione di partenza.* Anziché partire con MINCOST $= +\infty$, è in genere più conveniente determinare il costo di una soluzione ammissibile del problema ed inizializzare MINCOST al valore di tale costo;
2. *Branch.* Prestazioni migliori si possono ottenere variando la cosiddetta regola di branch, ovvero il criterio per determinare le scelte da effettuare e generare i figli di un nodo nell'albero delle scelte;
3. *Bound.* La "migliore" regola per calcolare LB è quella che dà il lower bound più stretto, cioè più vicino possibile al costo della soluzione ottima, ma che sia nel contempo calcolabile velocemente;
4. *Ordine di visita.* L'ordine più usato per generare e visitare l'albero delle scelte è senza dubbio quello anticipato (LIFO) precedentemente esposto, perché richiede meno spazio; infatti, utilizzando una pila, il massimo numero di "record di attivazione" [cfr. § 3] è proporzionale alla lunghezza di un percorso radice-foglia, che è $O(n)$; talvolta, sono usati criteri diversi, in cui l'albero è visitato per livelli (FIFO) o utilizzando una priorità (p.e. considerando prima quel figlio avente il più promettente lower bound);
5. *Scarto dall'ottimo.* Anziché raggiungere la soluzione ottima a tutti i costi, l'esecuzione può essere interrotta quando lo scarto dall'ottimo scenda sotto una certa soglia prestabilita, cioè quando (MINCOST − LB*)/LB* $\leq e$, con e costante prefissata ed LB* uguale al più piccolo lower bound calcolato per i nodi generati ma non ancora visitati dell'albero delle scelte.

Vediamo un paio di esempi di algoritmi "branch-&-bound" per risolvere due problemi computazionalmente intrattabili. Entrambi gli algoritmi sono specializzazioni dell'algoritmo generale BRANCH&BOUND appena introdotto. Il primo algoritmo risolve il problema del COMMESSO VIAGGIATORE (TSP), mentre il secondo algoritmo è progettato per risolvere un rilevante problema di trasmissione di dati per mezzo di una coppia di satelliti geostazionari tra loro intercomunicanti.

23.1 Commesso viaggiatore

Consideriamo il problema del COMMESSO VIAGGIATORE (senza diseguaglianze triangolari), per il quale sappiamo che non può esistere né un algoritmo ottimo pseudo-polinomiale né un algoritmo di approssimazione assoluta polinomiale (a meno che $\mathbb{P} = \mathbb{NP}$). Cerchiamo dunque di specializzare la procedura BRANCH&BOUND vista precedentemente.

Sia n il numero di città e $d[h, k]$ la distanza, intera non negativa, tra le città h e k, $1 \leq h, k \leq n$. Si assuma che al passo i-esimo siano state fatte le scelte $S = S[1], S[2], \ldots, S[i]$, con S sottoinsieme di $\{1, 2, \ldots, n\}$. Ovviamente, un qualsiasi percorso ammissibile del commesso viaggiatore generabile "espandendo" S deve attraversare le città $S[1], S[2], \ldots, S[i]$ in quest'ordine, passare da $S[i]$ ad una qualsiasi delle rimanenti $n - i$ città, attraversare queste ultime città in un ordine qualsiasi, e da una di queste ritornare ad $S[1]$. Si pongano:

$$
\begin{aligned}
A &= \min_h \{d[S[1], h] : h \notin S\}, \\
B &= \min_h \{d[S[i], h] : h \notin S\}, \\
C[i] &= C[i-1] + d[S[i-1], S[i]], \\
C[1] &= 0, \\
D[h] &= \min_{p,q} \{d[h, p] + d[h, q] : h \neq p \neq q\}, \text{ per ogni } h \notin S.
\end{aligned}
$$

Allora, un lower bound LB(S) è dato dalla seguente espressione:

$$
\begin{aligned}
\text{LB}(S) &= C[i] + \left\lceil (A + B + \sum_{h \notin S} D[h])/2 \right\rceil, && \text{se } i < n, \\
\text{LB}(S) &= C[i] + d[S[i], S[1]], && \text{se } i = n.
\end{aligned}
$$

Infatti, $C[i]$ è la distanza complessiva per attraversare, nell'ordine, le città $S[1]$, $S[2]$, ..., $S[i]$. Invece, A è un lower bound della distanza tra una qualsiasi delle rimanenti $n - i$ città ed $S[1]$, mentre B è un lower bound della distanza tra $S[i]$ ad una qualsiasi di tali $n - i$ città. Infine, $D[h]$ è un lower bound della distanza percorsa per attraversare una qualsiasi di queste ultime $n - i$ città, provenendo da (e dirigendosi verso) un'altra di queste $n - i$ città.

Esempio 23.1 (Lower bound per TSP). Si consideri la matrice delle distanze fornita nella Fig. 23.2 per $n = 5$ città. Siano $i = 2$, $S[1] = 1$ e $S[2] = 3$. Allora $A = \min\{d[1, 2], d[1, 4], d[1, 5]\} = \min\{3, 2, 7\} = 2$; $B = \min\{d[3, 2], d[3, 4], d[3, 5]\} = \min\{4,$

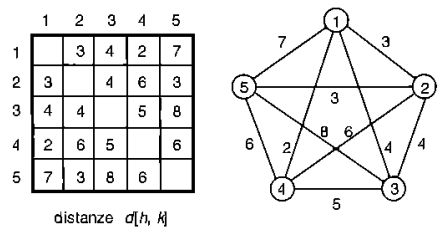


Figura 23.2: Dati di ingresso per il COMMESSO VIAGGIATORE (TSP).

$5, 8\} = 4$; $C[2] = C[1] + d[1, 3] = 4$; $D[2] = 3 + 3 = 6$; $D[4] = 2 + 5 = 7$; $D[5] = 3 + 6 = 9$; $\text{LB}(S) = 4 + \lceil(2 + 4 + 6 + 7 + 9)/2\rceil = 18$. □

Per non generare più volte lo stesso percorso a partire da tutte le città che lo compongono, la prima scelta può essere fatta una volta per tutte nel programma chiamante. Assumendo per esempio $S[1] = 1$, la seguente procedura ricorsiva B&B-TSP può essere richiamata direttamente con parametro $i = 2$, dopo aver inizializzato $C[1]$ ed LB a zero.

```
procedure B&B-TSP(i: integer);
    {opportune dichiarazioni};
    begin
        for each j ∉ S = S[1], ... , S[i − 1] do begin
            S[i] := j;
            C[i] := C[i − 1] + d[S[i − 1], S[i]];
            {calcola A, B, e D[h] per ogni h ∉ S};
            if i < n
                then LB := C[i] + ⌈(A + B + Σ_{h∉S} D[h])/2⌉
                else LB := C[i] + d[S[i], S[1]];
            if LB < MINCOST then
                if i < n
                thenB&B-TSP(i + 1)
                else begin
                    C[n] := LB;
                    MINSOL := S;
                    MINCOST := C[n]
                end;
        end
    end;
```

Per risparmiare tempo, anziché partire con $\text{MINCOST} = +\infty$, si può partire inizializzando MINCOST al valore di una qualsiasi soluzione ammissibile, cioè di una permutazione di $1..n$. Una tale permutazione può essere generata casualmente in tempo $O(n)$ avvalendosi della funzione RANDOM introdotta nel § 20. Infatti, si consideri un vettore $V[1..n]$ inizializzato in modo che $V[i] = i$ per $i = 1, 2, \ldots, n$. Allora, una permutazione delle n città può essere posta in un secondo vettore $P[1..n]$ come segue:

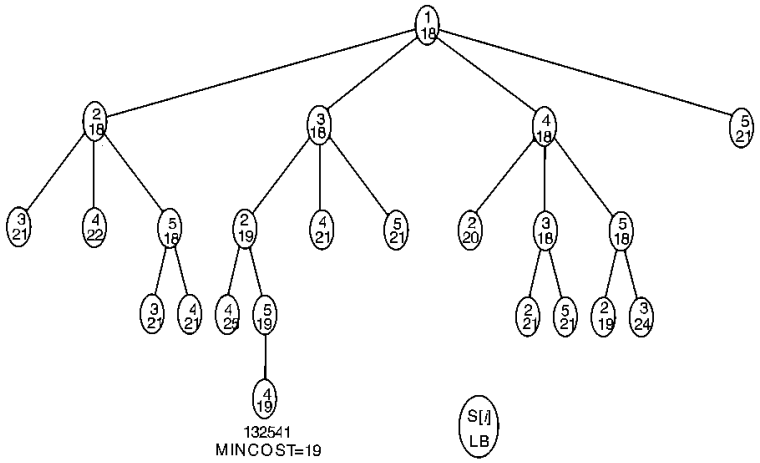


Figura 23.3: Albero delle scelte generato dalla procedura B&B-TSP per i dati della Fig. 23.2 con MINCOST inizializzato a 21.

```
    for i := 1 to n do begin
        j := RANDOM(i, n);
        P[i] := V[j];
        V[j] := V[i]
    end;
```

Esempio 23.2 (B&B-TSP). Sia 4, 3, 2, 5, 1 la permutazione casuale generata per $n = 5$ città. Questa corrisponde al circuito 1, 4, 3, 2, 5, 1 che, per i dati della Fig. 23.2, ha costo $2 + 5 + 4 + 3 + 7 = 21$. Si assuma pertanto di aver inizializzato MINCOST a 21. Allora, applicando la procedura B&B-TSP ai dati della Fig. 23.2, si genera l'albero delle scelte mostrato nella Fig. 23.3. Il confronto "LB < MINCOST" permette di "potare" ben 42 nodi sui 65 possibili, con un notevole risparmio del tempo totale di esecuzione. □

È bene rimarcare come la procedura B&B-TSP appena descritta non sia che un semplice esempio (e sicuramente non il migliore) di impiego della tecnica branch-&-bound per il COMMESSO VIAGGIATORE. In effetti, il branch-&-bound è una tecnica così generale che si possono progettare molte procedure diverse per lo stesso problema variandone le regole di branch, le funzioni lower bound, e l'ordine di visita dell'albero delle scelte. Per esempio, ciascun nodo dell'albero può riguardare la scelta di un arco (anziché un nodo) del grafo K_n e avere così due soli figli, corrispondenti all'inclusione oppure all'esclusione dell'arco, mentre può essere considerato l'arco la cui esclusione massimizza il lower bound. Una funzione alternativa per il lower bound può essere ottenuta con un "1-albero", dove un 1-albero per K_n è formato da un albero di copertura per i nodi $2, 3, \ldots, n$ più due archi incidenti nel nodo 1. È immediato osservare che ogni ciclo Hamiltoniano per K_n è un 1-albero, ma non viceversa, per cui il peso di un 1-albero minimo è un lower bound al costo della soluzione ottima (un 1-albero minimo si può facilmente trovare in tempo polinomiale). In pratica, uno dei migliori lower bound per il COMMESSO VIAGGIATORE è stato ottenuto da Held e

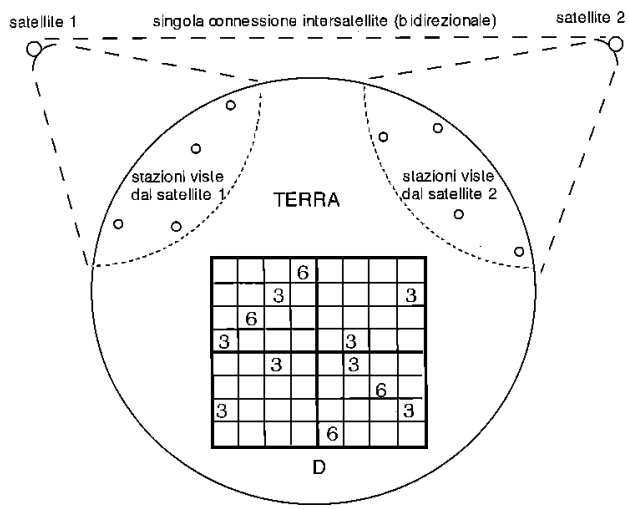


Figura 23.4: Matrice di traffico D per una coppia di satelliti che coprono 4 stazioni ciascuno.

Karp (1971) modificando ulteriormente la nozione di 1-albero. Infine, i nodi dell'albero delle scelte possono essere considerati per priorità, per esempio considerando per primo il nodo col lower bound più piccolo.

23.2 Trasmissione via satellite

Una matrice ISL di traffico $D = [d_{ij}]$ per una coppia di satelliti geostazionari, intercomunicanti per mezzo di una connessione singola (detta appunto ISL, per Inter-Satellite Link), è una matrice $2n \times 2n$ con elementi interi non negativi. L'elemento d_{ij} rappresenta l'ammontare di traffico che la stazione terrestre i deve trasmettere alla stazione terrestre j. Il numero di stazioni a terra è $2n$, che sono partizionate in due gruppi di n stazioni ciascuno. Ciascun gruppo di stazioni è visibile da un solo satellite. La stazione j è visibile dal satellite 1 se e solo se $1 \leq j \leq n$, mentre è visibile dal satellite 2 se e solo se $n + 1 \leq j \leq 2n$. Si denotano con $D(h, k)$ le due sottomatrici ISL $n \times n$ di D che rappresentano il traffico tra le stazioni visibili dal satellite h e le stazioni visibili dal satellite k.

Esempio 23.3 (Matrice ISL di traffico). La Fig. 23.4 mostra una coppia di satelliti che coprono $n = 4$ stazioni ciascuno ed una matrice ISL di traffico D di dimensioni 8×8 (le stazioni $1, \ldots, 4$ sono visibili dal satellite 1, mentre le stazioni $5, \ldots, 8$ sono visibili dal satellite 2). □

In linea di principio, tutte le $2n$ stazioni trasmittenti e riceventi possono comunicare tra loro, purché non sorgano "conflitti". Per evitare conflitti, ciascuna stazione

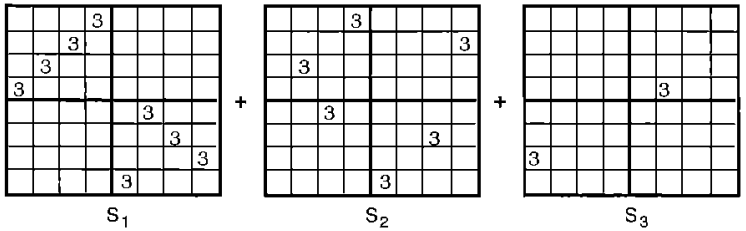


Figura 23.5: Uno schedule ottimo $S_1 + S_2 + S_3$ di lunghezza 9 per la matrice di traffico D della Fig. 23.4.

può trasmettere (ricevere) traffico verso (da) al più una stazione nello stesso istante. Inoltre, poiché la connessione tra i due satelliti è singola (ma bidirezionale), al più una stazione visibile dal primo satellite può trasmettere ad una stazione visibile dal secondo satellite (e viceversa).

Uno schedule di una matrice ISL è una scomposizione di D nella somma di "matrici di switch": $D = S_1 + S_2 + \cdots + S_m$. Ciascuna matrice di switch rappresenta il traffico che può essere trasmesso senza conflitti tra stazioni terrestri trasmittenti e riceventi e senza la necessità di riconfigurare le connessioni a bordo dei satelliti. Formalmente, una matrice di switch S_i è una matrice ISL di $2n \times 2n$ elementi uguali a 0 oppure ad un intero positivo L_i con al più un elemento uguale ad L_i in ogni riga, al più un elemento uguale ad L_i in ogni colonna, ed al più un elemento uguale ad L_i in ciascuna sottomatrice ISL. L_i è detta lunghezza della matrice di switch S_i e rappresenta le unità di tempo necessarie per completare la trasmissione di S_i. La lunghezza (o tempo di trasmissione) dello schedule $S_1, S_2, \ldots, S_m$ è $L = L_1 + L_2 + \cdots + L_m$. Pertanto, il problema della trasmissione del traffico può essere sintetizzato come segue:

SCHEDULING ISL: "Data una matrice ISL di traffico D di dimensioni $2n \times 2n$, trovarne una scomposizione $D = S_1 + S_2 + \ldots + S_m$ in una somma di matrici di switch, in modo che la lunghezza $L = L_1 + L_2 + \cdots + L_m$ sia minima".

Esempio 23.4 (Scheduling ISL). La Fig. 23.5 mostra uno schedule ottimo $S_1 + S_2 + S_3$ di lunghezza $L = L_1 + L_2 + L_3 = 3 + 3 + 3 = 9$ per la matrice D della Fig. 23.4. □

Poiché il traffico in ciascuna riga, in ciascuna colonna, e in ciascuna sottomatrice ISL deve essere trasmesso sequenzialmente, è immediato verificare che la lunghezza di un qualsiasi schedule per D non può essere inferiore a

$$B(D) = \max\{T(1,2), T(2,1), \max_{1 \leq i \leq 2n}\{r_i\}, \max_{1 \leq j \leq 2n}\{c_j\}\},$$

dove r_i è la somma di tutti gli elementi dell'i-esima riga di D, c_j è la somma di tutti gli elementi della j-esima colonna di D, e $T(h, k)$ è la somma di tutti gli elementi della sottomatrice ISL $D(h, k)$. Il valore $B(D)$ non sempre è raggiungibile da una soluzione ottima. Per esempio, un qualsiasi schedule ottimo per la matrice D della Fig. 23.4 ha lunghezza 9, mentre $B(D)$ è uguale a 6.

Vediamo come risolvere il problema dello SCHEDULING ISL con un algoritmo "branch-&-bound" B&B-ISL che utilizza la funzione B appena definita e nel quale ciascuna scelta $S[i]$ effettuata in un nodo dell'albero delle scelte consiste nella costruzione di una matrice di switch S_i.

In dettaglio, ciascun nodo dell'albero delle scelte ha associata in ingresso una matrice D_i che rappresenta il traffico rimanente non ancora trasmesso. Si assume che $D_1 = D$, la matrice ISL di traffico originaria. Ciascun nodo ha associato in ingresso anche il valore SL_i, che rappresenta la lunghezza dello schedule che, a partire da D_1, porta alla matrice ISL intermedia D_i associata al nodo. Ovviamente, si assume che $SL_1 = 0$. Effettuata la scelta di S_i, vengono calcolate la matrice di traffico successiva $D_{i+1} = D_i - S_i$ e la corrispondente lunghezza dello schedule $SL_{i+1} = SL_i + L(S_i)$, dove $L(S_i)$ denota la lunghezza della matrice di switch S_i. Inoltre, è richiamata $B(D_{i+1})$, cioè la funzione B precedentemente definita applicata alla matrice D_{i+1}. Pertanto, il lower bound LB viene calcolato come LB $= SL_{\times i+1} + B(D_{i+1})$. Uno schedule completo per la matrice originaria D è trovato quando la matrice D_{i+1} ha tutti elementi nulli.

Ad ogni nodo, la scelta S_i è effettuata generando tutte le possibili matrici di switch S' massimali della matrice di traffico D_i. Una matrice di switch massimale è una matrice di switch il cui insieme di elementi non nulli non è propriamente contenuto in un insieme di elementi non nulli di un'altra matrice di switch. Le matrici di switch massimali possono essere generate con tecniche di abbinamento bipartito basate sul flusso (cfr. § 15).

Come al solito, MINCOST è la lunghezza del migliore schedule finora generato, mentre MINSOL contiene tale schedule. La seguente procedura B&B-ISL va richiamata con l'istruzione B&B-ISL(1), dopo aver effettuato le inizializzazioni $D_1 := D$, $SL_1 := 0$, MINCOST $:= +\infty$, e MINSOL $:= \emptyset$.

```
procedure B&B-ISL(i: integer);
    {opportune dichiarazioni};
    begin
        for each matrice di switch S' massimale di D_i do begin
            S_i := S';
            D_{i+1} := D_i − S_i;
            SL_{i+1} := SL_i + L(S_i);
            LB := SL_{i+1} + B(D_{i+1});
            if LB < MINCOST then
                if D_{i+1} ≠ 0 then
                    B&B-ISL(i + 1)
                else begin
                    MINSOL := S_1, ... , S_i;
                    MINCOST := SL_{i+1};
                end;
        end;
    end;
```

Esempio 23.5 (B&B-ISL). La Fig. 23.6 mostra parte dell'albero delle scelte generato dall'algoritmo B&B-ISL per la matrice D della Fig. 23.4 (per mancanza di spazio, è indicato soltanto il percorso radice-foglia relativo allo schedule ottimo). Le matrici di switch dello schedule ottimo, illustrato nella Fig. 23.5, non sono indicate esplicitamente, ma si possono ricavare risalendo nell'albero delle scelte dalla foglia su fino alla radice. Ogni matrice di switch S_i è infatti data dalla differenza $D_i - D_{i+1}$. □

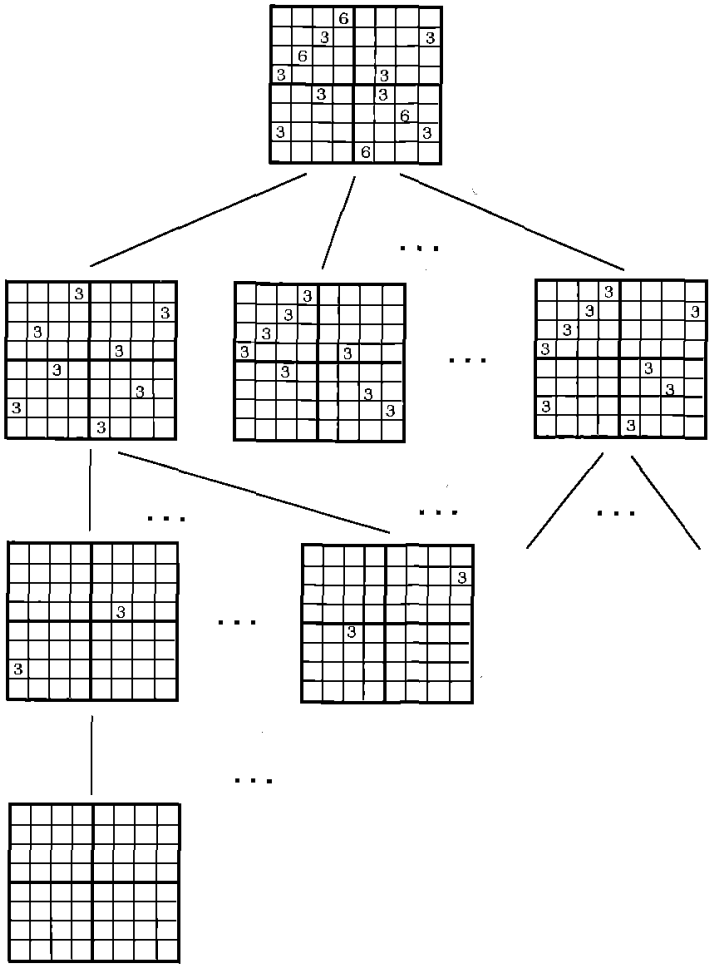


Figura 23.6: Algoritmo B&B-ISL. Ogni matrice di switch scelta è data dalla differenza tra la matrice del traffico rimanente del nodo padre e quella del figlio.

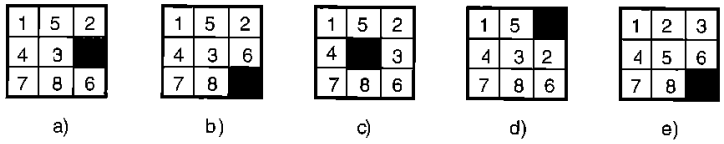


Figura 23.7: a) Una possibile configurazione iniziale del gioco del TAPPABUCO. b) - d) Le tre mosse possibili per la configurazione mostrata in a). e) La configurazione finale desiderata.

23.3 Esercizi

Esercizio 23.1 (Cricca di peso massimo). Si progetti, usando la tecnica del "branch-&-bound", un algoritmo per il problema della CRICCA DI PESO MASSIMO: "Dato un grafo non orientato $G = (N, A)$ con pesi interi non negativi sui nodi, trovare un sottoinsieme S dei nodi tale che esista un arco tra ogni coppia di nodi in S e la somma dei pesi dei nodi in S sia la più grande possibile."

Esercizio 23.2 (1-albero minimo). Si progetti un algoritmo di complessità polinomiale per trovare un 1-albero minimo di un grafo non orientato e pesato. Si riesegua poi l'algoritmo B&B-TSP, utilizzando un 1-albero minimo per calcolare il lower bound, sui dati dell'Esempio 23.2.

Esercizio 23.3 (Riduzione matrice delle distanze). Si progetti, usando la tecnica del "branch-&-bound", un algoritmo per il COMMESSO VIAGGIATORE utilizzando i seguenti criteri: (1) Scelta: inclusione/esclusione di un arco; (2) Regola di branch: considerare l'arco la cui esclusione massimizza il lower bound; (3) Visita: per priorità (cioè per lower bound più piccolo); (4) Lower bound: metodo di "riduzione" sulla matrice delle distanze (con LB inizializzato a 0):

a. calcolare $r_i = \min_j \{d_{ij}\}$ per ogni riga i della matrice;
b. se $r_i > 0$, sottrarre r_i a tutti gli elementi d_{ij} della riga i e sommare r_i ad LB;
c. calcolare $c_j = \min_i \{d_{ij}\}$ per ogni colonna j della matrice;
d. se $c_j > 0$, sottrarre c_j a tutti gli elementi d_{ij} della colonna j e sommare c_j ad LB;

Si esegua l'algoritmo sui dati dell'Esempio 23.2.

Esercizio 23.4 (Zaino 0/1). Si progetti un algoritmo "branch-&-bound" per il problema dello ZAINO 0/1 in forma di ottimizzazione.

Esercizio 23.5 (Matrici di switch massimali). Si progetti un algoritmo per generare tutte le matrici si switch massimali di una matrice ISL di traffico.

Esercizio 23.6 (Tappabuco). Si progetti un algoritmo "branch-&-bound" per il gioco del TAPPABUCO. È data una griglia quadrata di lato n, con $n^2 - 1$ tessere quadrate di lato unitario, numerate da 1 ad $n^2 - 1$, che coprono tutte le caselle della griglia, tranne una casella che rimane vuota. Una "mossa" consiste nello spostare orizzontalmente o verticalmente una tessera adiacente alla casella vuota, coprendo la casella vuota ma scoprendo la casella dove si trovava la tessera. Si vogliono porre le tessere in ordine crescente minimizzando il numero totale di mosse effettuate. (Si veda la Fig. 23.7, per $n = 3$). (Suggerimento: si definisca il lower bound basandosi sulla somma delle "distanze" delle tessere "fuori posto" e si visiti per priorità).

Capitolo 24

Algoritmi euristici

Quando si è presi dalla disperazione a causa della enorme difficoltà di un problema di ottimizzazione NP-arduo, si può ricorrere ad algoritmi "euristici" che forniscono una soluzione ammissibile, non necessariamente ottima né approssimata. Tali euristiche sono di solito basate su tecniche greedy (cfr. § 13) oppure di ricerca locale (cfr. § 15).

Per esempio, si consideri ancora il problema del COMMESSO VIAGGIATORE e sia π un circuito Hamiltoniano del grafo completo non orientato K_n. Una euristica ragionevole di ricerca locale può essere progettata sulla base del seguente intorno:

$$I_2(\pi) = \{\pi' : \pi' \text{ è ottenuto da } \pi \text{ cancellando due archi non consecutivi del circuito e sostituendoli con due archi esterni al circuito}\}.$$

Si noti che $|I_2(\pi)| = n(n-1)/2 - n$. Infatti, cancellati due archi non consecutivi, c'è un unico modo per riconnettere i due spezzoni in un circuito con archi che non fanno parte di π, mentre il numero di tutte le possibili coppie di archi non consecutivi di π è dato da quello di tutte le $n(n-1)/2$ coppie meno le n coppie di archi consecutivi. Pertanto, in un'euristica di ricerca locale, il passaggio da una soluzione π ad una $\pi' \in I_2(\pi)$ richiede $O(n^2)$ tempo.

Esempio 24.1 (Ricerca locale per COMMESSO VIAGGIATORE). Si consideri un'euristica di ricerca locale con intorno I_2 e la si esegua sui dati della Fig. 23.2. Come illustrato nella Fig. 24.1, partendo dal circuito iniziale $\pi = 1, 2, 3, 4, 5, 1$ con costo $c(\pi) = 25$, e scambiando gli archi [1, 5] e [3, 4] con gli archi [1, 4] e [3, 5], si ottiene la nuova soluzione $\pi' = 1, 2, 3, 5, 4, 1$ di costo $c(\pi') = 23$. Scambiando ulteriormente gli archi [1, 2] e [3, 5] con [1, 3] e [2, 5], si ottiene la soluzione ottima $\pi'' = 1, 3, 2, 5, 4, 1$ di costo $c(\pi'') = 19$. □

Ovviamente, una soluzione iniziale può essere generata casualmente oppure costruita tramite una semplice euristica greedy (per esempio, partendo da una città qualsiasi e aggiungendo ogni volta la città mancante più vicina a quella che è stata aggiunta per ultima). Purtroppo, il numero complessivo di soluzioni π esaminate dall'euristica di ricerca locale può essere esponenziale. Inoltre, non vi è alcuna garanzia che la soluzione trovata sia ottima. Infatti, come già è stato osservato nel § 15, un algoritmo di ricerca locale si arresta su un ottimo locale, cioè su una soluzione π il cui costo $c(\pi)$ è minore di $c(\pi')$ per ogni $\pi' \in I_2(\pi)$. In generale, niente assicura che tale soluzione sia anche un ottimo globale, poiché può esistere una soluzione

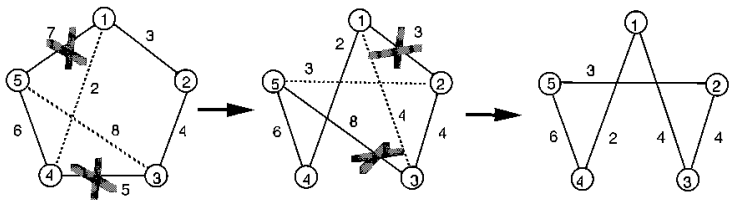


Figura 24.1: Euristica di ricerca locale con intorno I_2 per il COMMESSO VIAGGIATORE.

$\pi^* \notin I_2(\pi)$ con costo $c(\pi^*)$ minore di $c(\pi)$, per esempio una ottenibile scambiando 3 o 4 archi anziché due soli archi.

Generalizzando, si può definire, per $2 \leq k \leq n$, un intorno $I_k(\pi)$ dato da tutte le soluzioni π' ottenibili da π cancellando k archi del circuito e sostituendoli con k archi esterni al circuito. Purtroppo, è possibile dimostrare che l'unico intorno che garantisce sempre che l'ottimo locale trovato sia anche l'ottimo globale si ottiene solo dall'unione di $I_2(\pi)$, $I_3(\pi)$, ... , $I_n(\pi)$. Comunque, prove sperimentali hanno mostrato che per l'intorno dato dall'unione di $I_2(\pi)$ e $I_3(\pi)$ si ottiene una euristica di ricerca locale che fornisce in pratica "buone" soluzioni. Ovviamente, tale euristica ha complessità esponenziale nel caso pessimo e non vi è alcuna garanzia che la soluzione prodotta sia vicina a quella ottima! In tal caso, è pratica comune ripetere l'esecuzione dell'euristica di ricerca locale un numero prefissato di volte, p.e. 100, scegliendo ogni volta, in modo casuale, una soluzione ammissibile di partenza, e selezionando alla fine la migliore delle 100 soluzioni (ottimi locali) così individuate.

Consideriamo adesso il problema della BISEZIONE (cfr. § 18) in forma di ottimizzazione:

BISEZIONE: "Dato un grafo non orientato $G = (N, A)$ di n nodi, trovare una partizione di N in due sottoinsiemi N_0 ed N_1 di $n/2$ nodi ciascuno tale che il numero di archi con un estremo in N_0 e l'altro estremo in N_1 sia minimo."

Dato G con n pari, una semplice euristica greedy può essere basata sulle considerazioni seguenti. Si inizia selezionando casualmente un nodo che è posto in N_0 ed un altro nodo che è posto in N_1. Si prosegue aggiungendo alternatamente un nodo ad N_0 e il nodo successivo ad N_1, fino all'esaurimento dei nodi. Come mostrato nella Figura 24.2, il nodo i da aggiungere in ciascun passo ad N_k è scelto in modo da minimizzare il numero di archi $[i, j]$ con $j \in N_{1-k}$. Nel caso ci siano più nodi che soddisfano tale criterio, però, si sceglie tra questi il nodo i che massimizza il numero di archi $[i, j]$ con $j \in N_k$. Se ci sono più nodi che soddisfano anche il secondo criterio, allora il nodo i è scelto casualmente tra questi. Tale euristica, detta MIN-MAX-GREEDY, si comporta molto bene, proprio grazie al secondo criterio di scelta. Infatti, gli archi $[i, j]$ con $j \in N_k$ non andranno ad aumentare il costo della partizione, poiché i è aggiunto proprio ad N_k. Pertanto, più archi di questo tipo sono aggiunti subito, minore diviene la possibilità che nei passi successivi si sia costretti a scegliere un nodo che, introducendo parecchi archi tra N_k ed N_{1-k}, faccia aumentare di molto il costo della partizione.

L'euristica MIN-MAX-GREEDY può essere realizzata in modo da avere complessità $O(n^2)$ nel caso pessimo, utilizzando una apposita struttura di dati detta "a cestini"

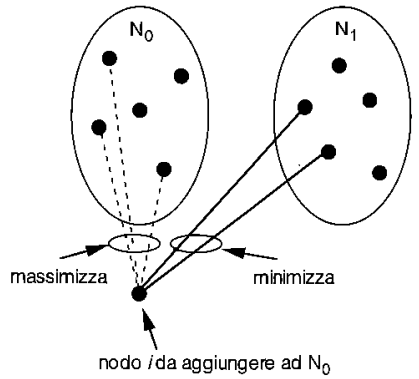


Figura 24.2: Euristica MIN-MAX-GREEDY per la BISEZIONE: situazione prima di aggiungere un nuovo nodo ad N_0.

(cfr. Esercizio 24.4). Prove sperimentali hanno mostrato che in pratica MIN-MAX-GREEDY fornisce "buone" soluzioni. Anche in questo caso, è opportuno ripetere l'esecuzione dell'euristica un numero prefissato di volte, p.e. 100, scegliendo ogni volta, in modo casuale, una nuova coppia di nodi di partenza da inserire in N_0 ed N_1, e selezionando alla fine la migliore delle 100 soluzioni individuate.

Anche per il problema della BISEZIONE si possono definire euristiche di ricerca locale, che magari iniziano proprio a partire dalla soluzione prodotta da MIN-MAX-GREEDY e poi continuano scambiando ripetutamente tra loro un nodo di N_0 con uno di N_1. Purtroppo, mentre per il problema del COMMESSO VIAGGIATORE le euristiche di ricerca locale risultano nettamente superiori a quelle greedy, per la BISEZIONE invece accade esattamente il contrario, e per ottenere soluzioni migliori occorre considerare tecniche euristiche più raffinate della ricerca locale.

La ricerca locale, infatti, tentando sempre di migliorare il costo della soluzione, presenta l'inconveniente di bloccarsi su un minimo locale. Per superare tale inconveniente, sono state introdotte tecniche più sofisticate di ricerca che permettono anche di peggiorare il costo della soluzione, nella speranza di sfuggire da un minimo locale. Tali tecniche devono però "evitare" la possibilità di effettuare troppo presto "mosse" inverse di "miglioramento", per non ricadere subito sul medesimo minimo locale. A tal fine, vengono introdotte delle "proibizioni" delle mosse che devono valere per un certo lasso di tempo, e per questo tali tecniche sono dette di "ricerca tabù". Nel seguito di questo capitolo, sarà descritta anche la tecnica di "ricerca tabù reattiva", dove il tempo di proibizione delle mosse è determinato automaticamente dall'algoritmo euristico stesso. In particolare, la tecnica sarà esemplificata proprio sul problema della BISEZIONE, descrivendone il migliore algoritmo euristico noto finora.

24.1 Ricerca tabù

La tecnica della ricerca locale passa attraverso una sequenza $S_0, S_1, \ldots, S_m$ di soluzioni, fino ad arrestarsi su un ottimo locale S_m. Nel passaggio da una soluzione S_t alla soluzione successiva S_{t+1}, è applicata una certa regola di trasformazione.

Un modo equivalente di descrivere tale procedimento corrisponde ad effettuare una "mossa", tra un certo insieme di mosse possibili, in modo da selezionare S_{t+1} nell'intorno di S_t. Denotato con $M = \{\mu_1, \ldots, \mu_p\}$ l'insieme delle mosse possibili, l'intorno $I(S_t)$ al passo t-esimo è definito come:

$$I(S_t) = \{S : S \text{ è ottenibile da } S_t \text{ applicando una mossa } \mu_i \in M\}.$$

In questo modo, l'intorno $I(S_t)$ dipende dall'ultima soluzione S_t e dall'intero insieme M delle mosse possibili. In generale, però, si può definire l'intorno in modo che dipenda dalla sequenza $S_0, S_1, \ldots, S_t$ di soluzioni e da un sottoinsieme M' di M. In altri termini, alcune delle soluzioni in $I(S_t)$ non possono essere selezionate perché le rispettive mosse μ_i sono proibite. Un semplice modo per proibire alcune mosse consiste nell'introduzione di un parametro T di proibizione, che determina per quanto tempo una mossa μ_i rimarrà proibita dopo la sua esecuzione. La tecnica di ricerca tabù fissa è ottenuta mantenendo costante il parametro T per tutta la durata della ricerca. In questo modo la scelta della soluzione S_{t+1} è effettuata nell'insieme $I_P(S_t) \subset I(S_t)$ delle soluzioni "permesse", che sono ottenibili cioè da S_t applicando una mossa che non è stata usata durante le ultime T iterazioni. In dettaglio, sia ULTIMOUSO$[\mu_i]$ l'iterazione t nella quale è stata usata per l'ultima volta la mossa μ_i (con ULTIMOUSO$[\mu_i]$ inizializzato a $-\infty$) e si indichino con μ_i^{-1} la mossa inversa di μ_i e con $c(S)$ il costo della soluzione S. Allora, si ha il seguente criterio, detto "della migliore mossa permessa":

$$\begin{aligned} I_P(S_t) &= \{S\text{: } S \text{ è ottenibile da } S_t \text{ applicando una mossa } \mu_i \in M \text{ tale che, } \\ &\quad \text{ULTIMOUSO}[\mu_i^{-1}] < t - T\} \\ S_{t+1} &= S \in I_P(S_t) \text{ tale che } c(S) \leq c(S') \text{ per ogni } S' \in I_P(S_t). \end{aligned}$$

Si noti che, poiché ci sono mosse proibite, non è detto che $c(S_{t+1}) \leq c(S_t)$, cioè, il costo della soluzione può peggiorare ed è così possibile sfuggire da un minimo locale, purché T sia scelto oculatamente. Chiaramente, se $T = 0$, allora non ci sono mai proibizioni e si riottiene la ricerca locale. In linea di principio, maggiore è T, maggiore è la diversificazione, cioè maggiore il numero di iterazioni prima che si rivisiti per la seconda volta la stessa soluzione. Però, T non può essere troppo grande, altrimenti tutte le mosse diventano presto proibite e la ricerca si blocca. In generale, il valore di T deve garantire l'esistenza di almeno due mosse permesse per ciascuna iterazione (altrimenti la ricerca non è neanche influenzata dal costo $c(S)$ delle soluzioni).

Supponiamo che una soluzione S sia individuata da una stringa di n bit e che la mossa μ_i consista nel complementare il bit i-esimo, $1 \leq i \leq n$. Si noti che ogni mossa è idempotente, cioè l'inversa di ogni mossa è uguale a se stessa: $\mu_i^{-1} = \mu_i$. Date due stringhe X ed Y, sia $H(X, Y)$ la loro distanza di Hamming, ovvero il numero di bit corrispondenti che sono diversi nelle due stringhe. In tal caso, $T \leq n - 2$ garantisce che ci siano sempre almeno due mosse permesse. Se sono eseguite solo mosse permesse, allora è facile dimostrare che il minimo intervallo R di ripetizione è $2(T + 1)$. In altri termini, se $S_{t+R} = S_t$, allora $R \geq 2(T+1)$. La dimostrazione è immediata poiché, quando un bit è complementato, resta "congelato" per le prossime T iterazioni.

Esempio 24.2 (Proibizioni e diversificazione). Sia la funzione da minimizzare $c(S)$ uguale al numero intero la cui rappresentazione binaria è data proprio da S (in tal caso, la soluzione ottima S^* è data dalla stringa nulla). Come illustrato nella Fig.

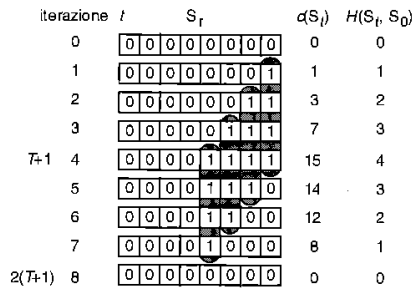


Figura 24.3: Correlazione tra proibizione T e diversificazione $H(S_t, S_0)$. Nell'esempio, $T = 3$.

24.3 per $n = 8$, si assuma di partire proprio da S^* e sia $T = 3$. All'iterazione 0 la mossa che dà il minimo valore della funzione c è quella che complementa il bit meno significativo. All'iterazione 1, il bit meno significativo è "proibito" (il periodo in cui un bit è "congelato", perché la rispettiva mossa è proibita, è mostrato in nero) e quindi la migliore mossa permessa consiste nel cambiare il penultimo bit meno significativo, e così via. Si noti che la distanza di Hamming è crescente nelle prime $T + 1$ iterazioni e che la massima distanza di Hamming è raggiunta all'iterazione $T + 1$. Successivamente, la distanza decresce ed all'iterazione $2(T + 1)$ si ripete la soluzione iniziale. Si noti comunque che le soluzioni visitate nelle prime $T + 1$ iterazioni (quando la distanza cresce) sono diverse da quelle visitate nelle successive $T + 1$ iterazioni (quando la distanza decresce). In questo modo, non si perde tempo a rivisitare soluzioni già visitate (a parte la soluzione iniziale). □

Vediamo innanzitutto come progettare una semplice euristica di "ricerca tabù fissa" per la BISEZIONE, dove il parametro T di proibizione è fornito come dato di ingresso alla procedura e rimane inalterato durante la computazione. Successivamente, vedremo altre due versioni sempre più raffinate dell'euristica: una "ricerca tabù randomizzata", dove T è scelto in modo casuale, e una "ricerca tabù reattiva", dove il valore più opportuno di T è messo a punto dalla euristica stessa, sempre tramite randomizzazione.

24.1.1 Proibizione fissa

Consideriamo una euristica di ricerca tabù fissa per la BISEZIONE dove una soluzione $S = (N_0, N_1)$ è individuata da una stringa di n bit e la mossa μ_i consiste nel complementare il bit i-esimo, $1 \leq i \leq n$, come nell'Esempio 24.2. In questo modo, N_0 è rappresentato dagli indici dei bit a 0, N_1 dagli indici dei bit ad 1, e la mossa μ_i può essere individuata specificando semplicemente l'indice i. Si noti che, per definizione, la soluzione S non è necessariamente ammissibile. Infatti, poiché una mossa consiste di fatto nello spostare un nodo da N_0 ad N_1 o viceversa, non è detto in generale che N_0 ed N_1 abbiano la stessa cardinalità. Solo se ciò avviene, cioè se $|N_0| = |N_1| = n/2$, allora la soluzione S è ammissibile.

Nel seguito, verrà utilizzato un parametro di proibizione frazionale T_f tale che $T = \lfloor nT_f \rfloor$, in modo da "normalizzare" la proibizione rispetto ad n. Inoltre, viene usata come contatore di iterazione una variabile globale t accessibile e modificabile

da tutte le procedure. Analogamente, c_{min} è una variabile globale che indica il costo $c(S_{min})$ della migliore soluzione ammissibile S_{min} incontrata durante la ricerca. Si assume che tale variabile sia inizializzata dalla procedura MIN-MAX-GREEDY. Anche ULTIMOUSO è un vettore globale, tale che ULTIMOUSO[i] registra l'ultimo valore della variabile di iterazione t per il quale il nodo i è stato spostato da un insieme all'altro. Infine, sono usate due variabili n_0 ed n_1 tali che $n_0 = |N_0|$ e $n_1 = |N_1|$.

La seguente procedura BISEZIONE-RTF (per Ricerca Tabù Fissa) riceve come parametri la proibizione frazionale T_f ed il numero di iterazioni da eseguire. Si assume che una soluzione ammissibile di partenza sia stata individuata con MIN-MAX-GREEDY. La procedura BISEZIONE-RTF sposta un nodo alla volta. L'indice k individua il sottoinsieme N_k nel quale aggiungere il nodo i tolto da N_{1-k}. Il nodo i è determinato dalla funzione MOSSAMIGLIORE in accordo al criterio della "migliore mossa permessa", cioè in modo che ULTIMOUSO[i] $< t - T$ e il costo c sia il più piccolo possibile spostando i da N_{1-k} a N_k (ciò può essere fatto efficientemente utilizzando ancora una struttura di dati "a cestini": si veda l'Esercizio 24.5). Se la soluzione $S = (N_0, N_1)$ è ammissibile ed il costo $c(S)$ è minore del costo c_{min} della migliore soluzione finora trovata, allora viene registrato il costo della nuova soluzione.

```
procedure BISEZIONE-RTF(T_f: real; iterazioni: integer);
  begin
    T := ⌊nT_f⌋;
    for j := 1 to iterazioni do begin
      if n_0 > n div 2 then k := 1 else k := 0;
      i := MOSSAMIGLIORE(1 − k);
      N_k := N_k ∪ {i};
      N_{1−k} := N_{1−k} − {i};
      ULTIMOUSO[i] := t;
      t := t + 1;
      if n_0 = n_1 and c(S) < c_min then c_min := c(S);
    end;
  end;
```

A causa della struttura stessa del problema della BISEZIONE, è bene non scegliere valori di T_f maggiori di 1/4. Infatti una stringa binaria ed il suo complemento (con gli zeri e gli uno tra loro scambiati) indicano di fatto la stessa soluzione, con i due insiemi della partizione scambiati. Pertanto, un valore di $T_f > 1/2$ comporterebbe che, dopo essere partiti con una soluzione, si raggiungerebbe dopo $T + 1$ iterazioni una partizione più vicina al complemento che a quella di partenza. La scelta di $T_f < 1/4$ elimina queste "oscillazioni" tra una soluzione ed il suo complemento. Una scelta ragionevole del parametro *iterazioni* è invece $100n$. Prove sperimentali hanno mostrato come la scelta di T_f abbia un effetto cruciale sulla soluzione trovata: una scelta "azzeccata" porta a buone bisezioni, mentre una scelta "sbagliata" porta a bisezioni scadenti.

24.1.2 Proibizione randomizzata

Per evitare di dover provare e riprovare "manualmente" svariati valori di T_f eseguendo ripetutamente la procedura BISEZIONE-RTF fino a selezionare il valore di T_f buono, si possono semplicemente effettuare delle scelte casuali! Si ottiene così la seguente procedura probabilistica BISEZIONE-RTR (per Ricerca Tabù Randomizzata):


```
procedure BISEZIONE-RTR(iterazioni, individuale: integer);
  begin
    t := 0;
    while t < iterazioni do begin
      MIN-MAX-GREEDY
      t_fine := t + individuale;
      while t < t_fine do begin
        temp := RANDOM(1, 25);
        T_f := temp/100;
        BISEZIONE-RTF(T_f, n)
      end
    end
  end;
```

Il parametro *iterazioni* fornisce il numero complessivo di iterazioni, mentre *individuale* indica il numero di iterazioni di ogni singola "fase", che riparte dopo aver richiamato MIN-MAX-GREEDY (se *individuale* = *iterazioni* allora avviene una sola fase). Ogni n iterazioni viene scelto casualmente un nuovo T_f. In questo modo, si evita di scegliere un "cattivo" valore di T_f e mantenerlo per molte iterazioni. Al contrario, si continua a scegliere sempre valori diversi di T_f, che sono mantenuti per poche iterazioni. Prove sperimentali hanno mostrato che, scegliendo *individuale* = $10n$ ed *iterazioni* = $100n$, le soluzioni trovate da questa "ricerca tabù randomizzata" sono paragonabili a quelle trovate dalla "ricerca tabù fissa" nella quale si usi il miglior valore di T_f! In altri termini, molti valori casuali usati per poco tempo danno gli stessi risultati del miglior valore possibile usato a lungo.

24.1.3 Proibizione reattiva

Per trovare soluzioni ancora migliori del problema della BISEZIONE, si può impiegare una "ricerca tabù reattiva", dove la messa a punto del parametro T_f è effettuata automaticamente dalla procedura stessa, tenendo conto delle proprietà del grafo in ingresso e dello spazio delle soluzioni.

L'euristica è progettata in modo che un valore di "successo" di T_f sia scelto con alta probabilità e mantenuto più a lungo. Il "successo" di un dato T_f è determinato in due modi: durante un "torneo" preliminare, misurando la "velocità di miglioramento" con la data proibizione per fasi molto brevi, e durante le successive fasi di ricerca vere e proprie, osservando se il valore corrente della proibizione ha portato all'individuazione di una soluzione migliore nella porzione di ricerca già effettuata.

Senza entrare nei dettagli, una funzione preliminare TORNEO valuta le possibili proibizioni $T_f \in \{0.01, 0.02, 0.03, \ldots, 0.25\}$ attraverso tre prove per ogni proibizione. In ciascuna delle tre prove, si parte con una soluzione fornita da MIN-MAX-GREEDY, raffinata con una ricerca locale senza proibizioni. Vengono registrati il valore iniziale c_{iniz} della funzione costo ed il relativo valore t_{iniz} dell'iterazione. Successivamente, è eseguita una serie di fasi "corte" di ricerche tabù con proibizione "fissa" T_f, di $2(T + 1)$ iterazioni ciascuna, seguite da ricerche locali senza proibizioni, finché non sono state accumulate $n/2$ iterazioni in tutto. Come sempre, una variabile globale c_{min} registra il costo della migliore soluzione ammissibile trovata, mentre t_{min} registra l'iterazione nella quale c_{min} è stata trovata. La migliore soluzione trovata durante ogni prova è memorizzata in un insieme. Dopo ogni prova è assegnato un "voto" a T_f, che è proporzionale alla "derivata" $(c_{iniz} - c_{min})/(t - t_{iniz})$ che,

poiché si sta minimizzando, è non negativa. Al termine del "torneo" viene restituita la proibizione più piccola che ha ottenuto il voto più alto.

Una procedura BISEZIONE-RTRR (per Ricerca Tabù Randomizzata Reattiva), con in ingresso i soliti parametri *iterazioni* e *individuale*, esegue

$$\lceil iterazioni / individuale \rceil$$

"fasi". Ogni "fase" dura per un numero di iterazioni uguale ad *individuale* e parte con la migliore soluzione estratta dall'insieme delle soluzioni trovate da TORNEO oppure, quando le soluzioni in tale insieme sono esaurite, con una soluzione trovata da MIN-MAX-GREEDY. La proibizione T_f è inizializzata con la migliore proibizione trovata da TORNEO. Ciascuna fase è formata da una sequenza di "sottofasi" più brevi. Ogni sottofase consiste in una "ricerca tabù fissa" di $2(T+1)$ iterazioni seguita da una ricerca locale senza proibizioni, e dura complessivamente n iterazioni. Se una soluzione ammissibile migliore è stata trovata in una sottofase, allora è mantenuto lo stesso T_f di "successo" anche per la sottofase successiva, altrimenti è scelto casualmente un nuovo T_f con probabilità proporzionale al voto ottenuto nel "torneo". Scelte ragionevoli di *individuale* ed *iterazioni* sono, rispettivamente, $10n$ e $100n$.

La procedura BISEZIONE-RTRR, dovuta a Battiti e Bertossi (1999), rappresenta il migliore algoritmo euristico noto finora per la BISEZIONE, sia per quanto riguarda la bontà delle soluzioni trovate, sia per i tempi di computazione piuttosto contenuti. Tutti i dettagli implementativi, nonché numerosissime prove sperimentali, si possono trovare nell'articolo originale.

24.2 Esercizi

Esercizio 24.1 (Ottimo locale per COMMESSO VIAGGIATORE). Si costruisca un grafo K_6 per il COMMESSO VIAGGIATORE per il quale esistano due soluzioni π e π^* tali che π sia un ottimo locale per l'intorno $I_2(\pi)$, mentre π^* sia un ottimo globale che non appartiene ad $I_2(\pi)$.

Si considerino i dati mostrati nella Fig. 24.4a), dove si assume che le distanze non esplicitate siano tutte uguali a 5. La soluzione π^* illustrata nella Fig. 24.4b ha costo $c(\pi^*) = 11$ ed è un ottimo globale. La soluzione π della Fig. 24.4c) ha costo $c(\pi) = 12$ ed è un ottimo locale, come si può verificare considerando i costi di tutte le soluzioni $\pi' \in I_2(\pi)$ mostrate nella Fig. 24.4d). L'ottimo globale π^* non appartiene a $I_2(\pi)$, poiché non è ottenibile da π scambiando due archi, ma scambiandone tre.

Esercizio 24.2 (Cardinalità di I_k). Si dimostri che la cardinalità di $I_k(\pi)$, per il COMMESSO VIAGGIATORE è $O(n^k)$. Per $k > 2$, esiste un unico modo di ricomporre il circuito dopo aver tolto k archi?

Esercizio 24.3 (Partizione). Si progettino un algoritmo euristico di tipo greedy ed uno di ricerca locale per il problema della PARTIZIONE in forma di ottimizzazione, dove si vuole minimizzare lo scarto tra la somma degli elementi del primo insieme e la somma di quelli del secondo insieme.

Esercizio 24.4 (MIN-MAX-GREEDY). Le operazioni richieste dall'euristica MIN-MAX-GREEDY possono essere realizzate usando una struttura di dati detta "a cestini". Vengono utilizzate due matrici $cestino[0..1, 0..n, 0..n]$ e $dimensione[0..1, 0..n]$. In generale, $cestino[k, h, d]$ contiene un nodo i tale che il numero di archi $[i,j]$ con $j \in N_k$ è uguale ad h, mentre $dimensione[k, h]$ contiene il numero di tali nodi. Inoltre,

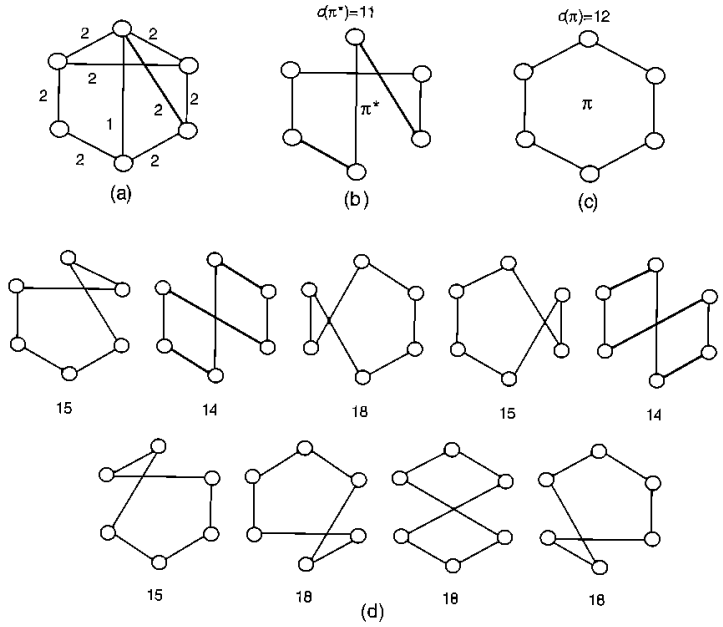


Figura 24.4: (a) Dati per il COMMESSO VIAGGIATORE (le distanze non indicate valgono 5). (b) Ottimo globale π^*. (c) Un ottimo locale π. (d) Tutte le soluzioni π' che stanno in $I_2(\pi)$.

sono mantenuti in $min[k]$ e $max[k]$ i valori del minimo e massimo "cestino" non vuoto (si veda la Fig. 24.5, notando come ogni nodo compaia in entrambi gli insiemi). Si considerino le seguenti operazioni:

INSERISCI(i, *cestino*), che inserisce il nodo i in *cestino*;

CANCELLA(i, *cestino*), che cancella il nodo i da *cestino*;

INCREMENTA(k, i, *cestino*), che aumenta di 1 il numero di archi $[i, j]$ con $j \in N_k$.

Si scrivano procedure Pascal efficienti per realizzare tali operazioni, aggiornando opportunamente i valori $min[k]$ e $max[k]$. Si mostri poi come realizzare efficientemente la scelta del nodo i da aggiungere alla partizione.

Esercizio 24.5 (Migliore mossa permessa per BISEZIONE-RTF). Si realizzi il criterio della "migliore mossa permessa" nell'euristica BISEZIONE-RTF per mezzo di procedure Pascal che utilizzano la struttura di dati "a cestini" (Suggerimento: al contrario dell'Esercizio 24.4, ogni nodo deve comparire in uno solo dei due insiemi e sarà caratterizzato da un valore che rappresenta lo "scarto" del costo c qualora il nodo sia spostato nell'altro insieme).

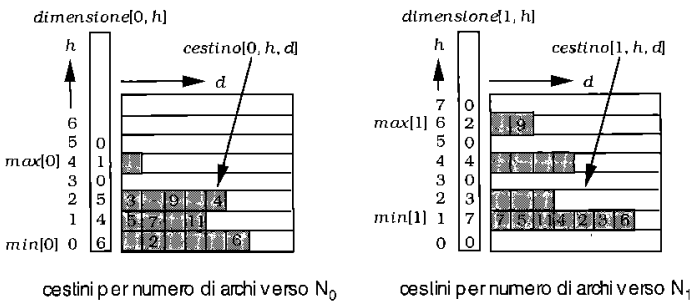


Figura 24.5: Struttura di dati "a cestini".

Capitolo 25

Indecidibilità ed universalità

Si consideri la seguente funzione Pascal per il calcolo del fattoriale di un numero naturale n:

```
function FATTORIALE(n: integer): integer;
  begin
    if n = 0 then
      FATTORIALE := 1
    else
      FATTORIALE := n * FATTORIALE(n - 1);
  end;
```

Cosa accade se la funzione è richiamata con argomento negativo? Per esempio, sia $n = -5$ e si computi FATTORIALE(-5). Poiché $-5 \neq 0$, si esegue il ramo **else**, che richiede di effettuare il prodotto $-5 *$ FATTORIALE(-6) prima di poter fornire il risultato di FATTORIALE(-5). Pertanto, per valutare FATTORIALE(-5) occorre prima valutare FATTORIALE(-6). Analogamente, per valutare FATTORIALE(-6) va prima valutato FATTORIALE(-7), e così via per argomento sempre decrescente. In altri termini, l'esecuzione entrerà in un ciclo infinito, senza possibilità alcuna di uscirvi! Questo comportamento è perfettamente in accordo con la definizione matematica del fattoriale, poiché la funzione $n!$ non è definita per argomenti negativi. La possibilità di ciclare all'infinito può essere tranquillamente eliminata dalla funzione FATTORIALE introducendo un confronto iniziale sul valore del parametro n, in modo da interrompere l'esecuzione e restituire un messaggio di errore qualora n sia negativo.

Si consideri adesso la seguente funzione Pascal, solo apparentemente simile alla precedente:

```
function COLLATZ(n: integer): boolean;
  begin
    if n = 3 then
      COLLATZ := true
    else if odd(n) then
      COLLATZ := COLLATZ((3*n + 1) div 2)
    else
      COLLATZ := COLLATZ(n div 2)
  end;
```

La funzione richiama se stessa con dato uguale a $(3n + 1)/2$, se n è dispari, o con dato uguale ad $n/2$, se n è pari, fornendo il valore **true** non appena è raggiunto 3. Ci sono dei valori positivi di n che provocano un ciclo infinito? Per tutti i valori positivi di n sui quali la funzione è stata provata finora, si è sempre ottenuto il risultato **true**, ma nessuno ha finora dimostrato formalmente che COLLATZ(n) = **true** per tutti gli interi positivi.

25.1 Il problema della terminazione

Le considerazioni precedenti mostrano come non sia affatto banale stabilire se una procedura, anche apparentemente semplice come la COLLATZ, fornisca sempre un risultato per ogni valore dei dati di ingresso oppure possa non terminare per particolari valori di tali dati.

Sebbene la non terminazione non presenti, in un certo senso, alcun problema nel caso che sia dovuta alla valutazione di una funzione non definita, essa è chiaro sintomo di errore qualora chi ha progettato l'algoritmo abbia commesso degli sbagli. Pertanto, sarebbe estremamente utile poter risolvere il seguente problema:

TERMINAZIONE: "Data la descrizione di un qualsiasi algoritmo A e di un qualsiasi dato legale x per esso, A termina se eseguito con dato x?"

A maggior ragione, sarebbe ancora più utile poter risolvere anche il problema seguente:

CORRETTEZZA: "Data la descrizione di un qualunque problema computazionale P e di un qualsiasi ipotetico algoritmo A che lo risolva, A produce la corretta risposta per ogni dato di ingresso legale per P?"

Chiaramente, la risolubilità del problema della CORRETTEZZA è intimamente legato a quello della TERMINAZIONE. Infatti, per essere corretto, un algoritmo deve innanzitutto terminare e fornire un risultato, sia esso corretto oppure no. Pertanto, il problema della CORRETTEZZA non può essere più facile di quello della TERMINAZIONE.

Come risolvere il problema della TERMINAZIONE? Esiste un modo più astuto di quello di eseguire l'algoritmo A sul dato x e attenderne la fine della computazione, ammesso che tale computazione termini? La risposta è no, poiché il problema della TERMINAZIONE non è risolvibile in tempo finito!

Per dimostrare questo risultato, si consideri il famoso paradosso di Russell, applicato agli aggettivi della lingua italiana. Un aggettivo è detto autologico, se è vero quando è applicato a se stesso, mentre è detto eterologico, se è falso quando è applicato a se stesso.

Esempio 25.1 (Aggettivi autologici ed eterologici). Sono aggettivi autologici: polisillabo, italiano, corto. Sono aggettivi eterologici: monosillabo, inglese, astratto. Infatti polisillabo, essendo formato da 5 sillabe, è davvero polisillabo e quindi autologico, mentre monosillabo, essendo anch'esso formato da 5 sillabe, non è monosillabo e quindi è eterologico. Analogamente, italiano e inglese sono entrambi aggettivi italiani, e quindi il primo è autologico mentre il secondo, non essendo inglese, è eterologico. Infine, corto è davvero un aggettivo corto, e quindi è autologico, mentre astratto, non essendo affatto un aggettivo astratto ma ben concreto, è eterologico. □

La questione che si pone è allora quella di stabilire se eterologico sia autologico o eterologico. Se eterologico è autologico, allora deve essere vero quando è applicato a se stesso, ma eterologico applicato a se stesso dà eterologico. Viceversa, se eterologico è eterologico, allora è applicabile a se stesso, e quindi è autologico per definizione. In entrambi i casi si giunge ad una contraddizione e il miglior modo di risolvere la questione, proposto dallo stesso Russell, consiste semplicemente nel non fornire alcuna risposta!

Si estenda il paradosso di Russell al caso di funzioni booleane Pascal. Si dice che una funzione è autologica se, applicata a se stessa, termina con risultato **true**, mentre è eterologica se, applicata a se stessa, o non termina o termina con risultato **false**.

Come è possibile applicare funzioni Pascal a se stesse? Apparentemente, ciò sembrerebbe impossibile. Infatti, per definire una funzione Pascal, occorre definirne i parametri, con i loro tipi, e il tipo del risultato. Se una funzione F deve essere applicabile a se stessa, occorre che almeno uno dei parametri sia una funzione G dello stesso tipo:

function F(n: integer; **function** G(m: integer): boolean): boolean;

ma anche i parametri di G devono includere una funzione dello stesso tipo, e si arriva all'assurdo che un parametro "attuale", cioè specificato in una "chiamata", debba avere una definizione più lunga di quella della funzione a cui è passato! Questa difficoltà può essere aggirata passando come parametro un dato che rappresenti la funzione. Per esempio, in Pascal basta passare come parametro un array di caratteri, i cui elementi contengono, carattere per carattere, la stringa di simboli che descrive la funzione.

Esempio 25.2 (Rappresentazione). La seguente funzione

```
function VERO(i: integer): boolean;
  begin
    VERO := true
  end;
```

può essere rappresentata con un array R di 51 caratteri tale che

R[1..51] = functionvero(i:integer):boolean;beginvero:=trueend;

analogamente, anche il dato d'ingresso i può essere rappresentato con un array N di caratteri. □

Sia R_f la rappresentazione di una funzione f e si supponga, senza perdere in generalità, che la funzione f abbia come tipo del parametro una stringa, rappresentata con un array di caratteri. Si abbia a disposizione la seguente funzione:

function INTERPRETA(R, N: vettoredicaratteri): boolean;

tale che:

INTERPRETA$(R, N) = b$,	se R è la rappresentazione di una, funzione f e $f(N) = b$, $b \in$ {**true**, **false**}
INTERPRETA(R, N) = **false**,	se R non è rappresentazione di alcuna f,

e che non termina se R è la rappresentazione di una funzione f ed $f(N)$ non termina.

La funzione INTERPRETA, detta funzione universale, riceve in ingresso la descrizione R_f di una funzione Pascal e quella di un dato, interpreta ed esegue le istruzioni della funzione f su quel dato, fornendo il risultato $f(N)$, se $f(N)$ termina. Si noti che, se R_f rappresenta f, allora

$$\text{INTERPRETA}(R_f, N) = f(N)$$

e, in particolare, qualora $N = R_f$,

$$\text{INTERPRETA}(R_f, R_f) = f(R_f).$$

Ritornando al paradosso di Russell e al problema della TERMINAZIONE, si assuma per assurdo di avere a disposizione la funzione TERMINA(R_f, N), che risolva tale problema fornendo sempre una risposta in tempo finito:

$$\text{TERMINA}(R_f, N) = \textbf{true}, \qquad \text{se } f(N) \text{ termina},$$
$$\text{TERMINA}(R_f, N) = \textbf{false}, \qquad \text{se } f(N) \text{ non termina}.$$

Si definisca la seguente funzione:

```
function ETERO(X: vettoredicaratteri): boolean;
  begin
    if TERMINA(X, X)
      then ETERO := not INTERPRETA(X, X)
      else ETERO := true
  end;
```

La funzione ETERO è eterologica? Si provi ad eseguire la computazione di ETERO con $X = R_{\text{ETERO}}$. Poiché TERMINA termina sempre, anche ETERO deve terminare sempre. E poiché TERMINA(R_{ETERO}, R_{ETERO}) = **true**, ETERO(R_{ETERO}) è uguale a **not** INTERPRETA(R_{ETERO}, R_{ETERO}). Essendo INTERPRETA(R_{ETERO}, R_{ETERO}) = ETERO(R_{ETERO}), si ottiene che ETERO(R_{ETERO}) = **not** ETERO(R_{ETERO}), che è una contraddizione che non può essere risolta, come nel caso degli aggettivi, col non fornire alcuna risposta. Infatti per ipotesi ETERO deve terminare, dato che è stato assunto che TERMINA termina sempre, e quindi ETERO deve fornire una risposta. Pertanto, la contraddizione deriva dall'aver assunto l'esistenza della funzione TERMINA che termina sempre!

Il precedente risultato può essere enunciato nel modo seguente:

Teorema 25.1. *Un qualsiasi linguaggio di programmazione che sia abbastanza potente per poter programmare il suo stesso interprete non può essere usato per programmare la propria funzione di terminazione.* □

In altri termini, se è programmabile la funzione TERMINA allora non è programmabile la funzione INTERPRETA, mentre se è programmabile la INTERPRETA allora non è programmabile la TERMINA.

Esempio 25.3 (INTERPRETA programmabile). La funzione universale INTERPRETA è disponibile in tutti i sistemi di elaborazione per eseguire i programmi Pascal scritti dagli utenti, purché ci sia memoria sufficiente per contenere R ed N, ed è essa stessa scritta in Pascal! Pertanto TERMINA non può essere programmata in Pascal. In altri termini, è impossibile per un professore scrivere un programma Pascal, che accetti come dato un qualsiasi programma scritto da uno studente e un qualsiasi dato per esso, e dica se quel programma produrrà un risultato per quel dato o entrerà in un ciclo infinito ... con gran sollievo per lo studente.[1] □

[1] Tale risultato negativo si riferisce ad un metodo generale che valga per ogni programma e non esclude quindi la possibilità che il professore si accorga se un particolare programma (d'esame) scritto dallo studente entrerà in un ciclo infinito ... con gran rammarico per lo studente.

Esempio 25.4 (TERMINA programmabile). Si consideri un sottolinguaggio L del Pascal, in cui non sia ammessa la ricorsione, né le istruzioni **goto**, **while** e **repeat**. In tal caso, l'unica istruzione iterativa disponibile resta il **for**. Poiché tale costrutto è di iterazione limitata, ogni programma scritto con istruzioni in L termina sempre e quindi TERMINA è uguale alla funzione che assume valore costante **true** per ogni argomento. Nel linguaggio L, però, non è possibile scrivere la funzione INTERPRETA. Infatti tale funzione dovrebbe impostare preventivamente i limiti inferiori e superiori delle variabili che controllano l'iterazione delle istruzioni **for** usate, per ogni programma interpretato. Ma ciò è impossibile, poiché occorrerebbe conoscere in anticipo il numero di passi richiesti dall'esecuzione del programma stesso. Pertanto, scrivendo programmi nel linguaggio L si preclude la possibilità di programmare molte funzioni di utilità pratica. Per esempio, una semplice ed utile funzione come $\mu(f) = \min\{n : f(n) = \textbf{true}\}$ non può essere programmata, poiché non è possibile scegliere un valore opportuno con cui impostare gli estremi dell'indice di un **for** prima di avviare la computazione. Infatti, per programmare la funzione μ occorre un **while**:

```
function MU(function F(i: integer): boolean): integer;
  var i: integer;
  begin
    i := 1;
    while not F(i) do
      i := i + 1;
    MU := i
  end;
```
□

Sorprendentemente, l'unico modo per poter programmare la TERMINA consiste nell'utilizzare un linguaggio di programmazione così poco potente, come il linguaggio L dell'Esempio 25.4, nel quale non è possibile programmare non solo la INTERPRETA, ma neanche molte altre funzioni utili, come la funzione μ. L'impossibilità generale di programmare la TERMINA ha come conseguenza negativa l'impossibilità di progettare molti altri programmi potenzialmente utilissimi. Per esempio, non è possibile scrivere un programma che, ricevuta in ingresso una proposizione matematica, decida se questa è un teorema oppure no. In effetti, un teorema si può rappresentare con una funzione booleana Pascal che termina sempre e restituisce sempre **true** per tutti i valori dei suoi parametri.

Esempio 25.5 (Proprietà commutativa). Il teorema della proprietà commutativa $x + y = y + x$ rispetto alla somma può essere espresso con la funzione Pascal:

```
function COMMUTATIVA(x, y: integer): boolean;
  begin
    COMMUTATIVA := (x + y = y + x);
  end;
```
□

Si assuma di poter programmare una funzione Pascal TEOREMA che termina sempre e restituisce **true** se e solo se il suo argomento è una funzione booleana che restituisce sempre **true**. Ma allora si può programmare la funzione TERMINA:

```
function TERMINA(function F(i: integer): boolean; x: integer): boolean;
  begin
    TERMINA := TEOREMA(G(F, x));
  end;
```

dove la funzione G è definita come segue:

```
function G(function F(i: integer): boolean; x: integer): boolean;
  begin
    if F(x) then G := true else G := true;
  end;
```

È facile vedere che l'argomento di TEOREMA termina esattamente quando termina $F(x)$ e restituisce **true** quando termina. Pertanto, TEOREMA restituisce **true** se e solo se $F(x)$ termina, e restituisce **false** se e solo se $F(x)$ non termina, che è una contraddizione perché TERMINA non è programmabile.

25.2 Gerarchia di indecidibilità

Poiché il problema della TERMINAZIONE è di tipo decisionale, ma è solo parzialmente risolvibile, si dice che esso è semidecidibile. Infatti, eseguendo l'algoritmo A con dato x ed attendendo una risposta si può decidere se $A(x)$ termina, ma non si può mai asserire, qualunque sia il lasso di tempo trascorso, che A non terminerà. Infatti, anche se si interrompesse la computazione dopo che sono state eseguite t istruzioni, niente potrebbe vietare che la computazione terminasse proprio dopo aver eseguito un'istruzione in più, e ciò vale comunque grande si scelga t. A differenza dei problemi trattati nei capitoli precedenti, la TERMINAZIONE non ammette alcun certificato, non solo polinomiale o superpolinomiale, ma nemmeno finito, poiché non è possibile verificare in tempo finito che un algoritmo A eseguito su un dato x fornisca una risposta oppure no.

È interessante paragonare il problema della TERMINAZIONE ad altri celebri problemi impossibili della Matematica e della Fisica:

1. È impossibile trovare due interi n ed m, maggiori di uno, tali che $m^2 = 2n^2$;
2. È impossibile trovare una frazione n/m che esprima il rapporto tra il raggio e la circonferenza di un cerchio;
3. È impossibile trisecare un angolo usando solo riga e compasso;
4. È impossibile osservare accuratamente sia la posizione che la velocità di un elettrone;
5. È impossibile costruire una macchina in moto perpetuo.

L'indecidibilità del problema della TERMINAZIONE ha implicazioni sia pratiche che teoriche:

a. non è possibile costruire un dispositivo che "protegga" un centro di calcolo evitando di eseguire quei programmi errati che entreranno in ciclo allorché saranno eseguiti;
b. è possibile dimostrare l'indecidibilità di molti altri problemi provando che la TERMINAZIONE si riduce ad essi, utilizzando tecniche di riduzione simili a quelle viste nei § 18 e 19 (imponendo semplicemente che la funzione usata nella trasformazione richieda tempo finito).

Tutti i problemi trattati nei capitoli precedenti sono decidibili, incluso la FORMULA WS1S che, pur non essendo in $\mathbb{E}$, è pur sempre risolvibile con un algoritmo che richiede tempo finito. Purtroppo, moltissimi altri problemi non sono affatto decidibili. Ovviamente, tali problemi devono avere uno spazio infinito di soluzioni,

poiché altrimenti potrebbero essere risolti in tempo finito "provando tutte le possibilità". È possibile dimostrare che ogni problema (linguaggio) semidecidibile L con spazio infinito di soluzioni può essere espresso nella forma:

$$L = \{x : \exists\, y\; P(x, y) = \textbf{true}\},$$

dove x appartiene allo spazio degli input del problema ed y a quello delle soluzioni del problema, mentre P è un predicato calcolabile con un algoritmo che richiede tempo finito.

Esempio 25.6 (Terminazione). Posto $x = (A, z)$, il problema della TERMINAZIONE si può esprimere come

$$L_{\text{TERMINA}} = \{x : \exists\, y\; P(x, y) = \textbf{true}\},$$

dove $P(x, y)$ è il predicato "l'algoritmo A eseguito con dato d'ingresso z termina dopo aver eseguito y istruzioni", che è chiaramente calcolabile in tempo finito. □

Un illustre problema il cui complemento è semidecidibile è il DOMINO ILLIMITATO, in cui si richiede se sia possibile ricoprire l'intero quadrante positivo $\mathbb{N} \times \mathbb{N}$ del piano, anziché una parte finita $[0, n] \times [0, n]$. Se la risposta è NO, allora deve esistere un'area R finita che non può essere ricoperta, e per trovarla basta provare un numero finito di coperture. Se la risposta è SI, allora non si può dire nulla in generale. Né si deve pensare che i problemi semidecidibili siano i più difficili, poiché è possibile individuare una gerarchia infinita di problemi con proprietà di indecidibilità sempre crescenti! Per esempio, il DOMINO ILLIMITATO RICORRENTE dove, oltre a coprire l'intero quadrante positivo del piano, si chiede se una particolare tessera è utilizzata in un numero infinito di copie non è neanche semidecidibile.

È possibile definire una gerarchia infinita di classi di complessità, dove ogni nuova classe è definita per mezzo delle altre classi già definite, in modo analogo a quanto visto nel § 19 per la gerarchia polinomiale. Infatti, per $k \geq 0$ si può definire la classe Σ_k come la classe contenente tutti i linguaggi seguenti

$$L = \{x : \exists y_1 \forall y_2 \exists y_3 \forall y_4 \cdots Q y_k\; P(x, y_1, \ldots, y_k) = \textbf{true}\},$$

dove $Q = \forall$ se k è pari, mentre $Q = \exists$ se k è dispari, x appartiene allo spazio degli input del problema, $y_1, \ldots, y_k$ appartengono allo spazio delle soluzioni del problema, e P è un predicato calcolabile in tempo finito. Analogamente, si può definire la classe Π_k come la classe co-Σ_k, cioè quella contenente i linguaggi seguenti

$$L = \{x : \forall y_1 \exists y_2\; y_3 \exists y_4 \cdots Q y_k\; P(x, y_1, \ldots, y_k) = \textbf{false}\},$$

dove $Q = \exists$ se k è pari, mentre $Q = \forall$ se k è dispari, e $x, y_1, \ldots, y_k$, e P hanno le stesse proprietà enunciate precedentemente. Chiaramente, le classi Σ_0 e Π_0 sono tra loro identiche e corrispondono alla classe dei problemi decidibili, mentre Σ_1 coincide con la classe dei problemi semidecidibili e Π_1 con quella dei problemi che sono il complemento di quelli semidecidibili (come il DOMINO ILLIMITATO). Le classi Σ_k e Π_k formano la cosiddetta gerarchia aritmetica (o gerarchia di Kleene), nella quale valgono le seguenti proprietà di inclusione stretta:

Teorema 25.2. *Per ogni* $k \geq 0$, $\Sigma_k \cup \Pi_k \subset \Sigma_{k+1} \cap \Pi_{k+1}$. □

In altri termini, all'aumentare del numero k di quantificatori aumenta l'indecidibilità del problema. Né le difficoltà si arrestano qui, poiché esistono addirittura problemi che non possono essere risolti da nessun algoritmo, neanche in tempo infinito! Ad esempio, si consideri l'insieme **P** di tutte le **function** Pascal di una variabile

intera non negativa che restituiscono un valore booleano. Ovviamente ogni procedura in **P** è rappresentata con una stringa finita di simboli e, fissata una lunghezza n, il numero di procedure diverse rappresentate con stringhe lunghe n è finito. È quindi possibile porre in corrispondenza biunivoca l'insieme **P** con l'insieme $\mathbb{N}$ dei numeri naturali (per esempio considerando la stringa che rappresenta una procedura come un numero in una opportuna base: si veda l'Esercizio 25.1).

Sia **F** l'insieme di tutte le funzioni booleane parziali di una variabile intera non negativa. Ogni procedura Pascal in **P** calcolerà una certa funzione in **F**, ma ci saranno in generale più procedure che calcolano la stessa funzione di **F**. Infatti, basta aggiungere istruzioni inutili ad una procedura per ottenerne infinite altre che calcolano la stessa funzione. Se ne deduce che $|\mathbb{N}| = |\mathbf{P}| \leq |\mathbf{F}|$. Vediamo che in effetti vale il minore stretto, $|\mathbf{P}| < |\mathbf{F}|$, e che quindi ci sarà qualche funzione non calcolata con procedure in **P**.

Sia **T** il sottoinsieme di **F** contenente solo funzioni booleane totali, cioè definite per ogni argomento. Se per assurdo fosse $|\mathbf{T}| = |\mathbb{N}|$, allora sarebbe possibile assegnare ad ogni funzione di **T** un indice progressivo e quindi sarebbe

$$\mathbf{T} = \{f_0, f_1, f_2, \ldots, f_i, \ldots\}.$$

Ciascuna funzione f_i può essere rappresentata con una sequenza binaria infinita

$$f_i = 01000111010101010\cdots$$

dove l'n-esimo bit è uguale ad $f_i(n)$. Ma allora si potrebbe costruire una tabella infinita contenente le $f_i(n)$ per ogni i ed n:

n	$=$	0	1	2	3	$\cdots$	k	$\cdots$
$f_0(n)$	$=$	0	1	1	0	$\cdots$	0	$\cdots$
$f_1(n)$	$=$	1	0	1	0	$\cdots$	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$
$f_k(n)$	$=$	1	1	0	1	$\cdots$	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$

La funzione g così definita

$$g(n) = \begin{cases} 0, & \text{se } f_n(n) = 1, \\ 1, & \text{se } f_n(n) = 0, \end{cases}$$

appartiene a **T**, ma non può comparire nella tabella, poiché differisce da ogni funzione nella tabella per il valore che si trova nella diagonale principale. Se ne conclude che $|\mathbf{P}| = |\mathbb{N}| < |\mathbf{T}| \leq |\mathbf{F}|$, e che esistono funzioni non calcolabili! Non solo, ma il risultato appena dimostrato è molto più forte.

Innanzitutto, osserviamo che la funzione g esclusa è anch'essa totale. Pertanto, quand'anche si restringesse l'attenzione al calcolo delle sole funzioni booleane totali, nella speranza di evitare programmi che, calcolando funzioni parziali, possano entrare in un ciclo infinito, non si potrebbero mai calcolare tutte le funzioni totali. Detto in altri termini, per poter calcolare funzioni booleane totali occorre poter calcolare anche funzioni booleane parziali, e quindi bisogna abituarsi all'idea di convivere con programmi che possono ciclare all'infinito.

In secondo luogo, anziché la diagonale principale della tabella, si potrebbe considerare una qualsiasi linea spezzata che tocchi ogni riga ed ogni colonna della tabella una e una sola volta. Per ognuna di queste linee spezzate si potrebbe definire una funzione totale analoga alla funzione g precedente, che abbia cioè un valore opposto ad ogni punto della linea. Poiché ci sono infinite linee spezzate di questa sorta, se ne conclude che esistono infinite funzioni booleane totali non incluse nella tabella e quindi infinite funzioni booleane totali non calcolabili e quindi, a maggior ragione, infinite funzioni parziali a valori interi oppure reali!

La tecnica di dimostrazione usata è stata introdotta da Cantor per dimostrare che i numeri reali non sono "numerabili", cioè che non è possibile assegnare ad ogni numero reale in $\mathbb{R}$ un indice progressivo: $r_0, r_1, r_2, \ldots, r_i, \ldots$ (o equivalentemente, che $\mathbb{R}$ non può essere messo in corrispondenza biunivoca con $\mathbb{N}$). Tale tecnica è detta di diagonalizzazione, ed è largamente impiegata, insieme a quella della riduzione, per dimostrare risultati di non calcolabilità come quello che è stato appena presentato.

25.3 Esercizi

Esercizio 25.1 (Numerabilità delle procedure). Si dimostri che è possibile porre in corrispondenza biunivoca l'insieme **P** delle procedure con l'insieme $\mathbb{N}$ dei numeri naturali, considerando la stringa che rappresenta una procedura come un numero in una opportuna base.

Per semplicità, assumiamo che le procedure siano descritte per mezzo di stringhe binarie, come nel caso di programmi scritti in "codice macchina" (ovviamente ogni stringa binaria rappresenta una procedura, anche se non necessariamente sintatticamente "corretta" né semanticamente "sensata"). Data una stringa binaria b, siano $|b|$ la sua lunghezza ed $N(b)$ il numero intero rappresentato da b. Allora il numero naturale associato a b è dato dalla funzione $C(b) = 2^{|b|} + N(b)$. Viceversa, dato un numero naturale n, la funzione inversa $C^{-1}(n)$ restituisce la stringa binaria b tale che $|b| = \lfloor \log n \rfloor$ ed $N(b) = n - 2^{|b|}$. Per esempio, indicata con λ la stringa vuota, la corrispondenza biunivoca è illustrata nella seguente tabella:

b	$\lvert b\rvert$	$N(b)$	$C(b) = 2^{\lvert b\rvert} + N(b)$
λ	0	0	1
0	1	0	2
1	1	1	3
00	2	0	4
01	2	1	5
10	2	2	6
11	2	3	7
000	3	0	8
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Esercizio 25.2 (Numerabilità coppie di naturali). Si dimostri che è possibile porre in corrispondenza biunivoca l'insieme delle coppie di numeri naturali con l'insieme dei numeri naturali. Si generalizzi successivamente per le terne, le quadruple, ..., le k-uple (con k costante fissata) di numeri naturali.

Esercizio 25.3 (Teorema di Cantor). Si dimostri, utilizzando la tecnica della diagonalizzazione, che l'insieme $\mathbb{R}$ dei numeri reali non è numerabile.

Esercizio 25.4 (Insieme delle parti). Si dimostri, utilizzando la tecnica della diagonalizzazione, che l'insieme di tutti i sottoinsiemi (o insieme delle parti) di $\mathbb{N}$ non è numerabile.

Esercizio 25.5 (Gerarchia di Kleene). Si collochino i seguenti problemi nella gerarchia di Kleene, esprimendoli nella opportuna forma

$$L = \{x : Q_1 y_1\ Q_2 y_2 \cdots Q_k y_k\ P(x, y_1, \ldots, y_k) = \mathbf{true}\},$$

dove i Q_i, $1 \leq i \leq k$, sono quantificatori esistenziali e universali alternati:

a. "Data la descrizione di un qualsiasi algoritmo A, A termina se eseguito con dato 0?"
b. "Data la descrizione di un qualsiasi algoritmo A, c'è un dato d'ingresso x per il quale A termina?"
c. "Data la descrizione di un qualsiasi algoritmo A, la funzione calcolata da A è totale?"
d. "Date le descrizioni di due qualsiasi algoritmi A e B, A e B terminano sugli stessi dati di ingresso?"

Capitolo 26

Macchine di Turing

Nei capitoli precedenti, sono stati considerati algoritmi per risolvere svariati problemi computazionali. Pur non avendo fornito una definizione rigorosa di "problema", si è visto come questo possa essere inteso matematicamente in termini di "linguaggio", cioè di un insieme formato da stringhe legali di simboli, oppure, equivalentemente, di "funzione", cioè di una corrispondenza tra dati di ingresso legali e dati di uscita legali. Per specificare algoritmi, sono state progettate procedure efficienti, scritte in un "linguaggio di programmazione ad alto livello", il Pascal, che sono interpretabili ed eseguibili senza possibilità di ambiguità alcuna da parte di un "calcolatore", di natura umana o elettronica. Si è anche visto come certi problemi siano intrinsecamente difficili o addirittura insolubili.

A questo punto, possono sorgere alcune domande legittime: i linguaggi di programmazione esistenti sono tutti equivalenti, o ce ne sono alcuni "più potenti" e/o "più semplici" di altri? Ci sono algoritmi descrivibili in un linguaggio, per esempio il Pascal, ma non descrivibili in un altro, o viceversa? È possibile che certi problemi oggi irrisolvibili possano essere risolti in futuro con altri linguaggi o con altri calcolatori? Ovviamente, la risposta a queste domande è intimamente legata ai concetti stessi di "algoritmo" e di "computazione", che saranno approfonditi in questo capitolo.

26.1 L'analisi di Turing

Nel 1936, prima ancora che fossero costruiti i primi calcolatori elettronici, il matematico britannico A. Turing, nel tentativo di rispondere alle precedenti domande, analizzò molto dettagliatamente il modo di agire di una persona che esegue calcoli ed arrivò alla definizione di un semplicissimo modello di calcolo che porta il suo nome. Vediamo per sommi capi l'argomentazione di Turing, schematizzata anche nella Fig. 26.1.

Memoria ausiliaria

a. Una persona che esegue un calcolo utilizza solitamente fogli di carta, disponibili in quantità illimitata, per annotare i risultati intermedi;
b. Per semplificarne l'utilizzo, si può supporre che ogni foglio sia diviso in caselle, come i fogli dei quaderni di scuola;

c. Il fatto che i fogli siano bidimensionali è del tutto ininfluente ai fini della computazione; pertanto, si può supporre che ogni foglio sia costituito da una striscia di carta unidimensionale;
d. Dalle considerazioni (a)-(c), segue che si può utilizzare un nastro infinito di carta suddiviso in caselle.

Alfabeto

e. Il calcolo avviene utilizzando un insieme di simboli (alfabeto), disponibili in quantità finita, poiché se ci fossero infiniti simboli allora alcuni sarebbero arbitrariamente "vicini" e quindi "indistinguibili";
f. La finitezza dell'alfabeto non è restrittiva, perché con esso si possono rappresentare infinite (numerabili) situazioni utilizzando sequenze di simboli (stringhe);
g. Mentre due simboli dell'alfabeto possono essere confrontati in un unico passo, stringhe molto lunghe devono essere confrontate in più passi.

Lettura e scrittura

h. Per effettuare il calcolo, occorre leggere e scrivere simboli utilizzando un dispositivo di lettura/scrittura;
i. Si può supporre che i simboli letti e/o scritti siano contenuti nelle caselle del nastro, uno per casella;
l. Senza perdere in generalità, si può supporre che non sia possibile leggere e/o scrivere più di un simbolo per volta.

Memoria di controllo

m. Per effettuare il calcolo occorre una memoria di controllo;
n. Il numero degli stati della memoria di controllo deve essere finito, poiché se ci fossero infiniti stati allora alcuni sarebbero arbitrariamente "vicini" e quindi "indistinguibili";
o. La finitezza degli stati non è restrittiva, perché stati più complessi possono essere evitati scrivendo stringhe di simboli sul nastro.

Operazioni elementari

p. Una operazione elementare consiste in una variazione del nastro e del controllo;
q. Si può supporre di modificare il contenuto di una casella solo dopo averla "osservata";
r. Si può supporre di non eccedere una certa "distanza" tra due caselle "osservate" e tale distanza può essere assunta unitaria.
s. Da tutte le considerazioni precedenti, segue che una operazione elementare consiste nella lettura di un simbolo contenuto in una casella, nella sua sostituzione con un altro simbolo, nel cambiamento dello stato del controllo in funzione del simbolo letto e del vecchio stato in cui si trovava il controllo, e nel predisporsi ad esaminare una casella immediatamente adiacente.

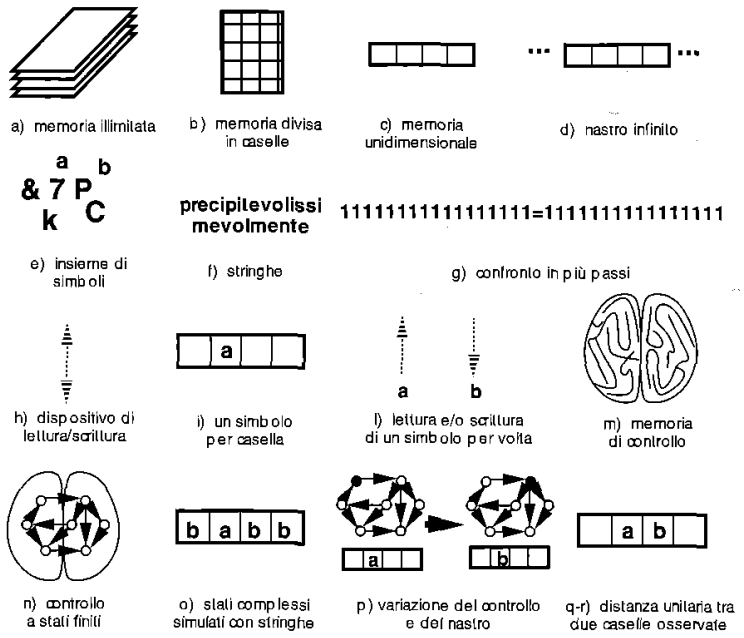


Figura 26.1: L'analisi di Turing.

La Macchina

Una "Macchina di Turing" M è una quintupla: $M = (K, V, s, t, F)$, dove K è un insieme finito di stati; V è un alfabeto finito di simboli, uno dei quali, indicato con $\emptyset$, è lo "spazio" o "bianco"; s è un particolare stato di K, detto stato iniziale; F è un *sottoinsieme di stati di K, detti stati finali*; $t : V \times (K - F) \rightarrow K \times V \times \{D, S\}$ è una funzione totale, detta funzione di transizione. $t(a, q) = (p, b, m)$ significa: "se lo stato attuale è q ed è letto il simbolo a dalla casella del nastro attualmente sotto il dispositivo di lettura/scrittura, allora il controllo si sposta nello stato p, viene scritto b al posto di a, e il dispositivo di lettura/scrittura si sposta di una casella a sinistra, se $m = S$, *o a destra, se* $m = D$".

Diagramma di transizione

Una macchina di Turing può essere rappresentata con un grafo orientato etichettato, detto diagramma di transizione, i cui nodi corrispondono agli stati K e i cui archi corrispondono alle transizioni, in modo che esiste l'arco (q, p) con etichetta "$a/b, m$" se e solo se $t(a, q) = (p, b, m)$. Nel disegnare tale grafo, il nodo s è solitamente evidenziato con una freccia, mentre i nodi in F sono doppiamente cerchiati, e da essi non esce alcuna freccia. Per comodità, se ci sono due archi (i, j) con etichette diverse,

si disegna una sola freccia, uscente da i ed entrante in j, avente due etichette. Inoltre, sono ammessi archi che escono ed entrano nello stesso nodo.

Computazione

La macchina si trova inizialmente nello stato s, con il dispositivo di lettura/scrittura posizionato sul primo simbolo a sinistra della stringa x di ingresso, mentre tutte le caselle del nastro, tranne quelle contenenti i simboli di x, contengono il simbolo $\emptyset$. La computazione procede effettuando le transizioni indicate dal diagramma, terminando quando il controllo entra in uno stato finale. Se il problema da risolvere è di tipo decisionale, allora ci saranno esattamente due stati finali, uno di "accettazione" e uno di "rifiuto", mentre se la macchina deve produrre un risultato diverso, questo sarà dato dalla più lunga stringa y di caratteri consecutivi tutti diversi da $\emptyset$ che si trovano a destra della casella sulla quale è posizionato il dispositivo di lettura/scrittura quando la macchina raggiunge uno stato finale.

Esempio 26.1 (Macchina di Turing). Consideriamo una macchina di Turing T per verificare che una stringa x formata da simboli a e b contenga lo stesso numero di a e di b, con tutte le a che precedono le b. Se x è la stringa vuota λ, essa viene accettata. Altrimenti, la macchina deve controllare che la stringa inizi con una a; in caso affermativo, può cancellare tale simbolo, sostituendolo con $\emptyset$, e scandire da sinistra verso destra la parte rimanente della stringa, fino ad individuarne l'ultimo simbolo. Tale simbolo è quello che precede il primo $\emptyset$ a sinistra del dispositivo di lettura/scrittura. Verificato che tale simbolo sia una b, la macchina può sostituirlo con $\emptyset$, e ripetere il procedimento, ma scandendo la stringa rimasta da destra verso sinistra, e scambiando l'ordine dei confronti tra a e b. Il diagramma di transizione di questa macchina è mostrato nella Fig. 26.2, insieme ad un esempio di computazione per $x =$ aabb. □

26.2 La Tesi di Church-Turing

Data l'estrema semplicità delle macchine di Turing, sembrerebbe lecito aspettarsi l'esistenza di problemi risolubili (o equivalentemente di linguaggi riconoscibili, oppure funzioni calcolabili) con procedure Pascal, ma non con macchine di Turing. Inoltre, parrebbe estremamente arduo trovare una macchina di Turing che calcolasse la funzione universale INTERPRETA. Invece, fortunatamente, ogni problema risolubile in Pascal lo è anche con le macchine di Turing, ed è abbastanza agevole, purché si sia acquisito un po' d'esperienza, progettare una macchina di Turing universale!

Esempio 26.2 (Macchina di Turing universale). La macchina universale U può ricevere la descrizione della funzione di transizione t di un'altra macchina $M = (K, V, s, t, F)$, codificandola come sequenza L di "quintuple", tale che (a, q, p, b, m) sta in L se e solo se $t(a, q) = (p, b, m)$. Ogni stato in K può essere codificato con una stringa binaria, utilizzando opportuni simboli per separare tra loro stringhe che codificano stati e quintuple. La sequenza L occuperà solo una porzione finita del nastro di U, e la rimanente parte del nastro potrà essere utilizzata da U per "simulare" lo stato corrente di M, la posizione del dispositivo di lettura/scrittura di M e il nastro di M. U potrà così interpretare le istruzioni di M ed eseguire le stesse operazioni che eseguirebbe M su un certo dato d'ingresso x, producendo lo stesso risultato. □

È facilmente verificabile che ogni algoritmo descrivibile con una macchina di Turing può essere descritto anche con una procedura Pascal. In effetti, è un facile

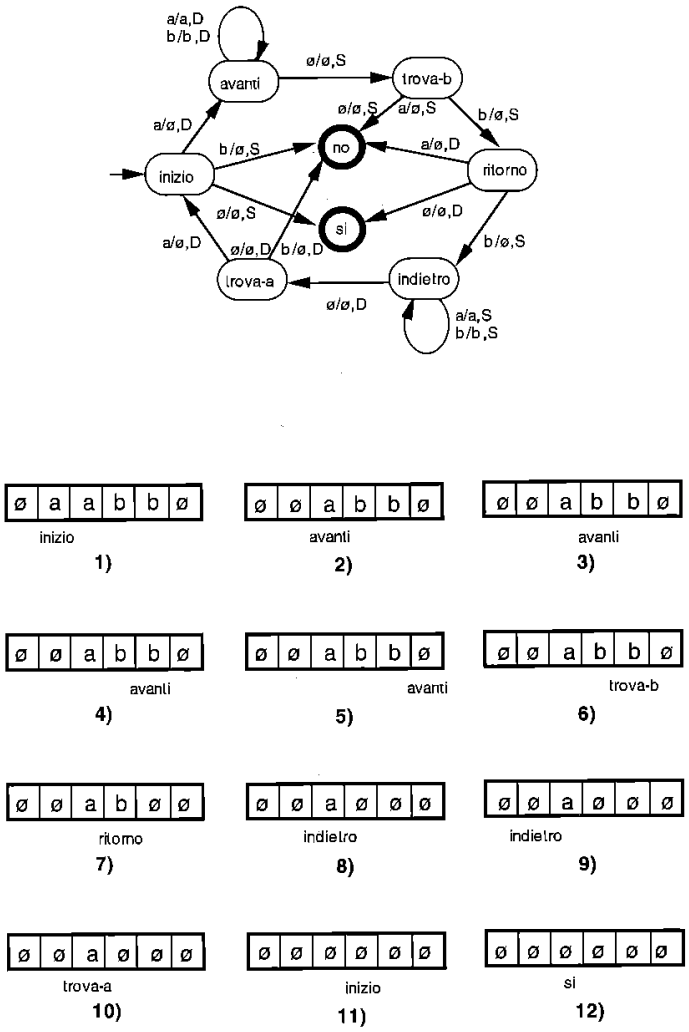


Figura 26.2: Una macchina di Turing T per riconoscere stringhe formate dallo stesso numero di a e di b, con tutte le a che precedono tutte le b. Esempio di riconoscimento della stringa $x =$ aabb.

esercizio "simulare" la macchina di Turing universale U con una procedura Pascal che riceva in ingresso le "quintuple" di una qualsiasi macchina di Turing M, le memorizzi in una matrice avente $|K|$ righe e $|V|$ colonne, e ne simuli il nastro infinito con una lista bidirezionale realizzata con puntatori, alla quale sono aggiunte caselle alle due estremità via via che servono. D'altro canto, anche se non è di immediata verifica, ogni algoritmo descrivibile con procedure Pascal può essere descritto con una macchina di Turing! Non solo, ma tutti coloro che, per vie diverse, hanno studiato finora i concetti di algoritmo e di computazione, definendo linguaggi di programmazione e modelli di calcolo apparentemente diversi tra loro, non hanno fatto che ridefinire la stessa classe di procedimenti!

In altri termini, pare che i concetti informali di algoritmo e di computazione possano essere "catturati" in una teoria matematica rigorosa. In effetti, tutti i diversi linguaggi di programmazione e modelli di calcolo definiti sinora (p.e. le macchine di Turing, il Pascal, le grammatiche a struttura di frase, il formalismo di Kleene, i sistemi di Post, le Random Access Machines, il Lisp, il Fortran, il Prolog, il C, il Java, il Cobol) sono stati dimostrati equivalenti tra loro, nel senso che risolvono esattamente la stessa classe di problemi (quelli risolvibili), ovvero riconoscono esattamente la stessa classe di linguaggi (quelli decidibili), ovvero calcolano esattamente la stessa classe di funzioni (quelle calcolabili)!

Una tesi accreditata e generalmente accettata, nota come Tesi di Church-Turing, afferma che tale equivalenza vale anche per ogni linguaggio di programmazione o modello di calcolo "ragionevole" che possa mai venire in mente in futuro. L'unico requisito richiesto è la possibilità di programmare la funzione INTERPRETA, e quindi avere a disposizione una quantità infinita di memoria: incrementi qualitativi alla struttura di una macchina o alle istruzioni di un linguaggio di programmazione servono solo ad abbassare il tempo di esecuzione o a rendere più agevole la programmazione.

Non solo, ogni istruzione elementare in uno qualsiasi dei formalismi noti finora può essere "simulata" in tempo polinomiale con una o più istruzioni elementari in un altro formalismo. In altri termini, tutti questi formalismi sono polinomialmente equivalenti. Una estensione universalmente accettata della Tesi di Church-Turing, nota come Tesi della Computazione Sequenziale, afferma che ogni linguaggio di programmazione o modello di calcolo "ragionevole" che possa mai venire in mente in futuro e che esegua le istruzioni sequenzialmente, cioè una alla volta, sia polinomialmente equivalente alle macchine di Turing, e quindi ad ogni altro formalismo noto finora. In altri termini, la classe $\mathbb{P}$ è "stabile", nel senso che non risente del modello di calcolo utilizzato (al più può variare soltanto il grado del polinomio).

26.3 Generalizzazioni e restrizioni

La definizione di macchina di Turing data precedentemente è soltanto una delle molte possibili. Sono state proposte molte altre definizioni, introducendo generalizzazioni o restrizioni varie, ma tutte queste definizioni alternative sono state provate tra loro equivalenti.

Esempio 26.3 (Macchine di Turing generalizzate). Aumentando il numero di nastri, il numero di dispositivi di lettura/scrittura, e il numero di dimensioni di ogni nastro non si aumenta la "potenza" di una ordinaria macchina di Turing con un solo nastro monodimensionale e un solo dispositivo di lettura/scrittura, ma se ne migliora l'"efficienza"! È stato dimostrato che ogni macchina di un tipo può essere simulata in tempo polinomiale con una macchina di un altro tipo. Pertanto, problemi risolubili

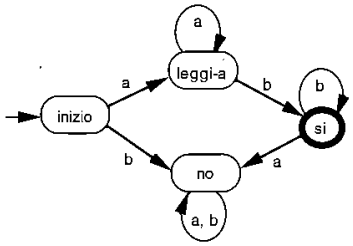


Figura 26.3: Un automa a stati finiti per riconoscere stringhe formate da una sequenza di almeno una a seguita da una sequenza di almeno una b.

in tempo polinomiale, cioè con un numero di transizioni polinomiale nella lunghezza $|x|$ della stringa di ingresso, restano tali qualunque sia il tipo di macchina usata (ovviamente, cambia il grado del polinomio). □

Esempio 26.4 (Macchina di Turing a nastro semi-infinito). Anche assumendo che il nastro sia infinito in una sola direzione, anziché in entrambe, la "potenza" non cambia. Infatti, una macchina T a nastro infinito in entrambe le direzioni può essere "simulata" con una macchina T' a nastro semi-infinito "tagliando" il nastro di T in un punto arbitrario, e formando il nastro di T' in modo che le caselle di posto pari corrispondano a quelle di T che stavano alla destra del taglio, mentre quelle di posto dispari corrispondano a quelle che stavano alla sinistra del taglio. T' simula una transizione di T effettuando uno spostamento di due caselle (mediante due transizioni), ed ogniqualvolta tenta di oltrepassare l'inizio del nastro passa dalla prima casella pari alla prima casella dispari, o viceversa, scambiando la destra con la sinistra. □

In contrasto, se si impone che una macchina di Turing possa scandire il nastro in una sola direzione, p.e. da sinistra verso destra, allora si ottiene una macchina molto meno potente, detta comunemente automa a stati finiti. Poiché una macchina siffatta non può tornare indietro, ogni casella non può essere esaminata più di una volta. Se il problema da risolvere è di tipo decisionale, ciò che l'automa scrive sul nastro è irrilevante, poiché non può più tornare indietro a leggerlo. Inoltre, una volta esaminata tutta la stringa, l'automa si trova di fronte ad una porzione infinita di nastro, contenente solo simboli $\emptyset$, che può ignorare. Pertanto, si può assumere che l'automa legga soltanto, utilizzi una porzione finita di nastro (quella contenente la stringa x d'ingresso) e si arresti appena ha esaminato tutta la stringa. Per convenzione, si può assumere che ci sia un solo stato finale e che tale stato dia risposta SI (se l'automa si arresta in un qualsiasi altro stato la risposta è NO) e che possano esistere transizioni che escono da tale stato finale.

Esempio 26.5 (Automa a stati finiti). La Fig. 26.3 illustra un automa a stati finiti per verificare che una stringa x formata da simboli a e b contenga un numero positivo di a e un numero positivo di b, con tutte le a che precedono tutte le b. □

Un automa, potendo solo leggere e muovendosi sempre nella stessa direzione, non può mai entrare in un ciclo infinito e termina sempre una computazione dopo $|x|$ transizioni! Pertanto, non esiste alcun "automa universale". Infatti, la memoria di un automa è formata da un numero finito $|K|$ di stati, e quindi esso non può "contare" oltre $|K|$.

Esempio 26.6 (Automi non "contano"). Non esiste alcun automa che riconosca le stringhe dell'Esempio 26.1. Infatti, si supponga che un tale automa esista (ed abbia p.e. n stati) e si consideri la stringa x contenente $n+1$ simboli a seguiti da $n+1$ simboli b. Leggendo le prime $n+1$ a, l'automa transiterà due volte per un medesimo stato, p.e. p. Ma cancellando la porzione di x compresa tra i due passaggi per p si ottiene una stringa x' che contiene meno di $n+1$ a ed esattamente $n+1$ b. Poiché l'automa non può distinguere tra i due passaggi in p, esso si comporterà nello stesso modo sia per x che per x', accettandole entrambe o rifiutandole entrambe. Ma questa è una contraddizione, poiché l'automa deve accettare x e rifiutare x'. □

26.4 Riduzioni mediante macchine di Turing

Per trovare limitazioni superiori della complessità di un problema, è conveniente utilizzare linguaggi di programmazione "ad alto livello", come il Pascal, che permettano di strutturare i dati su cui operare e di progettare algoritmi molto efficienti. Viceversa, per provare limitazioni inferiori della complessità, è conveniente usare modelli di calcolo "a basso livello", come le macchine di Turing, che permettano di provare agevolmente riduzioni che richiedano tempo limitato.

In particolare, l'estrema semplicità delle macchine di Turing e il fatto che modelli di calcolo diversi si possano simulare tra loro in tempo polinomiale, fanno delle macchine di Turing il modello ideale per provare risultati molto generali come il Teorema di Cook-Levin [cfr. § 18] che adesso ci accingiamo a dimostrare.

Innanzitutto, una macchina di Turing è non deterministica se è possibile scegliere tra più transizioni possibili da effettuare in corrispondenza dello stesso stato q del controllo e dello stesso simbolo a in lettura. In altri termini, la funzione

$$t : V \times (K - F) \to 2^{K \times V \times \{D,S\}}$$

è una funzione per cui può accadere contemporaneamente che $t(a,q) = (p,b,m)$ e $t(a,q) = (p',b',m')$. Ciò significa che nel diagramma di transizione ci possono essere più archi uscenti dallo stesso stato q marcati con etichetta "$a/\ldots,\ldots$" (un'analoga definizione vale anche per gli automi, che sono un caso particolare delle macchine di Turing).

Teorema di Cook-Levin. Ogni linguaggio $L \in \mathbb{NP}$ si riduce in tempo polinomiale al DOMINO LIMITATO.

Dimostrazione. Sia T una macchina di Turing non deterministica che si arresta in al più $p(|x|)$ passi, con p polinomio nella lunghezza $|x|$ della stringa x in input. Si supponga senza perdere in generalità che $T = (K, V, s, t, F)$ sia a nastro semi-infinito con un unico stato di r accettazione, ma eventualmente più di uno stato di rifiuto. Costruiamo un insieme di dati per il DOMINO LIMITATO tale che la macchina T accetta $x \in L$ in al più $p(|x|)$ passi se e solo se il DOMINO LIMITATO ha risposta SI.

Poiché T si arresta in al più $p(|x|)$ passi, non verranno utilizzate più di $p(|x|)$ caselle del nastro. Pertanto, imponiamo di ricoprire il quadrato $[0,p+1] \times [0,p+1]$, le cui righe corrispondono alle transizioni effettuate in una computazione e le cui colonne corrispondono alle caselle del nastro. Definiamo un insieme D di tessere nel modo seguente:

1. ogni riga di tessere "codifica" la configurazione del nastro, lo stato interno, e la posizione del dispositivo di lettura/scrittura di T;

2. è possibile passare da una riga di tessere ad una riga immediatamente superiore solo in modo consistente con la funzione t di transizione di T;
3. $[0, p+1] \times [0, p+1]$ è ricopribile con copie di tessere in D se e solo se T si arresta in al più $p(|x|)$ passi accettando x.

Per comodità, codifichiamo i colori usati nelle tessere con stringhe di simboli che tengano conto del simbolo dell'alfabeto contenuto in una casella del nastro, dello stato interno del controllo, della posizione del dispositivo di lettura/scrittura e del numero m di transizioni effettuate fino ad un certo istante. Se la configurazione delle $p = p(|x|)$ caselle del nastro al passo m-esimo è pari ad $s_1 s_2 \cdots s_i \mathtt{a} s_{i+2} s_{i+3} \cdots s_p$ il dispositivo di lettura/scrittura è sulla casella $(i+1)$-esima e il controllo è nello stato q, allora definiamo opportunamente i colori nei triangoli superiori delle tessere in modo che la m-esima riga della copertura di $[0, p+1] \times [0, p+1]$ sia come mostrato nella Fig. 26.4a). Sia \$ un simbolo che non compare in V. La definizione completa delle tessere è la seguente:

a. *codifica dell'input x.* Sia $|x| = h$, con $x = a_1 a_2 \cdots a_h$. Allora D contiene le tessere mostrate nella Fig. 26.4b). La tessera d da porre in basso a sinistra ha il colore "\$0" nel triangolo superiore e il colore "0" nel triangolo destro. Per ciascun a_i che compare in x, $2 \le i \le h$, c'è una tessera col colore "$a_i 0$" nel triangolo superiore, "$i-1$" nel triangolo sinistro, e "i" nel triangolo destro. Per a_1, invece, il colore nel triangolo superiore è "$sa_1 0$", poiché T si trova all'inizio nello stato s col dispositivo di lettura posizionato su a_1. Inoltre, per ogni j, con $h+1 \le j \le p$ c'è una tessera con colore "$\emptyset 0$" nel triangolo superiore, "$j-1$" in quello sinistro e "j" in quello destro. Infine, vi è un'ulteriore tessera con "\$0" in alto e "p" a sinistra.
b. *codifica della funzione di transizione t.* Se $t(a,q) = (k,b,D)$, allora, per ogni $m < p-1$, c'è una tessera con colore "qam" nel triangolo inferiore, "k" in quello destro, e "$bm+1$" in quello superiore. Viceversa, se $t(a, q) = (k, b, S)$, allora, per ogni $m < p-1$, c'è una tessera con colore "qam" nel triangolo inferiore, "k'''" in quello destro, e "$bm+1$" in quello superiore. Si noti che i due colori "k" e "k'''" sono distinti (si veda la Fig. 26.4c).
c. *codifica dello stato interno e del dispositivo di lettura/scrittura.* Per ogni simbolo $a \in V$, per ogni stato $q \in K$, e per ogni $m < p-1$, D contiene due tessere del tipo mostrato nella Fig. 26.4.d. Entrambe le tessere hanno colorato il triangolo superiore con "$qam+1$" e quello inferiore con "qam". Una delle due ha colorato il triangolo sinistro con "k'''", mentre l'altra ha colorato quello destro con "k".
d. *codifica delle caselle non esaminate.* Per ogni simbolo $a \in V \cup \{\$\}$ e per ogni $m < p+1$, D contiene una tessera con triangolo superiore colorato "$am+1$" e triangolo inferiore colorato con "am" (si veda la Fig. 26.4.e).
e. *codifica della terminazione.* Infine, per ogni simbolo $a \in V$ e per ogni $m < p+1$, D contiene una tessera con triangolo superiore colorato "$ram+1$" e triangolo inferiore colorato "ram" (si veda la Fig. 26.4.f).

Poiché il numero delle tessere in D è chiaramente polinomiale in p, $|K|$ e $|V|$, la riduzione richiede tempo polinomiale. Si noti che le tessere definite in (e) permettono di terminare la copertura della colonna relativa alla casella su cui si trova il dispositivo di lettura/scrittura qualora T si arresti al tempo $m < p+1$ e nello stato di accettazione r. Se T si arresta in uno stato in $F - \{r\}$, allora la copertura di tale colonna non è possibile. È facile verificare che T accetta la stringa x in al più p passi se e solo se è

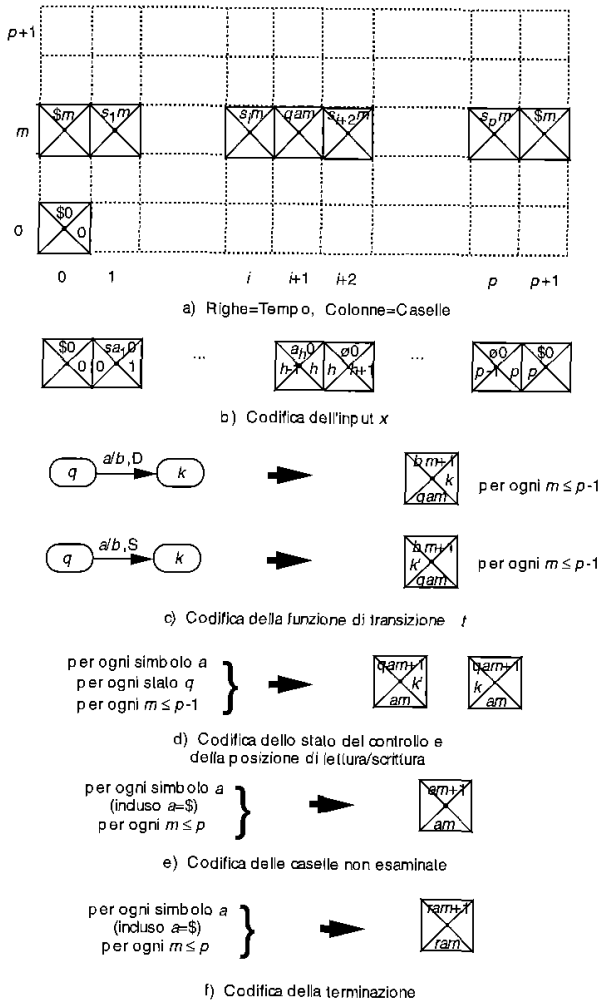


Figura 26.4: Tessere per il DOMINO LIMITATO nella riduzione usata per dimostrare il Teorema di Cook-Levin.

il particolare insieme di dati per il DOMINO LIMITATO ha una copertura. Pertanto, il teorema è così dimostrato.[1] □

Esempio 26.7 (Riduzione). Un esempio della riduzione usata precedentemente per la macchina di Turing T dell'Esempio 26.1 e la stringa $x =$ aabb è mostrato nella Fig 26.5. Poiché T accetta x in 11 passi, si è scelto $p = 12$ (i nomi degli stati di T sono stati abbreviati per semplicità). □

In modo analogo a quanto visto nella dimostrazione del Teorema di Cook-Levin, ma evitando di "codificare" nei colori il tempo m e lo stato di accettazione r, e considerando l'intero nastro semi-infinito anziché una sua porzione finita, si può dimostrare che esiste una copertura di $\mathbb{N} \times \mathbb{N}$ se e solo se la macchina di Turing T non si arresta su x, e che quindi il complemento del DOMINO ILLIMITATO è semidecidibile.

26.5 Esercizi

Esercizio 26.1 (Palindroma). Una stringa è palindroma se si legge nello stesso modo sia da sinistra verso destra che da destra verso sinistra (per esempio: radar). Si progetti una macchina di Turing che riconosce stringhe palindrome formate da a e b.

Esercizio 26.2 (Numero di a). Si progetti una macchina di Turing che riceve in ingresso stringhe formate da a e b e restituisce in uscita il numero di a contenute nella stringa in ingresso.

Esercizio 26.3 (Quadruple). Si dimostri che una definizione equivalente della funzione di transizione di una macchina di Turing può basarsi sull'uso di quadruple anziché quintuple. Formalmente, posto $t : V \times (K - F) \to K \times V \cup \{D, S\}$, si ha che $t(a, q) = (p, z)$, dove z indica un nuovo simbolo da scrivere nella casella attualmente scandita dal dispositivo di lettura/scrittura oppure uno spostamento del dispositivo stesso su una casella adiacente.

Esercizio 26.4 (Numero pari di a). Si progetti un automa a stati finiti che riconosce stringhe formate da a e b che contengono un numero pari di a.

Esercizio 26.5 (Vacuità). Si progetti un algoritmo di complessità polinomiale per decidere se il linguaggio accettato da un automa a stati finiti deterministico con n stati ed alfabeto di k simboli è vuoto (cioè non contiene alcuna stringa) oppure no. (Suggerimento: si formuli il problema come visita di un grafo).

Esercizio 26.6 (Finitezza). Si progetti un algoritmo di complessità polinomiale per decidere se il linguaggio accettato da un automa a stati finiti deterministico con n stati ed alfabeto di k simboli contiene un numero finito di stringhe oppure no. (Suggerimento: si formuli il problema come visita di un grafo).

Esercizio 26.7 (DOMINO ILLIMITATO). Si dimostri che il complemento del DOMINO ILLIMITATO è semidecidibile, modificando la riduzione vista nella dimostrazione del Teorema di Cook-Levin.

[1] Nelle macchine di Turing a nastro semi-infinito, di solito si impone che la prima casella del nastro contenga un carattere "speciale", p.e. "@", in modo che la macchina si accorga quando è arrivata alla fine del nastro ed eviti di uscirne fuori. Per semplicità si è evitato questo "trucco" nella dimostrazione, poiché si può sempre assumere che il primo simbolo dell'input x sia proprio uguale a @.

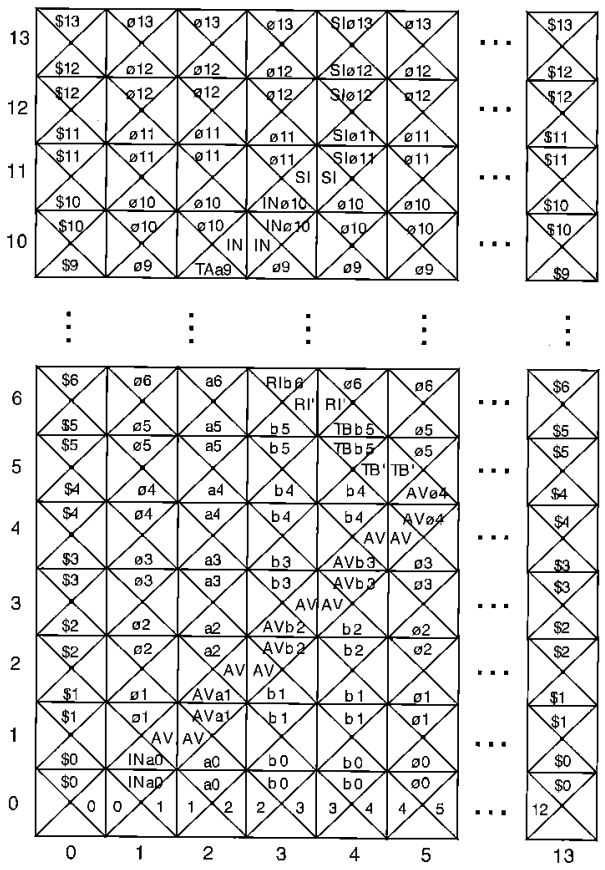


Figura 26.5: Riduzione per la macchina di Turing T dell'Esempio 26.1 con stringa d'ingresso $x = \texttt{aabb}$ ($p = 12$).

Parte quarta

Algoritmi paralleli

Capitolo 27

Algoritmi PRAM

Si consideri la seguente funzione Pascal per il calcolo della sommatoria di n numeri $a_1, a_2, \ldots, a_n$, contenuti in un vettore $a[1..n]$:

```
function SOMMATORIA(var a: vettore; n: integer): integer;
    var i, b: integer;
    begin
        b := a[1];
        for i := 2 to n do
            b := b + a[i];
    end;
```

La computazione procede nell'ordine evidenziato dalle parentesi:

$$((\cdots((a[1]+a[2])+a[3])+\cdots)+a[n-1])+a[n]$$

D'altro canto, per la proprietà associativa dell'operatore di somma, il risultato di $\sum_{i=1}^{k} a[i]$ non è indispensabile per calcolare $\sum_{i=k+1}^{n} a[i]$, e la computazione potrebbe benissimo procedere sommando nel seguente modo, assumendo n pari:

$$(a[1]+a[2])+(a[3]+a[4])+\cdots+(a[n-1]+a[n]).$$

Avendo a disposizione $n/2$ "esecutori" simultaneamente, ciascuna somma $a[i]+a[i+1]$, per ogni i dispari, potrebbe essere eseguita in tempo costante contemporaneamente alle altre. Pertanto, il problema sarebbe riconducibile in $O(1)$ tempo ad uno "più piccolo" in cui occorre sommare soltanto $n/2$ numeri. Iterando il procedimento e dimezzando via via la dimensione del problema, si riuscirebbe a calcolare la somma desiderata in tempo totale $O(\log n)$, che rappresenta un netto miglioramento rispetto al tempo $O(n)$ della funzione SOMMATORIA!

Un modello di computazione in cui vi sono più esecutori (o processori) che lavorano contemporaneamente è detto parallelo, mentre un modello in cui vi è un solo esecutore è detto sequenziale. Gli algoritmi descritti in un linguaggio di programmazione atto ad esplicitare le operazioni che possono essere eseguite simultaneamente sono detti algoritmi paralleli, in contrapposizione agli algoritmi sequenziali che sono stati trattati nei capitoli precedenti.

Il più semplice modello di computazione parallela è il cosiddetto modello PRAM (per "Parallel Random Access Machine"), in cui ci sono più processori che eseguono in modo sincrono la stessa istruzione e la memoria è globale e condivisa tra tutti

i processori. In particolare, ogni processore esegue la medesima istruzione ma su dati diversi (per esempio tutti i processori eseguono un'addizione, ma mentre un processore somma x ed y, un secondo processore somma a e b, ed un terzo processore somma a ed z). È possibile selezionare un sottoinsieme di processori che eseguano l'istruzione, mentre gli altri restano inattivi. I processori attivi sono sincronizzati, nel senso che tutti eseguono la stessa istruzione simultaneamente, e l'istruzione successiva non è iniziata finché tutti i processori non hanno terminato l'esecuzione dell'istruzione precedente.

Per esprimere istruzioni da eseguire in parallelo, è utilizzato il costrutto seguente:

"**for all** i **where** $inf \leq i \leq sup$ **do in parallel** $operazione_i$"

che significa: "si esegua $operazione_i$ contemporaneamente per ogni i compreso tra inf e sup".

Nel modello PRAM, i dati sono contenuti nella memoria globale comune, che è condivisa tra tutti i processori. La memoria è utilizzata dai processori per scambiarsi dati in $O(1)$ tempo. È infatti sufficiente che un processore modifichi con un'istruzione di scrittura il valore di una variabile della memoria e che con una successiva istruzione di lettura un altro processore legga il valore della medesima variabile. Si distinguono le seguenti varianti del modello PRAM:

1. EREW (per "Exclusive Read, Exclusive Write"), dove l'accesso contemporaneo alla stessa locazione di memoria da parte di più processori non è mai ammesso;
2. CREW (per "Concurrent Read, Exclusive Write"), dove l'accesso contemporaneo alla stessa locazione è ammesso per sola lettura;
3. CRCW (per "Concurrent Read, Concurrent Write"), dove è ammesso l'accesso contemporaneo sia in lettura che in scrittura.

In questo capitolo, è considerato pressoché esclusivamente il modello EREW, dalla maggioranza ritenuto il più "realistico", mentre l'impiego dei modelli CREW e CRCW è lasciato come curiosità in un paio di esercizi finali. Il modello EREW è utilizzato per risolvere efficientemente il problema della SOMMATORIA, delle SOMME PREFISSE, e dell'ORDINAMENTO, dove i dati sono contenuti in vettori e matrici. Inoltre, sono illustrate due tecniche generali (il "pointer jumping" e il "ciclo Euleriano") per elaborare liste ed alberi realizzati con puntatori.

27.1 Somme prefisse

Si consideri l'algoritmo parallelo introdotto precedentemente per il calcolo della sommatoria di n numeri. Si assuma per semplicità che n sia una potenza di due e che i numeri siano indicati $a_n, a_{n+1}, \ldots, a_{2n-1}$ anziché $a_1, a_2, \ldots, a_n$. Per meglio comprendere la computazione dell'algoritmo parallelo, si immagini che ciascun numero a_i sia contenuto nella foglia di un albero binario completo di $2n - 1$ nodi, indicati secondo la numerazione di uno "heap" [cfr. § 7], dove la radice ha l'indice 1 ed i figli sinistro e destro del generico nodo i hanno indici $2i$ e $2i + 1$, rispettivamente. Il problema può essere risolto in parallelo effettuando una fase ascendente su tale albero immaginario, dove si sale dal penultimo livello su fino alla radice e in parallelo, per ogni nodo i allo stesso livello, si calcola in a_i la somma dei figli a_{2i} e a_{2i+1}, che corrisponde alla sommatoria dei numeri che si trovano nelle foglie del sottoalbero

radicato nel nodo i, per $1 \leq i \leq n - 1$. Utilizzando un vettore $a[1..2n - 1]$, si ottiene la seguente funzione:[1]

```
function SOMMATORIA(var a: vettore; n: integer): integer;
  begin
    for k := log n − 1 downto 0 do
      for all i where 2^k ≤ i ≤ 2^(k+1) − 1 do in parallel
        a[i] := a[2i] + a[2i + 1];
    SOMMATORIA := a[1]
  end;
```

Anche se gli estremi degli indici del **for all** sono indicati per chiarezza con esponenziali, tali valori si possono in realtà individuare in tempo $O(1)$ usando una "maschera" di $\log n$ bit senza bisogno di calcolare esplicitamente gli esponenziali, ma effettuando traslazioni di sequenze di bit. Poiché il **for** esterno è ripetuto $O(\log n)$ volte, mentre il **for all** interno richiede tempo $O(1)$, la complessità di SOMMATORIA è $O(\log n)$ e il numero di processori impiegati è $O(n)$.

Si noti che un qualsiasi algoritmo parallelo che richieda T tempo e P processori impiega un ammontare TP di "lavoro", che rappresenta il tempo richiesto qualora il medesimo algoritmo venga sequenzializzato utilizzando un singolo processore. Nel caso della funzione SOMMATORIA, il lavoro risultante è $O(n \log n)$. Tale lavoro è però peggiore di quello del migliore algoritmo sequenziale che, impiegando $O(n)$ tempo con un solo processore, richiede $O(n)$ lavoro. Per ottenere lavoro ottimo anche per l'algoritmo parallelo, occorre ridurre il numero P dei processori, senza aumentare il tempo T, in modo che TP sia $O(n)$. Il numero di processori può essere ridotto ripartendo preventivamente gli n elementi del vettore in $O(n/\log n)$ gruppi di $O(\log n)$ elementi ciascuno, e facendo sommare sequenzialmente a ciascun processore tutti i numeri in ciascun gruppo. In questo modo, il problema originario di dimensione $O(n)$ viene ridotto, in tempo $O(\log n)$ e usando $O(n/\log n)$ processori, ad un problema di dimensione più piccola $O(n/\log n)$. Al problema ridotto può essere applicata la funzione SOMMATORIA, che quindi richiede $O(\log(n/\log n))$ tempo usando $O(n/\log n)$ processori. Il tempo complessivo T resta pertanto $O(\log n)$ mentre il numero di processori P scende ad $O(n/\log n)$. In questo modo, si ottiene lavoro ottimo TP che è $O(n)$.

Si consideri adesso il problema delle SOMME PREFISSE, che è una generalizzazione immediata della SOMMATORIA:

SOMME PREFISSE: "Dati n numeri interi $a_1, a_2, \ldots, a_n$, calcolare tutte le somme parziali $b_i = \sum_{j=1}^{i} a_j$ per $i = 1, 2, \ldots, n$".

Si assuma come prima che n sia una potenza di due, che i numeri siano indicati con $a_n, a_{n+1}, \ldots, a_{2n-1}$, e che essi siano contenuti nelle foglie di un albero binario completo di $2n - 1$ nodi, realizzato con un vettore $a[1..2n - 1]$. Il problema può essere risolto in parallelo effettuando due fasi. Nella fase ascendente, si esegue esattamente la stessa computazione della funzione SOMMATORIA, ponendo il risultato in b_1. Nella fase discendente, si calcola il valore b_i di ogni altro nodo come segue: il b_i

[1] Per semplicità, da ora in poi nelle procedure potranno essere usate variabili senza dichiarazioni ed espressioni matematiche (come "$n/\log n$", "$\lceil \log n \rceil$", "2^k", "$2i$" ecc.) senza curarsi di eventuali errore sintattici o di tipo, dato che il linguaggio parallelo impiegato, ancorché molto simile al Pascal, è puramente immaginario.

di un figlio destro è uguale a quello del padre, mentre il b_i di un figlio sinistro è uguale a quello del padre meno il valore a_{i+1} del fratello, per $2 \leq i \leq 2n - 1$. In questo modo, il valore di b_i viene ad essere uguale alla somma degli a_j dei nodi che si trovano allo stesso livello di b_i e che hanno indice j minore o uguale ad i. Pertanto $b_{n+i} = a_n + a_{n+1} + \cdots + a_{n+i}$, che è proprio il risultato desiderato, per $i = 0, 1, \ldots, n - 1$. L'algoritmo può essere descritto come segue:

```
procedure SOMMEPREFISSE(var a, b: vettore; n: integer);
  begin
    b[1] := SOMMATORIA(a, n);
    for k := 1 to log n do
      for all i where 2^k ≤ i ≤ 2^(k+1) − 1 do in parallel
        if odd(i)
          then b[i] := b[i/2]
          else b[i] := b[i/2] − a[i + 1];
  end;
```

La complessità della procedura è $O(\log n)$ utilizzando $O(n)$ processori, che però possono essere ridotti ad $O(n/\log n)$ con l'accorgimento spiegato precedentemente per la funzione SOMMATORIA, ottenendo un lavoro ottimo di $O(n)$.

27.2 Ordinamento

Un algoritmo parallelo per ordinare n numeri distinti $a_1, \ldots, a_n$, può essere progettato svolgendo un "torneo" tra tutte le coppie di numeri e vedendo, per ciascun a_i, con quanti altri numeri esso risulti "perdente" (cioè minore o uguale, a_i incluso) e con quanti altri risulti "vincente" (cioè maggiore). Il numero di perdenti dà la posizione in cui deve stare a_i nella sequenza ordinata. L'algoritmo risultante è il seguente:

```
procedure TORNEO(var a: vettore; n: integer);
  begin
    for all i, j where 1 ≤ i, j ≤ n do in parallel
      if a[i] ≤ a[j] then V[i, j] := 1 else V[i, j] := 0;
    for all j where 1 ≤ j ≤ n do in parallel
      P[j] := SOMMATORIA(j-esima colonna di V, n);
    for all j where 1 ≤ j ≤ n do in parallel
      a[P[j]] := a[j];
  end;
```

La procedura richiede $O(\log n)$ tempo ed n^2 processori, utilizzando però il modello CREW. Per utilizzare il modello EREW, occorre evitare che nello svolgersi del "torneo" avvengano conflitti in lettura sulla stessa cella di memoria. Infatti, il generico elemento $a[i]$ del vettore, per ciascun $1 \leq i \leq n$, è coinvolto contemporaneamente in ben n istruzioni diverse nel confronto "**if** $a[i] \leq a[j]$ **then** ... ", per $1 \leq j \leq n$. Lo stesso ovviamente vale anche per $a[j]$, per ciascun $1 \leq j \leq n$, che è coinvolto in n istruzioni diverse di confronto per $1 \leq i \leq n$. Questo inconveniente può essere eliminato effettuando n copie di ciascun $a[i]$ ed n copie di ciascun $a[j]$, in tempo $O(\log n)$, in modo che ciascuno degli n^2 processori possa accedere in lettura in tempo $O(1)$ ad un'unica coppia dei valori di $a[i]$ e $a[j]$ riservatagli.

Per convincersi di come ciò sia possibile, si consideri la seguente procedura, che effettua n copie di un dato d ponendole negli n elementi di un vettore $copia[1..n]$, con n potenza di due:

```
procedure REPLICA(var copia: vettore; d, n: integer);
  begin
    copia[1] := d;
    for k := 0 to log n − 1 do
      for all i where 1 ≤ i ≤ 2^k do in parallel
        copia[i + 2^k] := copia[i];
  end;
```

La procedura REPLICA richiede $O(\log n)$ tempo, poiché ad ogni iterazione il numero delle copie di d è raddoppiato, ed $O(n)$ processori. Pertanto, richiamandola in parallelo $2n$ volte, si possono effettuare n copie di ciascun $a[i]$ ed n copie di ciascun $a[j]$ in tempo $O(\log n)$ utilizzando $O(n^2)$ processori.

Infine, la procedura TORNEO può essere eseguita con il modello EREW sempre in tempo $O(\log n)$, ma usando solo $O(n^2/\log n)$ processori. Infatti, ciascun processore può eseguire sequenzialmente il calcolo di $O(\log n)$ $V[i, j]$, mentre il numero dei processori richiesto dalla procedura REPLICA per effettuare n copie di ciascun $a[i]$ può essere facilmente ridotto ad $O(n/\log n)$. Il lavoro di questa versione della procedura TORNEO risulta pertanto $O(n^2)$, che resta comunque peggiore del lavoro ottimo $O(n \log n)$ richiesto dal miglior algoritmo sequenziale.

27.3 Elaborazione di puntatori

Nei paragrafi precedenti abbiamo visto algoritmi PRAM per i problemi delle SOMME PREFISSE e dell'ORDINAMENTO dove i dati sono memorizzati in vettori e matrici. Illustriamo adesso due tecniche generali per elaborare liste ed alberi realizzati con puntatori.

Riconsideriamo il problema delle SOMME PREFISSE, dove gli n dati sono stavolta memorizzati in una lista (monodirezionale) e l'ordine col quale calcolarne le somme prefisse è determinato da come tali dati sono tra loro concatenati nella lista tramite i puntatori. Per semplificare la computazione parallela, si assuma che la lista sia in realtà realizzata con cursori (che sono equivalenti ai puntatori: cfr. § 2) e che ci siano esattamente n processori, tali che il processore i-esimo è responsabile di uno e un solo elemento della lista. Indichiamo con $a[i]$ il valore della lista gestito dal processore i-esimo, e con $succ[i]$ il cursore all'elemento successivo della lista, dove inizialmente $succ[i] = 0$ se e solo se $a[i]$ è l'ultimo valore della lista.

```
procedure SOMMEPREFISSE(var a, b, succ: vettore; n: integer);
  begin
    for all i where 1 ≤ i ≤ n do in parallel
      b[i] := a[i];
    for k := 1 to ⌈log n⌉ do
      for all i where 1 ≤ i ≤ n do in parallel
        if succ[i] ≠ 0 then begin
          b[succ[i]] := b[i] + b[succ[i]]; succ[i] := succ[succ[i]]
        end
  end;
```

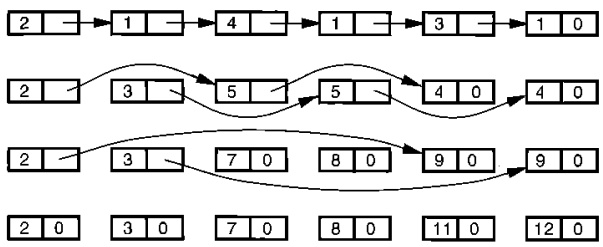


Figura 27.1: Calcolo delle somme prefisse con la tecnica del salto dei puntatori.

La tecnica utilizzata nel **for all** più interno è detta "pointer jumping" (cioè salto dei puntatori). Si osservi che con tale tecnica non ci sono mai conflitti da parte di più processori per accedere alla medesima locazione di memoria. Infatti, per $i \neq j$ vale sempre $succ[i] \neq succ[j]$ oppure $succ[i] = succ[j] = 0$, e questa proprietà è mantenuta invariante per l'intera computazione. Poiché tutti i valori di *succ* diversi da 0 sono distinti, non avviene mai alcun conflitto in lettura, ed il modello utilizzato è EREW. La procedura richiede $O(\log n)$ tempo ed n processori, poiché si parte con un'unica lista e ad ogni iterazione si ha un numero doppio di liste disgiunte, fino ad ottenere n liste di un solo elemento ciascuna. La correttezza della procedura deriva dal fatto che ad ogni iterazione ciascun valore $b[i]$ è uguale al vecchio valore di $b[i]$ più il valore di $b[j]$ contenuto nell'elemento che lo precede nell'insieme di liste disgiunte. Inoltre, lo stesso valore non è mai sommato due volte, poiché dopo che è stato sommato viene "scavalcato" dal salto dei puntatori.

Esempio 27.1 (Pointer jumping). La Fig. 27.1 illustra un esempio di esecuzione della procedura SOMMEPREFISSE, che utilizza la tecnica del salto dei puntatori, per una lista con $n = 6$ elementi. Poiché $\lceil \log 6 \rceil = 3$, vengono eseguite 3 iterazioni, ed i valori di b per ciascuna di esse sono mostrati nella figura. □

Il lavoro della procedura è $O(n \log n)$, che non è ottimo. È comunque possibile progettarne una versione che richiede $O(\log n)$ tempo ed $O(n/\log n)$ processori, e che quindi ha lavoro ottimo $O(n)$. Tale versione è piuttosto complicata e non può essere qui riportata.

La tecnica del pointer jumping è molto utile per risolvere non solo il problema delle SOMME PREFISSE, ma anche molti altri problemi. Vediamone un ulteriore utilizzo per calcolare il livello dei nodi di un albero binario realizzato con puntatori. A tal fine, la tecnica del salto dei puntatori è combinata con un'altra tecnica detta del "ciclo Euleriano". Ricordiamo che un ciclo Euleriano di un grafo orientato G è un ciclo che include ogni arco una e una sola volta e che G contiene un ciclo Euleriano se e solo se è connesso e per ciascun nodo il numero di archi entranti è uguale al numero di archi uscenti (cfr. Esercizio 15.3). Pertanto, dato un qualsiasi grafo non orientato G, il grafo orientato G' ottenuto sostituendo ogni arco di G con una coppia di archi orientati in verso opposto contiene sempre un ciclo Euleriano. Per calcolare le profondità di un albero binario B di n nodi, si considera un ciclo Euleriano del grafo orientato B' ottenuto da B, dove tale ciclo è rappresentato con

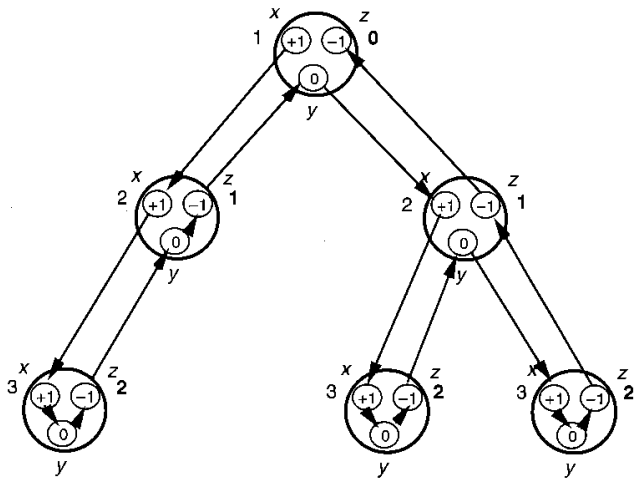


Figura 27.2: Utilizzo della tecnica del ciclo Euleriano per calcolare i livelli dei nodi di un albero binario. I valori degli elementi x, y e z sono mostrati nei cerchietti piccoli. I livelli dei nodi corrispondono ai valori delle somme prefisse degli elementi z e sono mostrati in grassetto in alto alla destra di ogni nodo.

una lista monodirezionale di $3n$ elementi sulla quale viene applicata la tecnica del salto dei puntatori.

In dettaglio, ogni nodo u di B è sostituito con tre elementi di una lista, che individuiamo con x, y, e z, in modo che i risultanti $3n$ elementi siano gestiti ciascuno da uno ed un solo processore. La lista è costruita in modo che:

1. l'elemento x del nodo u vale $+1$ e punta all'elemento x del figlio sinistro di u, se tale nodo esiste, oppure all'elemento y di u stesso, altrimenti;
2. l'elemento y del nodo u vale 0 e punta all'elemento x del figlio destro di u, se tale nodo esiste, oppure all'elemento z di u stesso, altrimenti;
3. l'elemento z del nodo u vale -1 e punta all'elemento y del padre di u, se u è un figlio sinistro, oppure all'elemento z del padre di u, se u è un figlio destro, oppure a 0, se u è la radice.

In questo modo, la lista che rappresenta il ciclo Euleriano ha inizio con l'elemento x della radice e, dopo aver seguito l'ordine anticipato dell'albero B, termina con l'elemento z della radice. È immediato verificare che, data la rappresentazione tramite puntatori (o cursori) di B, la lista può essere costruita in tempo $O(1)$. Eseguendo la procedura SOMMEPREFISSE su tale lista, è facile convincersi che il valore b di ciascun elemento z rappresenta il livello del corrispondente nodo u. Infatti, per come sono stati scelti i valori di x, y e z, la visita di un sottoalbero radicato in un qualsiasi nodo v ha come effetto di sommare 0 alla somma prefissa, poiché scendendo da v si somma 1, ma poi risalendo a v si sottrae 1. Pertanto, i livelli dei nodi di B possono essere calcolati in $O(\log n)$ tempo usando $O(n)$ processori.

Esempio 27.2 (Calcolo livelli dei nodi di un albero binario). La Fig. 27.2 illustra un esempio di utilizzo della tecniche del ciclo Euleriano e del salto dei puntatori per

calcolare i livelli dei nodi di un albero binario. I valori degli elementi x, y e z sono mostrati nei cerchietti piccoli all'interno di ogni nodo. I livelli dei nodi corrispondono ai valori delle somme prefisse degli elementi z e sono mostrati in grassetto in alto alla destra di ogni nodo. □

27.4 La Tesi della Computazione Parallela

Abbiamo visto come l'introduzione di più processori che lavorano in parallelo possa abbassare radicalmente la complessità in tempo di alcuni problemi. I problemi trattati nei paragrafi precedenti erano però già risolubili in tempo polinomiale con algoritmi sequenziali. Una legittima aspirazione sarebbe dunque quella di risolvere in tempo polinomiale con algoritmi paralleli problemi che sono intrattabili con algoritmi sequenziali. Purtroppo, una tesi universalmente accreditata, detta Tesi della Computazione Parallela fa svanire questa aspirazione!

La Tesi della Computazione Parallela asserisce che ogni problema risolubile in tempo polinomiale con un algoritmo parallelo può essere risolto in spazio polinomiale con un algoritmo sequenziale, e viceversa. Pertanto, la possibilità che esistano problemi sequenzialmente intrattabili che diventano trattabili in parallelo è intimamente legata alla possibilità che non esistano problemi intrinsecamente superpolinomiali che stanno nella classe $\mathbb{P}$-$\mathbb{SPAZIO}$. Come si ignora se $\mathbb{P} = \mathbb{NP}$ oppure no, così neanche si conosce se $\mathbb{P}$-$\mathbb{SPAZIO} = \mathbb{P}$ oppure no. La questione rimane pertanto irrisolta, anche se è opinione comune che $\mathbb{P}$-$\mathbb{SPAZIO}$ contenga problemi di complessità intrinsecamente superpolinomiale e che quindi problemi intrattabili sequenzialmente restino tali anche in parallelo. In effetti, è facile individuare algoritmi paralleli che risolvano in tempo polinomiale problemi NP-completi, ma tali algoritmi utilizzano sempre un numero superpolinomiale di processori!

In pratica, si considerano "trattabili" in parallelo quei problemi che sono risolubili con algoritmi paralleli (deterministici) che richiedono un numero di processori polinomiale nella dimensione del problema e tempo polinomiale nel logaritmo della dimensione del problema (o polilogaritmico). La classe di questi problemi è detta $\mathbb{NC}$, e come conseguenza della Tesi della Computazione Parallela, si ha che $\mathbb{NC}$ coincide con la classe dei problemi in $\mathbb{P}$ che sono risolubili con una quantità di memoria (esclusa quella per l'input e l'output) che è $O(\log^k n)$, con k costante. Ad esempio, sono problemi in $\mathbb{NC}$ tutti i problemi trattati nei paragrafi precedenti del presente capitolo. Una domanda che spesso viene posta a questo punto consiste nel chiedersi perché si imponga che il tempo debba essere proprio polilogaritmico, anziché semplicemente inferiore a quello del miglior algoritmo sequenziale che risolve lo stesso problema. Una risposta sensata è che, impiegando un numero polinomiale di processori anziché uno solo, si richiede di diminuire il tempo di computazione in modo "significativo", e quindi possibilmente in modo esponenziale. Non basta quindi che il tempo di computazione dell'algoritmo parallelo sia un polinomio, ancorché di grado più basso di quello sequenziale, ma si richiede che sia polilogaritmico, poiché la funzione logaritmo garantisce tale diminuzione esponenziale.

Come all'interno della classe $\mathbb{NP}$ c'è una classe di problemi "più difficili" rispetto al tempo polinomiale, quelli NP-completi, così all'interno della classe $\mathbb{P}$ c'è una classe di problemi "più difficili" rispetto allo spazio polilogaritmico, quella dei cosiddetti problemi LOG-SPAZIO-completi. Un problema in $\mathbb{P}$ è LOG-SPAZIO-completo se ogni altro problema in $\mathbb{P}$ è riducibile ad esso con una trasformazione che richiede spazio polilogaritmico. Come conseguenza della Tesi della Computazione Parallela, è altamente improbabile che anche un solo problema LOG-SPAZIO-completo possa essere

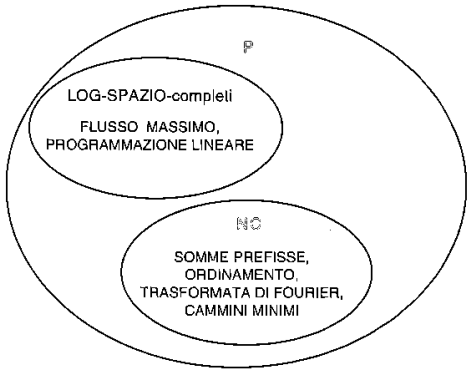


Figura 27.3: Presunte relazioni di contenimento tra sottoclassi di $\mathbb{P}$.

risolto da un algoritmo parallelo deterministico che richiede tempo polilogaritmico e un numero polinomiale di processori.

Un illustre problema LOG-SPAZIO-completo è il FLUSSO MASSIMO in forma decisionale (cfr. § 15). Un altro è

PROGRAMMAZIONE LINEARE: "Data una matrice A di elementi interi e di dimensione $m \times n$, ed un vettore b di m elementi interi, esiste un vettore x di n elementi reali tale che $Ax \leq b$?"

poiché include il FLUSSO MASSIMO come caso particolare. In contrasto, oltre alle SOMME PREFISSE ed all'ORDINAMENTO, altri problemi appartenenti ad $\mathbb{NC}$ sono la MOLTIPLICAZIONE DI MATRICI, i CAMMINI MINIMI, e la TRASFORMATA DISCRETA DI FOURIER (come vedremo nel capitolo seguente, dove saranno considerati i modelli di calcolo delle reti a grado limitato, che sono casi particolari dei modelli PRAM).

Come si ignora se $\mathbb{P} = \mathbb{NP}$ oppure no, così neanche si conosce se $\mathbb{P} = \mathbb{NC}$ oppure no. Ovviamente, se si trovasse un algoritmo parallelo con complessità temporale polilogaritmica che utilizzi una quantità polinomiale di processori per un solo problema LOG-SPAZIO-completo, allora se ne potrebbero trovare per tutti i problemi in $\mathbb{P}$ e quindi risulterebbe $\mathbb{P} = \mathbb{NC}$, assumendo vera la Tesi della Computazione Parallela. La presunte relazioni di contenimento tra le classi $\mathbb{P}$, $\mathbb{NC}$, e quella dei problemi LOG-SPAZIO-completi sono illustrate nella Fig. 27.3, assumendo $\mathbb{P} \neq \mathbb{NC}$.

27.5 Esercizi

Esercizio 27.1 (Sommatoria). Si scriva una procedura per effettuare la sommatoria di n numeri usando $O(\log n)$ tempo e $O(n/\log n)$ processori col modello EREW.

Si assuma per semplicità che sia n che $n/\log n$ siano potenze di due e che gli elementi da sommare siano indicati con $a_n, a_{n+1}, \ldots, a_{2n-1}$. Si esegue una prima fase in cui ogni processore effettua sequenzialmente la sommatoria di un gruppo di $\log n$ elementi. In seguito, si effettua la sommatoria parallela degli $n/\log n$ elementi, che contengono i risultati della prima fase, seguendo lo schema logico di un albero binario completo. Si usa il vettore $a[1..2n-1]$, contenente inizialmente i dati da sommare

in $a[n..2n-1]$ ed il risultato in $a[1]$. Si osservi che non sono usati necessariamente tutti i $2n-1$ elementi di a, ma che il lavoro è $O(n)$ e quindi ottimo.

```
procedure SOMMATORIA(var a: vettore; n: integer);
  begin
    for all j where 1 ≤ j ≤ n/ log n do in parallel begin
      a[n/log n + j − 1] := a[n + (j − 1)log n];
      for i := 1 to log n − 1 do
        a[n/log n + j − 1] := a[n/log n + j − 1] + a[n + (j − 1)log n + i];
    end;
    for k := log(n/ log n) downto 1 do
      for all j where 2^(k−1) ≤ j ≤ 2^k − 1 do in parallel
        a[j] := a[2j] + a[2j + 1];
  end;
```

Esercizio 27.2 (Somme prefisse). Si scriva una procedura per effettuare le somme prefisse di n numeri usando $O(\log n)$ tempo e $O(n/\log n)$ processori col modello EREW.

Esercizio 27.3 (Ordinamento). Si modifichi la procedura TORNEO per ordinare n numeri $a_1, \ldots, a_n$, non necessariamente distinti col modello CREW.

Esercizio 27.4 (Replica). Si modifichi la procedura REPLICA per effettuare n copie del dato d usando $O(\log n)$ tempo e $O(n/\log n)$ processori col modello EREW.

Esercizio 27.5 (Ordinamento). Si modifichi la procedura TORNEO per ordinare n numeri distinti $a_1, \ldots, a_n$, richiamando la procedura REPLICA dell'Esercizio 27.4, usando $O(\log n)$ tempo e $O(n/\log n)$ processori col modello EREW. (Suggerimento: si costruiscano due matrici R ed S tali che l'i-esima riga di R contenga n copie di a_i e la j-esima colonna di S n copie di a_j).

Esercizio 27.6 (Valutazione di un polinomio). Dato il polinomio $p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$, si vuole valutarlo nel punto q, cioè calcolare $p(q)$. Scrivere una funzione ottima per il modello EREW.

Si calcolano inizialmente le potenze $q^0, \ldots, q^{n-2}, q^{n-1}$ di q. Per far ciò, dato che $q^i = qq_{i-1}$ con $1 \le i \le n-1$, si considerano i prodotti prefissi b del vettore x contenente un 1 seguito da $n-1$ copie di q. La procedura per i prodotti prefissi si ottiene dalla procedura SOMMEPREFISSE dell'Esercizio 27.2 sostituendo l'assegnamento "$b[i] := b[i/2] - a[i+1]$" con "$b[i] := b[i/2]\ /\ x[i+1]$" ed utilizzando una procedura PRODOTTORIA al posto di SOMMATORIA, ottenuta sostituendo le operazioni "+" con "·" ed il vettore a con x. Siccome il modello usato è EREW, questa fase richiede n copie di q. Infine si calcolano i prodotti a_iq^i per $0 \le i \le n-1$ nel vettore r, e poi ne viene effettuata la sommatoria. Il tempo è $O(\log n)$ ed il numero di processori è $O(n/\log n)$. Poiché un polinomio si può valutare sequenzialmente in tempo ottimo $O(n)$, la funzione VALUTAPOLINOMIO richiede lavoro ottimo $O(n)$.


```
function VALUTAPOLINOMIO(var a: vettore; q: real): real;
  begin
    {copia q in x_n, ..., x_(2n−1) usando REPLICA};
    x_n := 1;
    PRODOTTIPREFISSI(x, n);
    for all j where 0 ≤ j ≤ n/ log n − 1 do in parallel
      for i := 0 to log n − 1 do
        r[n + j * n/log n + i] := a[j * n/log n + i] * b[n + j * n/log n + i];
    SOMMATORIA(r, n);
    VALUTAPOLINOMIO := r[1];
end;
```

Esercizio 27.7 ("And" prefissi). Data una sequenza $a_n, \ldots, a_{2n-1}$ di $n = 2^k$ booleani, si vuole calcolare la sequenza $b_n, \ldots, b_{2n-1}$, dove $b_i = a_n$ **and** $a_{n+1} \cdots$ **and** a_i, per $n \le i \le 2n-1$. Si scriva una procedura che richieda $O(\log n)$ tempo ed $O(n/\log n)$ processori con il modello EREW. (Suggerimento: si calcolino opportune somme prefisse).

Esercizio 27.8 (Rango). Data una lista di n elementi, si vuole calcolare per ciascun elemento della lista il numero di elementi che lo seguono. Si scriva una procedura efficiente con il modello EREW.

È sufficiente rovesciare il verso della lista (che può essere fatto in tempo $O(1)$ col modello EREW), associare ad ogni elemento della lista il valore 1, calcolare le somme prefisse di tale sequenza di n uni, e quindi sottrarre 1 ad ogni valore così calcolato.

Esercizio 27.9 (Diffusione). Dati una lista di n elementi ed un valore x, si vuole diffondere x a tutti gli elementi della lista. Si scriva una procedura efficiente con il modello EREW.

Esercizio 27.10 (Partizione). Data una lista di n elementi booleani, si vogliono formare due liste contenenti, rispettivamente, tutti i valori veri e tutti i valori falsi. Si scriva una procedura efficiente con il modello EREW.

Esercizio 27.11 (Dimensione sottoalberi). Dato un albero binario di n nodi, si vuole calcolare per ciascun nodo u il numero di nodi del sottoalbero radicato in u. Si scriva una procedura efficiente con il modello EREW. (Suggerimento: si calcoli la differenza di due opportuni valori nelle somme prefisse del ciclo Euleriano).

Esercizio 27.12 (Visite alberi binari). Si scrivano procedure efficienti con il modello EREW per calcolare gli ordini di visita anticipato, simmetrico, e differito di un albero binario.

Esercizio 27.13 (Punto interno ad un poligono). Siano dati un punto p ed un poligono P nel piano Cartesiano. Il poligono non sia necessariamente convesso ed abbia n lati. Si dimostri che p è interno a P se e solo se è dispari il numero di intersezioni tra lati di P che stanno "sopra" p e la retta verticale V passante per p. Si fornisca una procedura efficiente per determinare se il punto è interno al poligono oppure no usando il modello CREW.

Esercizio 27.14 (Radici). Sia data una foresta di alberi radicati realizzata con un vettore dei padri. Si vuole determinare per ciascun nodo u l'identità della radice dell'albero al quale appartiene u. Si fornisca una procedura efficiente usando il modello CREW. (Suggerimento: si utilizzi la tecnica del salto dei puntatori lungo tutti i percorsi foglia-radice).

Esercizio 27.15 ("And" con CRCW). Si fornisca una funzione per determinare in tempo O(1) l'**and** di un vettore $b[1..n]$ di n valori booleani usando il modello CRCW.

Si assume che se più processori vogliono scrivere contemporaneamente nella stessa locazione di memoria, allora devono cercare di scrivere lo stesso valore ed uno solo di tali processori riesce effettivamente a scrivere.

```
function AND-CRCW(var b: vettorebooleano; n: integer): boolean;
  begin
    x := true;
    for all j where 1 ≤ j ≤ n do in parallel
      if not b[j] then x := false;
    AND-CRCW := x;
  end;
```

Esercizio 27.16 (Massimo con CRCW). Si fornisca una procedura per determinare in tempo O(1) il massimo di un vettore $a[1..n]$ di n valori interi usando il modello CRCW. (Suggerimento: si effettui un "torneo" e poi si utilizzi la funzione AND-CRCW dell'Esercizio 27.15).

Capitolo 28

Algoritmi per reti a grado limitato

Nel modello PRAM introdotto nel capitolo precedente, i processori possono accedere simultaneamente ad una memoria globale condivisa da tutti ed una qualsiasi coppia di processori può comunicare in tempo O(1) attraverso tale memoria comune. Il modello PRAM è però di interesse puramente teorico, poiché l'attuale tecnologia non ne permette la costruzione. In pratica, affinché un modello parallelo sincrono sia effettivamente costruibile, occorre che la memoria non sia affatto condivisa e che i processori comunichino tra loro soltanto attraverso una prefissata rete di interconnessione. Tale rete può essere rappresentata con un grafo, in genere non orientato, i cui nodi corrispondono ai processori ed i cui archi corrispondono ad interconnessioni tra processori. In questo modo, due processori possono comunicare direttamente tra loro in tempo O(1) se e solo se c'è un arco che li collega. In quest'ottica, i processori del modello PRAM possono essere visti come se fossero interconnessi con un grafo completo, cioè con una cricca che li include tutti. Perché sia fisicamente costruibile, però, una rete di interconnessione non può essere un grafo completo! In pratica, detto g il grado massimo dei nodi, cioè il numero massimo di archi incidenti in un nodo, occorre che g sia "piccolo", possibilmente limitato superiormente da una costante (o al più da un logaritmo del numero di processori). Infatti, nel costruire fisicamente una rete di interconnessione, ciascun arco è realizzato con un cavetto che deve essere collegato a due processori, ed è chiaramente impensabile che la circuiteria di un singolo processore permetta la connessione di un numero elevato di cavetti. Per ragioni di efficienza, si richiede inoltre che il diametro della rete, cioè la distanza tra i due processori più lontani, sia "piccolo". Dati due processori, la distanza tra essi è pari al numero minimo di archi tra tutte le catene che li uniscono, ed il diametro d della rete è uguale alla distanza massima tra tutte le coppie di processori. In pratica, d rappresenta una limitazione superiore al tempo di comunicazione tra ogni coppia di processori, ed un diametro piccolo, per esempio logaritmico nel numero di processori, garantisce tempi di comunicazione veloci tra processori pur non connessi direttamente tra loro.

Pertanto, in un modello parallelo con rete di interconnessione a grado limitato, ci sono più processori che eseguono in modo sincrono la stessa istruzione, come per la PRAM, ma su dati diversi. La memoria non è però condivisa tra tutti, ma ogni processore ha una memoria ad esso riservata. Ogni processore può accedere in lettura/scrittura ai dati contenuti nella propria memoria e può accedere in sola lettura a quelli di un vicino, cioè di un processore ad esso connesso tramite un arco. Nelle procedure, viene ancora usato il costrutto "**for all** ... **where** ... **do in parallel** ...", dove si esplicitano gli indici dei processori coinvolti nell'esecuzione parallela delle istruzioni.

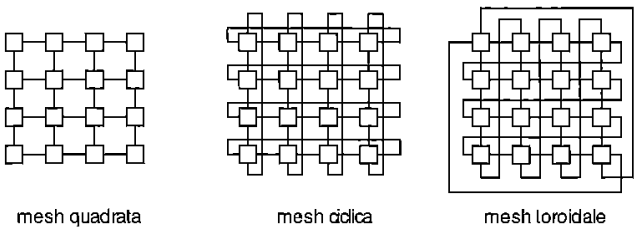


Figura 28.1: Una mesh quadrata, una ciclica e una toroidale, per $n = 16$ processori.

In questo capitolo, sono dapprima considerate le principali reti di interconnessione, come la mesh, la shuffle, la butterfly e l'ipercubo. Sono poi mostrate procedure efficienti per risolvere svariati problemi, come la SOMMATORIA (su mesh, shuffle e ipercubo), l'ORDINAMENTO tramite "mergesort bitonico" (su ipercubo), la MOLTIPLICAZIONE DI MATRICI e i CAMMINI MINIMI (su mesh e ipercubo), e la TRASFORMATA DI FOURIER (su butterfly e ipercubo). Poiché il modello PRAM include come caso particolare il modello con rete di interconnessione a grado limitato, tutte le procedure illustrate in questo capitolo possono essere eseguite anche sul modello PRAM con le medesime prestazioni, cioè impiegando lo stesso tempo e lo stesso numero di processori.

28.1 Reti di interconnessione

Le principali reti di interconnessione che prendiamo in considerazione sono la mesh, la shuffle, la butterfly e l'ipercubo. In una mesh (o "maglia") quadrata, gli n processori, con $n = p^2$ per un p intero positivo, sono disposti in una matrice quadrata di dimensione $\sqrt{n} \times \sqrt{n}$, dove il generico processore $P_{i,j}$, con $1 \leq i, j \leq \sqrt{n}$, è connesso con i processori $P_{i,j\pm1}$ e $P_{i\pm1,j}$, se esistono. Il grado massimo g di ogni nodo è 4, mentre il diametro d è $\sqrt{n}$. Si incontrano spesso due varianti della mesh quadrata: quella ciclica, dove sono connessi tra loro anche il primo e l'ultimo processore di ogni riga (e di ogni colonna), e quella toroidale, dove invece l'ultimo processore di ogni riga (o colonna) è connesso con il primo processore della riga (colonna) successiva, con la prima riga (colonna) considerata successiva della $\sqrt{n}$-esima. Tutte e tre le versioni presentano il vantaggio di una struttura estremamente semplice e regolare, al prezzo di un diametro piuttosto elevato.

Esempio 28.1 (Mesh). La Fig. 28.1 mostra una mesh quadrata, una mesh ciclica, ed una mesh toroidale per $n = 16$ processori. □

In una shuffle (spesso chiamata anche "perfect-shuffle" o "shuffle-exchange"), gli n processori, dove $n = 2^p$ per un p intero positivo, sono numerati (con indici binari di p bit) da 0 a 2^{p-1}, e vi sono due tipi di connessioni: quelle non orientate di scambio e quelle orientate di mescolamento. Le connessioni non orientate di scambio (o "exchange") sono tra i processori P_i e P_{i+1}, per ogni i pari, ovvero tra coppie di processori i cui indici in binario differiscono nel bit meno significativo. Le connessioni orientate di mescolamento (o "shuffle") escono dal processore P_i, $0 \leq i \leq n-2$, ed entrano nel processore P_j, dove $j = 2i \textbf{ mod } (n-1)$, ed inoltre ce n'è una che esce

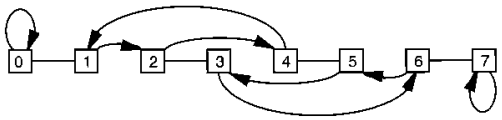


Figura 28.2: Una shuffle per $n = 8$ processori.

da P_{n-1} e rientra in P_{n-1} stesso. In altri termini, tali connessioni sono tra coppie di processori i cui indici in binario differiscono per una traslazione ciclica sinistra dei loro bit, dove il bit più significativo dell'indice i diviene il bit meno significativo di j e tutti gli altri bit sono scalati a sinistra di una posizione, ovvero la sequenza binaria $b_p b_{p-1} \cdots b_1$ diviene $b_{p-1} \cdots b_1 b_p$.

Esempio 28.2 (Shuffle). La Fig. 28.2 mostra una shuffle per $n = 8$ processori. □

Si osservi che, percorrendo per $p = \log n$ volte le connessioni di mescolamento a partire da un processore P_i, per $1 \leq i \leq n-2$, si ritorna al processore P_i stesso. Grazie alle connessioni di scambio, è però possibile uscire da tale ciclo e raggiungere un qualsiasi altro processore in $O(\log n)$ passi. Infatti, è facile verificare che, mentre il grado massimo g di ogni nodo è 3, il diametro d è proprio $O(\log n)$. Una interpretazione delle connessioni di mescolamento è la seguente. Sia $n = 8$ e si abbia a disposizione un mazzo ordinato di 8 carte da gioco, indicate con 0, 1, 2, 3, 4, 5, 6, 7. Suddividendo il mazzo in due mazzetti contenenti ciascuno la metà delle carte, si ottiene un mazzetto con 0, 1, 2, 3 ed un secondo mazzetto con 4, 5, 6, 7. Se i due mazzetti sono mescolati in modo "idealmente perfetto", ci si aspetta che ciascuna carta del primo mazzetto sia alternata con una carta del secondo mazzetto, ottenendo il seguente mazzo mescolato: 0, 4, 1, 5, 2, 6, 3, 7. Questo mazzo fornisce una permutazione del mazzo originale, che corrisponde proprio alle connessioni di mescolamento:

$$\begin{array}{cccccccc} 0 & 4 & 1 & 5 & 2 & 6 & 3 & 7 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

In una butterfly (o "farfalla") di grado k, gli n processori, con $n = (k+1)2^k$ per un k intero positivo, sono disposti in $k+1$ righe e 2^k colonne, ed il generico processore $P_{i,j}$, con $1 \leq i \leq k$, $0 \leq j \leq 2^k - 1$, è connesso con i processori $P_{i-1,j}$ e $P_{i-1,m}$, dove m è l'intero ottenuto complementando l'i-esimo bit più significativo della rappresentazione binaria di j. Il grado massimo g di ogni nodo è 4, mentre il diametro d è $O(k) = O(\log n)$. Si noti che se $P_{i,j}$ è connesso con $P_{i-1,m}$, allora $P_{i,m}$ è connesso con $P_{i-1,j}$, e la corrispondente topologia di interconnessione tra questi quattro processori ha una sagoma che ricorda appunto quella di una farfalla. Inoltre, i processori della riga 0 (o della riga k) possono essere visti come le foglie comuni a 2^k alberi binari completi le cui radici sono i processori della riga k (o della riga 0).

Esempio 28.3 (Butterfly). La Fig. 28.3 mostra una butterfly di rango $k = 3$ per $n = 32$ processori. □

In un ipercubo di dimensione k, con k intero non negativo, gli $n = 2^k$ processori sono numerati da 0 a $2^k - 1$, ed il generico processore P_i è connesso con tutti i processori P_j tali che le rappresentazioni binarie di i e j differiscono per il valore di un singolo bit. Il grado massimo g ed il diametro d sono entrambi uguali a $k = \log n$.

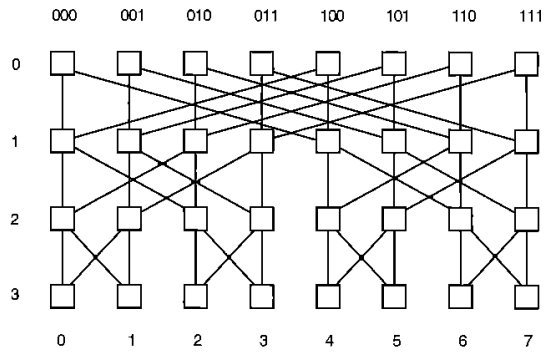


Figura 28.3: Una butterfly di rango $k = 3$ con $n = (k+1)2^k = 32$ processori.

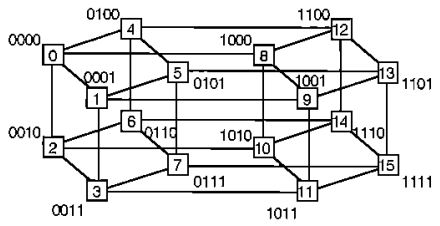


Figura 28.4: Un ipercubo di dimensione $k = 4$ ed $n = 2^4 = 16$ processori.

Esempio 28.4 (Ipercubo). La Fig. 28.4 mostra un ipercubo di dimensione $k = 4$, avente quindi $n = 16$ processori. Si noti che un ipercubo di dimensione k si ottiene da due ipercubi di dimensione $k - 1$, connessi lungo la dimensione k-esima. □

Poiché il grado g dell'ipercubo non è una costante, ne è stata introdotta una variante, detta CCC (per "cube-connected-cycles") che consiste in un ipercubo di dimensione k, dove però ognuno dei 2^k vertici dell'ipercubo è sostituito con un circuito di k processori, per un totale di $k2^k$ processori. Si indichi con $P_{i,j}$ l'i-esimo processore del circuito corrispondente al vertice j-esimo dell'ipercubo, con $1 \leq i \leq k$ e $0 \leq j \leq 2^k - 1$. Oltre alle connessioni dei circuiti, il generico processore $P_{i,j}$ è connesso con $P_{i,m}$, dove m è il numero intero ottenuto complementando l'i-esimo bit più significativo della rappresentazione binaria di j. In questo modo, il grado massimo g del CCC diventa 3. Si noti la seguente analogia tra il CCC e la butterfly: ad ogni connessione tra $P_{i,j}$ e $P_{i-1,m}$ della butterfly corrisponde una connessione tra $P_{i,j}$ e $P_{i,m}$ del CCC, in modo che $P_{i,j}$ e $P_{i,m}$ possono comunicare nella butterfly in due passi, seguendo dapprima la connessione tra $P_{i,j}$ e $P_{i-1,m}$ e poi quella tra $P_{i-1,m}$ e $P_{i,m}$ (o viceversa).

Esempio 28.5 (CCC). La Fig. 28.5 mostra un CCC di dimensione $k = 3$, avente quindi $n = 24$ processori. □

È possibile dimostrare che una qualsiasi computazione svolta da un ipercubo di dimensione k può essere effettuata nello stesso tempo da un CCC della stessa dimensione. Non solo, ma Galil e Paul (1981) hanno dimostrato che esiste un modello parallelo "universale", che può simulare ogni altro modello in cui g è $O(1)$ (come la mesh, la shuffle e la butterfly) con una "piccolissima" perdita di tempo ed usando "all'incirca" lo stesso numero di processori. Tale modello universale è basato su una rete per l'ORDINAMENTO che serve da "ufficio di smistamento" per scambiare dati tra i processori. Poiché Galil e Paul hanno mostrato che tale rete può essere equivalentemente un CCC, una shuffle, o una butterfly, ne segue che queste tre reti sono reti di interconnessione universali!

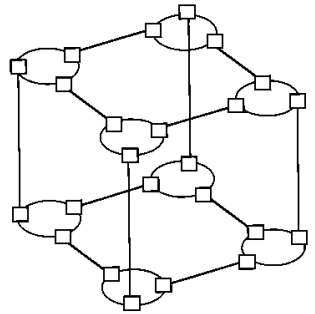


Figura 28.5: Un CCC di dimensione $k = 3$ ed $n = 3 \cdot 2^3 = 24$ processori.

28.2 Sommatoria

Nel progettare algoritmi paralleli per modelli di computazione con reti a grado limitato, occorre tenere conto della struttura di interconnessione specifica del modello utilizzato. Pertanto, si avranno algoritmi diversi (e con prestazioni diverse) per risolvere lo stesso problema con reti di interconnessione diverse. Vediamo ad esempio tre algoritmi paralleli per risolvere il problema della SOMMATORIA. Siano $a_0, a_1, \ldots, a_{n-1}$ gli n numeri da sommare, con $n = 2^k$, e si consideri un ipercubo di dimensione k, tale che il dato a_i sia contenuto inizialmente nel processore P_i, per $0 \leq i \leq n - 1$. In un algoritmo efficiente per calcolare la sommatoria, si considerano dapprima gli $n/2$ processori con indici minori (cioè quelli che hanno il bit più significativo a 0), i quali individuano un ipercubo di dimensione $k - 1$, ed ognuno di essi somma in parallelo il suo dato con quello contenuto nel corrispondente processore dell'altro ipercubo di dimensione $k - 1$ formato dagli $n/2$ processori con indici maggiori (cioè quelli che hanno il bit più significativo a 1).

In questo modo, si riduce in tempo $O(1)$ il problema di sommare n dati contenuti in un ipercubo di dimensione k al problema di sommare $n/2$ dati contenuti in un ipercubo di dimensione $k - 1$. Il procedimento viene iterato considerando gli $n/4, n/8, \ldots, n/n$ processori con indici minori, che corrispondono ad ipercubi di dimensione $k-2, k-3, \ldots, 0$, considerando cioè gli indici con anche il secondo, terzo, ..., ultimo bit più significativo a 0. Dopo $k = \log n$ iterazioni, il risultato è contenuto nel processore P_0.

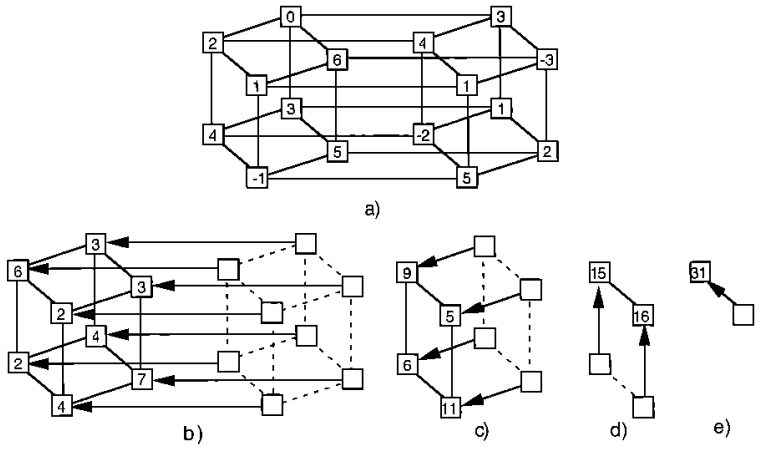


Figura 28.6: Esecuzione della procedura per la sommatoria su un ipercubo. a) Gli elementi da sommare. b)-e) Le quattro iterazioni della procedura.

La procedura è la seguente e richiede $O(\log n)$ tempo, utilizzando n processori.[1]

procedure SOMMATORIA-IPERCUBO($a_0, a_1, \ldots, a_{n-1}$);
  begin
    for $i := \log n - 1$ **downto** 0 **do begin**
      $h := 2^i$;
      for all j **where** $0 \le j \le h - 1$ **do in parallel**
        begin $b_j \leftarrow a_{j+h}$; $a_j := a_j + b_j$ **end**;
    end;
  end;

Esempio 28.6 (Sommatoria su ipercubo). La Fig. 28.6 mostra un esempio di computazione della sommatoria della sequenza di $n = 16$ elementi 2, 1, 4, −1, 0, 6, 3, 5, 4, 1, −2, 5, 3, −3, 1, 2, su un ipercubo di dimensione $k = 4$. Il risultato è uguale a 31 ed è contenuto nel processore P_0. □

Il lavoro della procedura SOMMATORIA-IPERCUBO è $O(n \log n)$, che può essere reso ottimo considerando una sequenza di $O(n \log n)$ elementi da sommare, e facendo preventivamente sommare a ciascun processore in modo sequenziale $O(\log n)$ elementi.

Risolviamo adesso lo stesso problema con una shuffle di $n = 2^k$ processori, assumendo ancora che a_i sia contenuto inizialmente nel processore P_i, per $0 \le i \le n - 1$.

[1]Per semplicità e chiarezza, da ora in poi le sequenze memorizzate distribuendo un elemento per ciascuna memoria riservata dei processori saranno denotate con $a_0, a_1, \ldots, a_{n-1}$ anziché con $a[0]$, $a[1], \ldots, a[n-1]$. Infatti, quest'ultima notazione è utilizzata dal Pascal per gli array e tende ad evocare una sequenza di elementi memorizzati in modo contiguo all'interno della medesima memoria anziché distribuiti tra più memorie. È indicata inoltre con "←" l'operazione di lettura di un dato da un processore adiacente.

A tal fine, è utile considerare due procedure SCAMBIA e MESCOLA, che sono eseguite in parallelo su tutti i processori. La procedura SCAMBIA(a_i) ha come effetto quello di scambiare in tempo $O(1)$ gli a_i dei processori di indice pari con quelli dei processori di indice dispari, in accordo alle connessioni di scambio. La procedura MESCOLA(a_i) ha invece l'effetto di spostare in tempo $O(1)$ gli a_i tra i processori in accordo alle connessioni di mescolamento.

Esempio 28.7 (SCAMBIA e MESCOLA). Sia $n = 8$. La procedura SCAMBIA, applicata alla sequenza 4, 0, 5, 1, 7, 3, 2, 6, sposta gli elementi adiacenti ottenendo la nuova sequenza 0, 4, 1, 5, 3, 7, 6, 2. La procedura MESCOLA, applicata alla medesima sequenza 4, 0, 5, 1, 7, 3, 2, 6, li sposta ottenendo 4, 7, 0, 3, 5, 2, 1, 6, come si può verificare osservando le connessioni di mescolamento della shuffle illustrata nella Fig. 28.2. □

La procedura per la sommatoria è molto semplice, e consiste di $\log n$ chiamate di MESCOLA e SCAMBIA alternate, che utilizzano una sequenza di appoggio $b_0, b_1, \ldots, b_{n-1}$. Il tempo richiesto è $O(\log n)$ usando n processori.

procedure SOMMATORIA-SHUFFLE($a_0, a_1, \ldots, a_{n-1}$);
  begin
    for $i := 1$ **to** $\log n$ **do**
      for all j **where** $0 \le j \le n - 1$ **do in parallel begin**
        MESCOLA(a_j);
        $b_j := a_j$;
        SCAMBIA(b_j);
        $a_j := a_j + b_j$
      end
  end;

Esempio 28.8 (Sommatoria su shuffle). La Fig. 28.7 mostra un esempio di computazione della sommatoria, per la stessa sequenza dell'Esempio 28.6, su una shuffle di $n = 16$ processori. Si noti che vengono effettuate le stesse somme, e nello stesso ordine, come nell'ipercubo! □

Risolviamo infine il problema della SOMMATORIA su una mesh quadrata. Siano $a_{1,1}, \ldots, a_{1,p}, a_{2,1}, \ldots, a_{2,p}, \ldots, a_{p,1}, \ldots, a_{p,p}$ gli $n = p^2$ elementi da sommare, tali che $a_{i,j}$ sia contenuto inizialmente nel processore $P_{i,j}$, per $1 \le i, j \le \sqrt{n}$. La procedura per la sommatoria consiste nel sommare gli elementi dell'ultima riga a quelli della penultima riga, poi sommare i risultati agli elementi della terzultima riga, risalendo così fino alla prima riga, dove infine sono sommati i risultati parziali da sinistra verso destra, ottenendo il risultato in $P_{1,1}$. Il tempo richiesto è $O(\sqrt{n})$ usando n processori. Il lavoro risultante è $O(n\sqrt{n})$, che può essere reso ottimo considerando una sequenza di $O(n\sqrt{n})$ elementi, e facendo preventivamente sommare in modo sequenziale $O(\sqrt{n})$ elementi ad ogni processore.

procedure SOMMATORIA-MESH($a_{1,1}, a_{1,2}, \ldots, a_{p,p}$);
  begin
    for $i := p - 1$ **downto** 1 **do**
      for all i, j **where** $1 \le j \le p$ **do in parallel**
        begin $b_{i,j} \leftarrow a_{i+1,j}$; $a_{i,j} := a_{i,j} + b_{i,j}$ **end**;
    for $j := p - 1$ **downto** 1 **do**
      for all i, j **where** $i = 1$ **do in parallel**
        begin $b_{i,j} \leftarrow a_{i,j+1}$; $a_{i,j} := a_{i,j} + b_{i,j}$ **end**
  end;

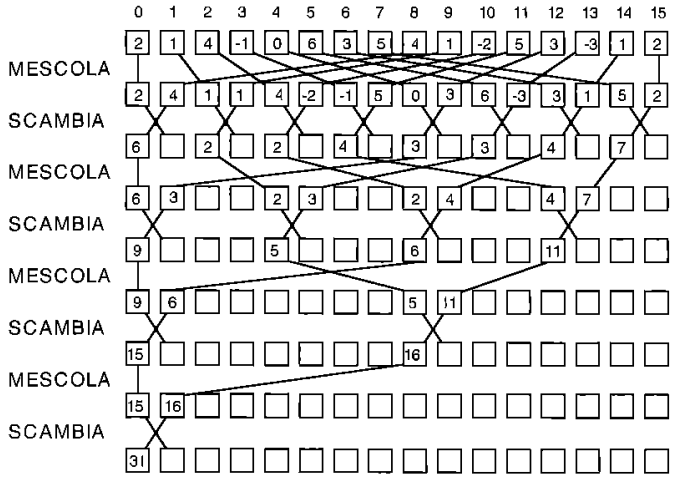


Figura 28.7: Esecuzione della procedura per la sommatoria su una shuffle.

Esempio 28.9 (Sommatoria su mesh). La Fig. 28.8 mostra un esempio di computazione della sommatoria, per la stessa sequenza dell'Esempio 28.6, su una mesh di $n = 16$ processori. □

28.3 Mergesort bitonico

Uno degli algoritmi paralleli più efficienti e più utilizzati per risolvere il problema dell'ORDINAMENTO è il "mergesort bitonico", introdotto da Batcher negli anni '60. Una sequenza $a_0, a_1, \ldots, a_{n-1}$ è unimodale se esiste un indice i, $0 \leq i \leq n-1$, tale che $a_0 \leq a_1 \leq \ldots \leq a_i$ e $a_i \geq a_{i+1} \geq \cdots \geq a_{n-1}$. Una sequenza $a_0, a_1, \ldots, a_{n-1}$ è bitonica se è unimodale oppure è ottenibile con una traslazione ciclica da una sequenza unimodale.

Esempio 28.10 (Sequenze bitoniche). La sequenza 2, 5, 7, 8, 10, 3, 1, 0 è unimodale, e quindi anche bitonica. Anche le sequenze 3, 1, 0, 2, 5, 7, 8, 10 e 7, 8, 10, 3, 1, 0, 2, 5, sono bitoniche, perché ottenute dalla prima sequenza unimodale traslando ciclicamente gli elementi sottolineati. Ovviamente, qualsiasi sequenza ordinata è bitonica. □

Una importante proprietà delle sequenze bitoniche è la seguente. Data la sequenza bitonica $A = a_0, a_1, \ldots, a_{2m-1}$, allora le due sequenze

$A_{\text{MIN}} = \min\{a_0, a_m\}, \min\{a_1, a_{m+1}\}, \ldots, \min\{a_{m-1}, a_{2m-1}\}$

$A_{\text{MAX}} = \max\{a_0, a_m\}, \max\{a_1, a_{m+1}\}, \ldots, \max\{a_{m-1}, a_{2m-1}\}$

sono entrambe bitoniche, ed ogni elemento di A_{MIN} è minore o uguale ad ogni elemento di A_{MAX}.

Esempio 28.11 (continua). Data la sequenza A = 7, 8, 10, 3, 1, 0, 2, 5, si ottengono $A_{\text{MIN}} = 1, 0, 2, 3$ e $A_{\text{MAX}} = 7, 8, 10, 5$. □

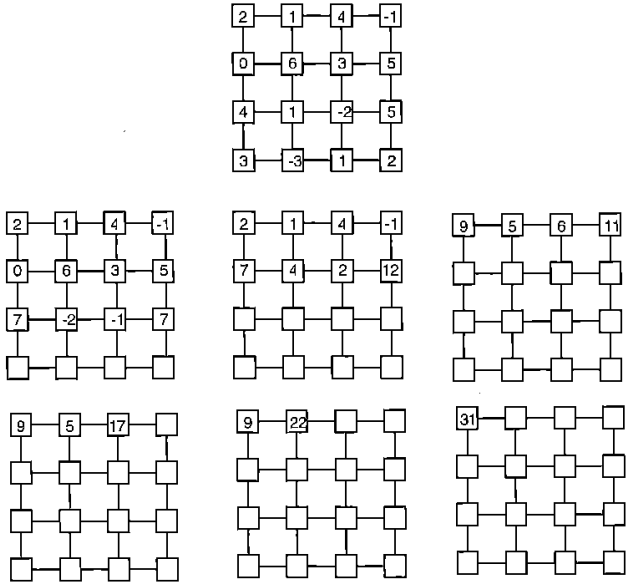


Figura 28.8: Esecuzione della procedura per la sommatoria su una mesh.

Questa proprietà suggerisce immediatamente un algoritmo parallelo di tipo "divide et impera" per ordinare una sequenza bitonica A di lunghezza n, con n potenza di 2. L'algoritmo consiste, se $n > 1$, nel dividere A in A_{MIN} e A_{MAX} e riapplicare ricorsivamente il procedimento alle due sequenze bitoniche A_{MIN} e A_{MAX}. Con una PRAM, la divisione può essere effettuata in tempo $O(1)$ usando $O(n)$ processori, le due chiamate ricorsive possono essere eseguite in parallelo, ed il lavoro di ricombinazione è nullo, poiché ogni elemento di A_{MIN} è minore o uguale ad ogni elemento di A_{MAX}.

Le relazioni di ricorrenza per il tempo $T(n)$ risultano pertanto:

$$T(n) = d, \qquad \text{per } n = 1,$$
$$T(n) = T(n/2) + c, \qquad \text{per } n > 1,$$

dove c e d sono due costanti, e quindi $T(n)$ è $O(\log n)$, usando $O(n)$ processori.

Per ordinare una sequenza qualsiasi, cioè non necessariamente bitonica, ci si riconduce al caso di sequenze bitoniche. Infatti, una sequenza qualsiasi A di lunghezza $n = 2^k$ può essere vista come $n/2$ sequenze bitoniche di lunghezza 2. Una volta ordinate in parallelo queste $n/2$ sequenze bitoniche, in modo che siano alternate una sequenza ordinata in modo crescente ed una ordinata in modo decrescente, si sfrutta il fatto che la concatenazione di una sequenza crescente con una decrescente forma una sequenza bitonica di lunghezza doppia. In questo modo, si considerano $n/4$ sequenze bitoniche lunghe 4, che vengono ordinate con l'algoritmo per le sequenze bitoniche,

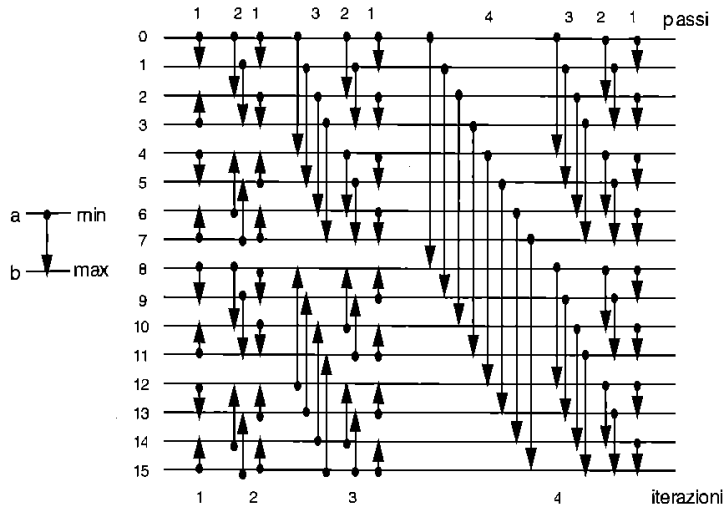


Figura 28.9: Diagramma di ordinamento dell'algoritmo "mergesort bitonico" per $n = 16$.

poi $n/8$ sequenze bitoniche lunghe 8, e così via, finché si giunge ad ordinare una singola sequenza bitonica lunga n.

Per valutare il tempo richiesto da questo algoritmo di ordinamento, detto "mergesort bitonico", su un modello PRAM, osserviamo che la costruzione di sequenze bitoniche lunghe 2^{i+1} richiede l'ordinamento di sequenze bitoniche lunghe 2^i, che costa $O(\log 2^i) = O(i)$ tempo. Pertanto, il tempo complessivo è $O(\sum_{i=1}^{k} i)$, che cresce come $O(\log^2 n)$. Poiché il numero massimo di confronti che vengono eseguiti contemporaneamente è $n/2$, il numero di processori richiesto è $O(n)$. Il lavoro dell'algoritmo è $O(n \log^2 n)$, che è nettamente migliore di quello della procedura TORNEO vista nel § 27, anche se purtroppo il tempo richiesto è un po' più alto.

Per meglio comprendere come realizzare l'algoritmo "mergesort bitonico" su un modello di calcolo parallelo con rete di interconnessione a grado limitato, è comodo visualizzarne lo schema di computazione tramite un "diagramma di ordinamento", come mostrato nella Fig. 28.9 per $n = 2^4 = 16$. In tale diagramma, le linee orizzontali rappresentano "canali" sui quali transitano da sinistra verso destra gli elementi da ordinare. Le frecce verticali rappresentano confronti tra coppie di elementi, che entrano da sinistra ed escono a destra, eventualmente scambiati, in modo che l'elemento maggiore esca sul canale indicato dalla freccia ed il minore su quello indicato dal pallino. Si osservi che l'algoritmo "mergesort bitonico" ha k iterazioni (4 nella Fig. 28.9) e che ogni iterazione i, per $1 \leq i \leq k$, richiede proprio i passi per ordinare $n/2^i$ sequenze bitoniche lunghe 2^i. Da un attento esame della Fig. 28.9, si può notare come al j-esimo passo, $i \geq j \geq 1$, siano confrontati e scambiati tra loro elementi le cui posizioni differiscono nel j-esimo bit meno significativo. Questo suggerisce immediatamente l'utilizzo di un ipercubo di dimensione k, poiché in un ipercubo sono connessi tra loro proprio i processori i cui indici differiscono per un bit!

La seguente procedura BITONIC-MERGESORT-IPERCUBO realizza direttamente l'algoritmo, come visualizzato nel diagramma della Fig. 28.9, dove l'indice j individua il "passo" e l'indice i individua l'"iterazione". Per comodità, l'indice i varia tra 0 e $k - 1$ anziché tra 1 e k.

```
procedure BITONIC-MERGESORT-IPERCUBO(a_0, a_1, ..., a_{n-1});
  begin
    for i := 0 to k - 1 do
      for j := i downto 0 do begin
        d := 2^j;
        for all h where 0 ≤ h ≤ n-1 do in parallel begin
          if h mod 2^d < d then begin
            t_h ← a_{h+d};
            if h mod 2^{i+2} < 2^{i+1} then begin
              b_h := max{a_h, t_h};
              a_h := min{a_h, t_h}
            end else begin
              b_h := min{a_h, t_h};
              a_h := max{a_h, t_h}
            end
          end;
          if h mod 2d ≥ d then
            a_h ← b_{h-d}
        end
      end
  end;
```

L'unica difficoltà della procedura è rappresentata dalla determinazione dei "versi" corretti dei confronti, cioè degli orientamenti delle frecce mostrate nella Fig. 28.9. Ciò è effettuato mediante la verifica " h **mod** $2^{i+2} < 2^{i+1}$", la cui dimostrazione di correttezza è lasciata per esercizio. Ovviamente, il tempo richiesto dalla procedura resta $O(\log^2 n)$ mentre il numero di processori rimane $O(n)$.

Esempio 28.12 (Mergesort bitonico su ipercubo). La Fig. 28.10 mostra un esempio di computazione dell'algoritmo "mergesort bitonico" su un ipercubo di dimensione $k = 3$, per ordinare la sequenza di $n = 8$ elementi 5, 6, 3, 13, 7, 10, 9, 2. □

La procedura BITONIC-MERGESORT-IPERCUBO può essere adattata facilmente anche al CCC, mantenendo inalterati il tempo di computazione ed il numero di processori. Inoltre, Thompson e Kung (1980) ne hanno fornita una versione che funziona su una mesh quadrata di dimensioni $\sqrt{n} \times \sqrt{n}$ e che richiede tempo ottimo $O(\sqrt{n})$. L'algoritmo "mergesort bitonico", comunque, non è il migliore algoritmo di ordinamento su un ipercubo, in quanto ne è noto un altro ottimo che richiede soltanto $O(\log n \log \log n)$ tempo.

28.4 Moltiplicazione di matrici

Riconsideriamo adesso il problema di moltiplicare due matrici $A = [a_{ij}]$ e $B = [b_{ij}]$ di dimensioni $n \times n$, dove la matrice prodotto $C = [c_{ij}] = AB$ è tale che $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, per $1 \leq i, j \leq n$ (cfr. § 16). Vediamo due algoritmi paralleli diversi, che usano una mesh ciclica ed un ipercubo.

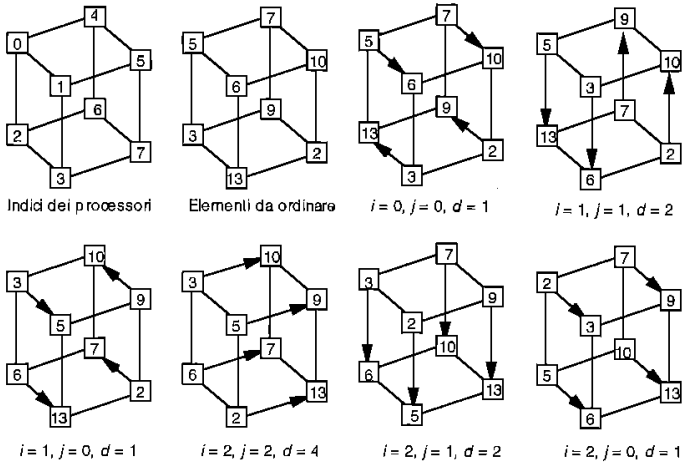


Figura 28.10: Esecuzione del "mergesort bitonico" su un ipercubo.

Si consideri una mesh ciclica di dimensioni $n \times n$ e si assuma che le due matrici A e B siano memorizzate nella mesh in modo che il processore $P_{i,j}$ contenga inizialmente gli elementi a_{ij} e b_{ij} ed al termine della computazione contenga c_{ij}. L'algoritmo consta di due fasi. Nella prima fase gli elementi della riga i-esima di A sono traslati ciclicamente a sinistra di $i - 1$ posizioni, $1 \leq i \leq n$, mentre gli elementi della colonna j-esima di B sono traslati ciclicamente verso l'alto di $j - 1$ posizioni, $1 \leq j \leq n$. La prima fase è puramente preparatoria, ed ha lo scopo di poter far incontrare nel processore $P_{i,j}$, durante la seconda fase, ciascuna coppia di elementi a_{ik} e b_{kj}, che contribuisce al calcolo di c_{ij}. È bene notare che le coppie a_{ik} e b_{kj} non si incontrano necessariamente nella sequenza data da $k = 1, 2, \ldots, n$, ma in accordo ad una sua traslazione ciclica. Infatti, nella seconda fase, che dura n iterazioni, sono eseguiti n^2 prodotti per ciascuna iterazione, ed ogni elemento di A e di B è coinvolto in un solo prodotto per iterazione. La risultante procedura è la seguente.

```
procedure MOLTMATRICI-MESH(A, B, n);
  begin
    for p := 1 to n − 1 do
      for all i, j where 1 ≤ i, j ≤ n do in parallel begin
        if i > p then a_{i,j} ← a_{i,j+1};
        if j > p then b_{i,j} ← b_{i+1,j}
      end;
    for all i, j where 1 ≤ i, j ≤ n do in parallel c_{i,j} := 0;
    for q := 1 to n do
      for all i, j where 1 ≤ i, j ≤ n do in parallel begin
        c_{i,j} := a_{i,j} * b_{i,j};
        a_{i,j} ← a_{i,j+1};
        b_{i,j} ← b_{i+1,j}
      end
  end;
```


La procedura MOLTMATRICI-MESH richiede $O(n)$ tempo ed n^2 processori per moltiplicare due matrici $n \times n$. Il lavoro è $O(n^3)$, che è lo stesso dell'usuale algoritmo sequenziale.

Esempio 28.13 (Moltiplicazione matrici su mesh ciclica). La Fig. 28.11 illustra un esempio di computazione del prodotto di due matrici 3×3 su una mesh ciclica di 9 processori. La Fig. 28.11a) mostra la memorizzazione iniziale delle matrici A e B. Le Figg. 28.11b)-c mostrano le traslazioni di A e B effettuate nella prima fase, dove le configurazioni delle matrici sono "fotografate" all'uscita del primo ciclo **for** sequenziale, per i valori di p uguali ad 1 e 2. Le Figg. 28.11d)-e mostrano invece le configurazioni delle matrici nella seconda fase, all'uscita del secondo ciclo **for** sequenziale, per i valori di q uguali ad 1 e 2 (mentre 3 non è mostrato). All'ingresso del secondo **for** sequenziale, sono effettuati i prodotti tra le coppie di elementi illustrati nelle Figg. 28.11c)-e. Si osservi l'ordine col quale sono calcolati gli elementi di C. Per esempio, il valore finale di $c_{3,3}$ è ottenuto sommandoci via via i prodotti $a_{3,2}b_{2,3}$ (Fig. 28.11c), $a_{3,3}b_{3,3}$ (Fig. 28.11d), e $a_{3,1}b_{1,3}$ (Fig. 28.11e). □

Si consideri adesso un ipercubo di n^3 processori, e si assuma che $n = 2^h$. In tal caso, gli indici dei processori dell'ipercubo hanno valori compresi tra 0 e $2^{3h} - 1$, e la loro rappresentazione binaria consta di $3h$ bit, che possono essere visti come formati da tre "blocchi" di h bit. Si può immaginare per comodità che ciascun blocco di h bit rappresenti un valore compreso tra 0 e $n - 1 = 2^h - 1$, con il blocco più a sinistra che rappresenta un indice k, quello centrale che rappresenta un indice i, e quello di destra che rappresenta j, così che un generico processore può essere indifferentemente individuato con un unico indice $m = 2^{2h}k + 2^h i + j$ oppure con una terna di indici (k, i, j).

Ciascun processore P_m ha cinque variabili locali A_m, B_m, C_m, D_m, ed E_m. Si assuma che le due matrici A e B abbiano indici di riga e colonna compresi tra 0 ed $n-1$, anziché 1 ed n, e che siano memorizzate nell'ipercubo in modo che il processore $P_m = P_{2^h i+j} = P_{0,i,j}$ contenga inizialmente gli elementi a_{ij} e b_{ij} nelle variabili A_m e B_m, rispettivamente, ed al termine della computazione contenga c_{ij} nella variabile C_m, per $0 \leq i, j \leq n - 1$.

L'algoritmo consta di tre fasi. Nella prima fase, i dati sono distribuiti in tempo $O(\log n)$ a tutti i processori dell'ipercubo per mezzo di tre cicli **for** effettuati, rispettivamente, su tutti i bit di ciascuno dei tre "blocchi". Al termine della prima fase, gli elementi a_{ij} e b_{ij} sono distribuiti a tutti gli A_m e B_m dei processori $P_m = P_{2^{2h}k+2^h i+j} = P_{k,i,j}$ tali che $0 \leq k \leq n - 1$. Similmente, a_{ik} è distribuito a tutti gli A_m dei processori $P_m = P_{2^{2h}k+2^h i+j} = P_{k,i,j}$ tali che $0 \leq j \leq n - 1$, mentre b_{kj} è distribuito a tutti i B_m dei $P_m = P_{2^{2h}k+2^h i+j} = P_{k,i,j}$ tali che $0 \leq i \leq n - 1$. In questo modo, ciascuno degli n^3 processori contiene una coppia distinta di elementi delle matrici A e B da moltiplicare, e per l'esattezza $P_m = P_{2^{2h}k+2^h i+j} = P_{k,i,j}$ contiene a_{ik} e b_{kj} . Nella seconda fase, sono effettuati in tempo $O(1)$ tutti questi prodotti. Infine, nella terza fase sono sommati i prodotti sui processori $P_{k,i,j}$ tali che $0 \leq k \leq n - 1$, ottenendo i risultati c_{ij} in tempo $O(\log n)$ nei processori $P_{0,i,j}$.

Nella seguente procedura MOLTMATRICI-IPERCUBO, sono utilizzate due funzioni BIT(m, p), che restituisce il $(p + 1)$-esimo bit meno significativo della rappresentazione binaria dell'intero m, per $p \geq 0$, e COMPLEMENTA(m, p), che restituisce l'intero ottenuto complementando il $(p + 1)$-esimo bit meno significativo della rappresentazione binaria di m.

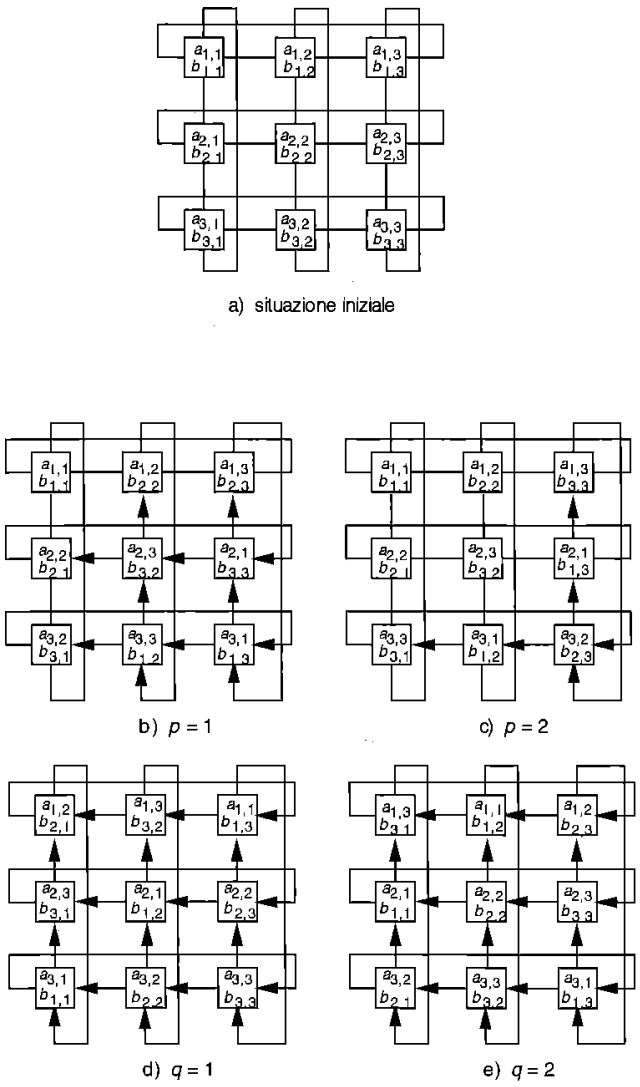


Figura 28.11: Moltiplicazione di matrici con una mesh ciclica.

Esempio 28.14 (BIT e COMPLEMENTA). Sia $m = 5$, che in binario è rappresentato con 101. Allora BIT(5, 0) = 1, BIT(5, 1) = 0, e BIT(5, 3) = 0. Invece, COMPLEMENTA(5,0) = 4, COMPLEMENTA(5, 1) = 7, e COMPLEMENTA(5, 3) = 21. □

```
procedure MOLTMATRICI-IPERCUBO(A, B, n);
  begin
    for p := 3h − 1 downto 2h do
      for all m where BIT(m, p) = 1 do in parallel
        begin
          D_m := COMPLEMENTA(m, p);
          A_m ← A_{D_m};
          B_m ← B_{D_m}
        end;
    for p := h − 1 downto 0 do
      for all m where BIT(m, p) ≠ BIT(m, 2h+p) do in parallel
        begin
          D_m := COMPLEMENTA(m, p);
          A_m ← A_{D_m}
        end;
    for p := 2h − 1 downto h do
      for all m where BIT(m, p) ≠ BIT(m, h + p) do in parallel
        begin
          D_m := COMPLEMENTA(m, p);
          B_m ← B_{D_m}
        end;
    for all m where 0 ≤ m ≤ 2^{3h} − 1 do in parallel
      C_m := A_m * B_m;
    for p := 2h to 3h − 1 do
      for all m where 0 ≤ m ≤ 2^{3h} − 1 do in parallel
        begin
          E_m := COMPLEMENTA(m, p);
          D_m ← D_{E_m};
          C_m := C_m + D_m;
        end;
  end;
```

Il tempo richiesto dalla procedura è $O(\log n)$, usando n^3 processori. Il lavoro è $O(n^3 \log n)$. Tale procedura può essere adattata al CCC o alla shuffle, mantenendo invariati tempo di computazione e numero di processori.

Esempio 28.15 (Moltiplicazione matrici su ipercubo). Nella Fig. 28.12 è illustrato un esempio di computazione del seguente prodotto di due matrici 2×2 su un ipercubo di $2^3 = 8$ processori:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -5 & -6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix}$$

□

Le procedure viste per la moltiplicazione di matrici possono essere usate, con lievi modifiche, per risolvere il problema dei CAMMINI MINIMI tra tutte le coppie di

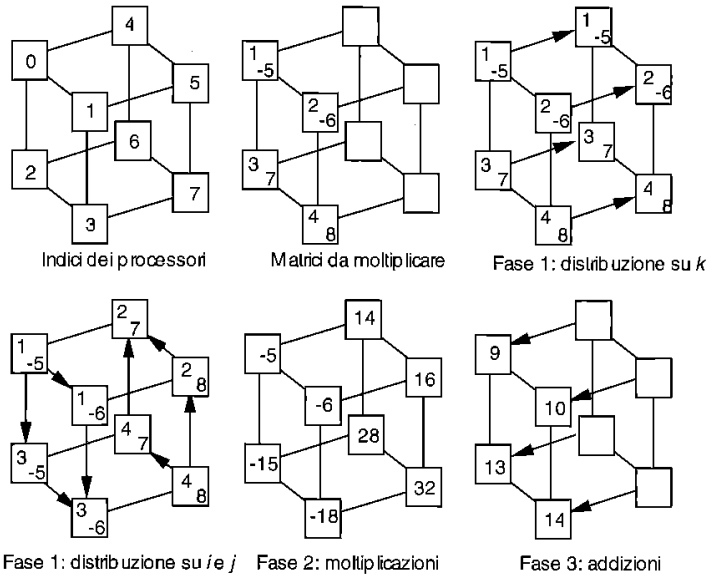


Figura 28.12: Computazione del prodotto di due matrici 2×2 su un ipercubo di 8 processori.

nodi di un grafo orientato e pesato con il metodo della "quadratura di matrici" discusso nel § 14. Utilizzando l'algoritmo per la mesh ciclica il tempo diviene $O(n \log n)$ usando n^2 processori, mentre con quello per l'ipercubo il tempo è $O(\log^2 n)$ usando n^3 processori. In entrambi i casi, il lavoro è solo di poco superiore a quello richiesto dal migliore algoritmo sequenziale per grafi densi, che richiede $O(n^3)$ tempo.

28.5 Trasformata di Fourier

Consideriamo adesso un algoritmo parallelo per calcolare la trasformata discreta di Fourier di un vettore reale $a = (a_0, a_1, \ldots, a_{n-1})$. Tale trasformata è data dal vettore complesso $b = (b_0, b_1, \ldots, b_{n-1})$ tale che

$$b_h = \sum_{k=0}^{n-1} a_k \left(\omega^h\right)^k, \qquad \text{per } 0 \le h \le n-1,$$

dove ω è la radice primitiva n-esima dell'unità. Si assuma n potenza di 2 e si riconsideri l'algoritmo "divide et impera" FFT introdotto nel § 11.

Riesaminiamo attentamente la Fig. 11.8, che mostra la ricombinazione dei risultati nell'algoritmo ricorsivo FFT durante il calcolo di $b_1 = a(\omega)$ per $n = 8$. Osserviamo che se si possono appaiare gli elementi $a_0, a_1, \ldots, a_6, a_7$ nelle quattro coppie $(a_0, a_4), \ldots, (a_3, a_7)$ che compaiono nel penultimo livello dell'albero, allora è possibile calcolare direttamente le quattro trasformate $a_0 - a_4, \ldots, a_3 - a_7$. Se poi si

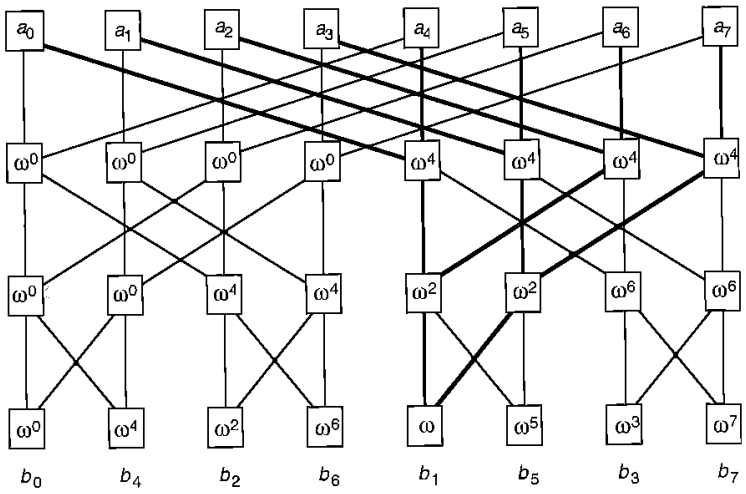


Figura 28.13: Computazione della trasformata discreta di Fourier con una butterfly (si ricordi che per $n = 8$ si hanno $\omega_4 = -\omega_0 = -1$, $\omega_5 = -\omega$, $\omega_6 = -\omega_2$, e $\omega_7 = -\omega_3$).

accoppiano opportunamente questi quattro valori, si possono calcolare le due trasformate al livello 1 dell'albero, ed infine da questi l'intera trasformata nella radice dell'albero. In altri termini, la computazione può procedere in modo iterativo "dal basso verso l'alto" nell'albero di Fig. 11.8, purché ci sia un opportuno schema di accoppiamento che permetta il calcolo di b_1 (e possibilmente di tutti i b_h per $0 \le h \le 7$). Ma tale schema di accoppiamento è esattamente quello di una butterfly!

Come illustrato nella Fig. 28.13, la computazione della trasformata di Fourier può procedere in una butterfly dall'alto verso il basso. I processori della riga 0 contengono inizialmente gli elementi del vettore da trasformare. Successivamente, tutti i processori della riga 1 ricevono in parallelo due dati x ed y dai processori della riga precedente, ed effettuano l'operazione $x + \omega^h y$, per una opportuna potenza di ω. Il procedimento si ripete successivamente per tutti i processori della riga 2, e così via, finché non si giunge all'ultima riga. Per esempio, nella Fig. 28.13 è evidenziato in grassetto l'albero relativo al calcolo di b_1 (si ricordi che per $n = 8$ risulta $\omega_4 = -\omega_0 = -1$).

L'unica difficoltà di questo algoritmo parallelo consiste nel determinare il corretto ω^h da utilizzare in ciascun processore per effettuare il calcolo di $x + \omega^h y$. Con questo algoritmo, la trasformata di Fourier può essere calcolata in tempo $O(\log n)$ usando $O(n \log n)$ processori, e quindi con $O(n \log^2 n)$ lavoro. Osservando però che soltanto i processori di una riga della butterfly sono attivi ad ogni iterazione e che le connessioni di una butterfly possono essere simulate da un ipercubo di n processori, si deduce che è possibile calcolare la trasformata di Fourier in tempo $O(\log n)$ usando un ipercubo di n processori, con un lavoro ottimo di $O(n \log n)$.

Nella seguente procedura FFT-IPERCUBO si utilizzano le funzioni BIT0(i, j) e BIT1(i, j), che restituiscono un intero m ottenuto ponendo a 0 oppure a 1, rispettivamente, il $(j+1)$-esimo bit meno significativo della rappresentazione binaria di i,

per $j \geq 0$. Si utilizza inoltre la funzione OMEGA(i, j, k) che restituisce ω^h, dove h è l'intero ottenuto invertendo l'ordine degli $i + 1$ bit più significativi dell'intero j (di k bit) e ponendo a 0 i rimanenti $k - i - 1$ bit. La funzione OMEGA richiede tempo $O(1)$ sia per calcolare h che per restituire il valore ω^h, qualora le potenze di ω siano state calcolate preventivamente una volta per tutte e memorizzate in una matrice accessibile direttamente tramite gli indici h ed n.

Esempio 28.16 (Funzione OMEGA). OMEGA(1, 9, 4) $= \omega^4$, poiché la rappresentazione binaria di 9 su 4 bit è $\underline{10}01$, e invertendo l'ordine dei primi 2 bit e ponendo a 0 gli altri si ottiene $\underline{01}00$, che rappresenta 4. □

```
procedure FFT-IPERCUBO(a_0, a_1, ..., a_{n-1});
  begin
    k := log n;
    for all j where 0 ≤ j ≤ n − 1 do in parallel b_j := a_j;
    for i := 0 to k − 1 do
      for all j where 0 ≤ j ≤ n − 1 do in parallel begin
        x_j := b_j;
        d_j := COMPLEMENTA(j, k − 1 − i);
        if d_j = BIT0(j, k − 1 − i)
          then x_j ← b_{d_j}
          else x_j := b_j;
        if d_j = BIT1(j, k − 1 − i)
          then y_j ← b_{d_j}
          else y_j := b_j;
        b_j := x_j + OMEGA(i, j, k) * y_j
      end
  end;
```

28.6 Esercizi

Esercizio 28.1 (Trasposta su mesh). Si scriva una procedura di complessità $O(n)$ per trasporre una matrice $A = [a_{ij}]$, $1 \leq i, j \leq n$, con una mesh quadrata di n^2 processori. L'elemento a_{ij} è inizialmente memorizzato nel processore P_{ij} ed al termine della computazione deve trovarsi nel processore P_{ji}. Ogni processore può utilizzare un numero costante di celle di memoria.

Esercizio 28.2 (Trasposta su shuffle). Si scriva una procedura efficiente per trasporre una matrice $A = [a_{ij}]$, $1 \leq i, j \leq n$, con una shuffle di n^2 processori $P_0, P_1, \ldots, P_{n^2-1}$. Si assumano $n = 2^q$ e la matrice memorizzata per righe: l'elemento a_{ij} è inizialmente memorizzato nel processore P_k tale che $k = 2^q(i - 1) + (j - 1)$.

Si può notare che l'indice k di un processore della shuffle è rappresentato in binario con $2q$ bit, dove i q bit più significativi rappresentano l'indice $i - 1$, mentre i q bit meno significativi rappresentano $j - 1$. Per trasporre la matrice $A = [a_{ij}]$, il processore P_k contenente a_{ij}, con $k = 2^q(i - 1) + (j - 1)$, e il processore P'_k contenente a_{ji}, con $k' = 2^q(j - 1) + (i - 1)$, devono perciò scambiarsi i dati. La rappresentazione binaria di k si ottiene con q traslazioni cicliche sinistre rispetto a quella di k' (e viceversa). Quindi, per trasporre la matrice è sufficiente effettuare q operazioni MESCOLA(b_k), ognuna delle quali permette di ottenere uno spostamento degli elementi in accordo alle connessioni di mescolamento. Il tempo richiesto è

$O(\log n)$. Si noti che alcuni processori effettuano delle operazioni MESCOLA in più di quelle necessarie.

```
procedure TRASPOSTA-SHUFFLE (A = [a_ij]);
  begin
    for all k where 0 ≤ k ≤ n^2 − 1 do in parallel
      b_k := a_k;
    for h := 1 to q do
      for all k where 0 ≤ k ≤ n^2 − 1 do in parallel
        MESCOLA(b_k)
  end;
```

Esercizio 28.3 (Distanze minime su ipercubo). Dato un grafo orientato connesso, la distanza tra due nodi u e v è uguale al numero di archi in un cammino tra u e v che contiene il numero minimo di archi. Si fornisca una procedura per trovare le distanze minime tra tutte le coppie di nodi del grafo in tempo polilogaritmico, usando un ipercubo.

Esercizio 28.4 (Massimo su butterfly). Sia dato un insieme di $O(n \log^2 n)$ interi, memorizzato in una butterfly di rango $\log n$. Si scriva una procedura per trovare il massimo in tempo $O(\log n)$. Al termine della computazione, si vuole ottenere il risultato in tutti i processori della butterfly.

Esercizio 28.5 (Chiusura transitiva su mesh ciclica). Dato un grafo orientato di n nodi, se ne vuole trovare la chiusura transitiva. Si scriva una procedura di complessità $O(n \log n)$ per una mesh ciclica bidimensionale di $n \times n$ processori.

Esercizio 28.6 (Ricerca su shuffle). Sia dato un insieme di $n \log n$ elementi interi, memorizzato in una shuffle di n processori. Si scriva una procedura che richieda $O(\log n)$ tempo per decidere se un elemento x appartiene o no all'insieme, assumendo che soltanto un processore (p.e. P_0) possa comunicare con l'esterno per leggere x e restituire il risultato.

Esercizio 28.7 (Massimo su ipercubo). Dati $n \log n$ valori, con $n = 2^k$, se ne vuole trovare il massimo. Si scriva una procedura che richieda $O(\log n)$ tempo per un ipercubo avente n processori, in modo che ciascun processore contenga prima della computazione $\log n$ valori e al termine della computazione contenga il massimo.

Esercizio 28.8 (Somme prefisse su mesh). Si scriva una procedura di complessità $O(n)$ per calcolare le somme prefisse di una sequenza di n^2 interi memorizzata (per righe) in una mesh di $n \times n$ processori.

L'algoritmo è simile a quello per la sommatoria, ove però si effettua dapprima una somma sugli elementi di ciascuna riga, da sinistra verso destra, ed un'ulteriore somma sull'ultima colonna, dall'alto verso il basso, con l'aggiunta di alcuni passi supplementari. Dopo aver fatto la sommatoria, memorizzando i risultati parziali nelle variabili $b_{i,j}$, i processori della prima riga e dell'ultima colonna contengono già i risultati desiderati. Le rimanenti somme prefisse sono ottenute per ciascuna riga da destra verso sinistra. La somma prefissa in un processore è ottenuta sottraendo la somma prefissa ed il dato di partenza del processore di destra.

```
procedure SOMME-PREFISSE-MESH(a_{1,1}, a_{1,2}, ..., a_{n,n});
  begin
    for all i, j where 1 ≤ i ≤ n, j = 1 do in parallel
      b_{i,j} := a_{i,j};
    for j := 2 to n do
      for all i, j where 1 ≤ i ≤ n do in parallel
        begin d_{i,j} ← b_{i,j-1}; b_{i,j} := d_{i,j} + a_{i,j} end;
    for i := 2 to n do
      for all i, j where j = n do in parallel
        begin d_{i,j} ← b_{i-1,j}; b_{i,j} := d_{i,j} + b_{i,j} end;
    for j := n - 1 downto 1 do
      for all i, j where 2 ≤ i ≤ n do in parallel
        begin d_{i,j} ← b_{i,j+1}; e_{i,j} ← a_{i,j+1}; b_{i,j} := d_{i,j} - e_{i,j} end
  end;
```

Capitolo 29

Algoritmi VLSI

Negli ultimi anni, l'enorme sviluppo della tecnologia microelettronica ha reso possibile la produzione in grosse quantità di circuiti realizzati con materiale semiconduttore (cristalli di silicio) che consentono l'integrazione di centinaia di migliaia di transistori su di una piastrina (o chip) di superficie inferiore al centimetro quadrato. Questa tecnologia di miniaturizzazione è chiamata VLSI (per Very Large Scale Integration, ovvero "integrazione su grandissima scala") e consente di realizzare circuiti molto grandi e complessi su piastrine estremamente piccole, apportando notevoli vantaggi, tra i quali una maggior velocità, una minore dissipazione di energia, e una riduzione dei costi di produzione. La tecnologia VLSI è stata ampiamente utilizzata per produrre non solo componenti dei calcolatori elettronici, come moduli di memoria o microprocessori, ma anche circuiti specializzati nella risoluzione di un singolo problema computazionale. In particolare, per quel che riguarda proprio gli aspetti computazionali, col VLSI va scomparendo la classica distinzione tra il "software", cioè la descrizione di algoritmi mediante procedure scritte in un linguaggio di programmazione, ed l'"hardware", cioè la circuiteria di un calcolatore elettronico che esegue le procedure. Infatti, col VLSI gli algoritmi sono sia descritti che realizzati direttamente con circuiti hardware!

Nel progettare algoritmi VLSI, cambiano un po' le regole del gioco, rispetto a quanto abbiamo considerato finora. Intanto, gli algoritmi operano su rappresentazioni binarie (e non decimali) dei dati, poiché i collegamenti del circuito possono trasmettere solo segnali binari. Inoltre, dalla regolarità "logica" di un algoritmo si passa a considerarne la regolarità "geometrica" e di conseguenza variano anche le misure per analizzare la complessità. In un circuito VLSI, il tempo T richiesto da una computazione tende a essere dominato più dalla necessità di distribuire i dati all'interno del circuito che dall'esecuzione di istruzioni aritmetico/logiche. Dovendo inoltre realizzare il circuito su una piastrina di silicio, è fondamentale considerare anche l'area A richiesta dal circuito, poiché il costo di fabbricazione di una piastrina cresce esponenzialmente con l'area della piastrina stessa. Comunque, anche l'area di un circuito tende ad essere dominata più dallo spazio richiesto dai collegamenti che dalle unità di calcolo vere e proprie. Naturalmente, area e tempo richiesti da un circuito non sono tra loro scorrelati, e la diminuzione di uno comporta spesso l'aumento dell'altro. Una misura che "lega" insieme l'area A ed il tempo T è data dal prodotto AT^2, in quanto risulta proporzionale all'"energia dissipata" dal circuito durante la computazione. È chiara a questo punto la necessità di strumenti matematici per analizzare la complessità AT^2 e, in particolare, per ricavarne limitazioni inferiori.

Quando si progetta un algoritmo VLSI, se ne definisce un grafo, detto "grafo della computazione", dove i nodi corrispondono ad unità di elaborazione e gli archi a connessioni. In questo grafo, i dati entrano da certi nodi (le "porte di ingresso"), transitano sugli archi, attraversando le unità di elaborazione, finché giungono a certi altri nodi (le "porte di uscita") che trasmettono il risultato all'esterno. Un aspetto fondamentale consiste nel trovare un "layout" di area minima per il grafo della computazione, ovvero nel disporre il grafo su una griglia nel piano Cartesiano, rispettando opportune regole imposte dalla tecnologia VLSI. Tipici grafi di computazione sono gli alberi binari, le mesh, e in genere tutti i grafi a grado limitato (da una costante) introdotti nel § 28.

Nel seguito di questo capitolo, è dapprima descritto il cosiddetto "modello a griglia", introdotto da Thompson (1980). Pur astraendo dalla miriade di dettagli fisici dettati dalla tecnologia microelettronica, le ipotesi assunte nel modello a griglia sono aderenti alla realtà e permettono quindi di studiare proprietà computazionali dei circuiti VLSI in modo attendibile. Dopo aver considerato tecniche per ricavare limitazioni inferiori della complessità, sono trattati in dettaglio problemi di "layout" di grafi nel modello a griglia. Infine, sono presentati algoritmi VLSI efficienti per l'ORDINAMENTO e la MOLTIPLICAZIONE DI MATRICI.

29.1 Il modello a griglia

Come per ogni modello matematico, consideriamo un modello astratto VLSI che sia generale, cioè indipendente da una particolare tecnologia, fedele alla realtà fisica, nel senso che la potenza computazionale del modello astratto sia equivalente a quella della sua controparte fisica, e semplice, cioè facile da analizzare matematicamente al fine di determinarne le misure di complessità. Consideriamo il modello a griglia introdotto da Thompson, ritenuto pressoché unanimamente come il miglior modello astratto VLSI. Per le motivazioni "reali" che rendono tale modello aderente alla tecnologia microelettronica si possono consultare i testi di Mead e Conway (1980) e di Ullman (1983).

Nel modello di Thompson, un circuito VLSI è formato da una griglia rettangolare, lungo la quale scorrono dei fili. Senza perdere in generalità, si assume che la griglia abbia solo due livelli: su un livello scorrono i fili verticali e sull'altro quelli orizzontali. Due fili distinti possono scorrere su due linee di griglia che si incrociano, ma non possono sovrapporsi sulla stessa linea di griglia (si veda la Fig. 29.1). Lo spazio tra due linee di griglia è considerato unitario e ciascun filo trasporta un segnale binario. Talvolta è utile considerare un fascio di fili, che trasportano un numero costante di segnali binari, come un unico filo di ampiezza costante. I processori sono unità di elaborazione specializzate per calcolare semplici operazioni aritmetico/logiche (come addizioni, confronti, ecc.), e si trovano agli incroci delle linee di griglia. I fili che passano per l'incrocio dove si trova un'unità di eleborazione sono ingressi e/o uscite dell'unità di elaborazione. Ogni unità di elaborazione ha un numero costante di ingressi e uscite. Si assume che le unità di elaborazione richiedano area costante, sotto l'ipotesi che effettuino una funzione su un numero costante di bit. Se una unità di elaborazione richiede al più 4 collegamenti, allora occupa un singolo incrocio, altrimenti più incroci adiacenti, ma il numero di collegamenti è comunque limitato da una costante (si veda ancora la Fig. 29.1). Sotto queste ipotesi, due unità di elaborazione con lo stesso numero di ingressi e uscite hanno la stessa area, ma tale ipotesi non è restrittiva poiché si considerano unità con un numero costante di ingressi e uscite e si analizza la complessità in ordine di grandezza.

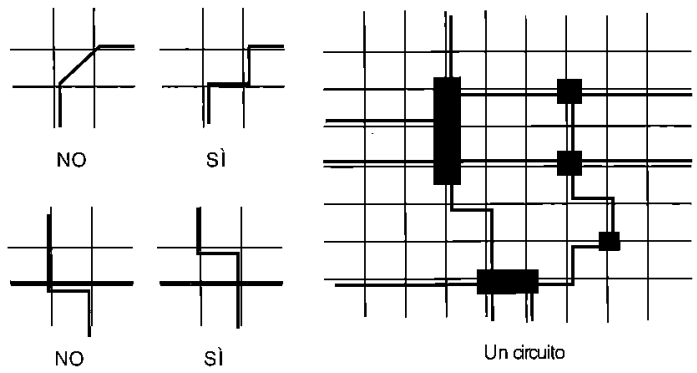


Figura 29.1: Il modello a griglia di Thompson.

Il circuito ha sempre forma rettangolare. Infatti, se così non fosse, si potrebbe sempre racchiudere il circuito in un rettangolo aumentandone l'area di, al più, una costante moltiplicativa. L'area A del circuito è data dal prodotto tra il numero di linee di griglia verticali (base) ed il numero di linee di griglia orizzontali (altezza). Il circuito è sincrono ed il tempo T di computazione, misurato in unità, è uguale al numero di unità di tempo che intercorrono tra l'invio del primo segnale di ingresso e la produzione dell'ultimo segnale di uscita. I segnali di ingresso si presentano ad istanti prefissati su fili prefissati che entrano dal bordo del circuito e quelli di uscita sono prodotti ad istanti prefissati su altri fili prefissati che escono anch'essi dal bordo del circuito. Ogni dato di ingresso è letto una e una sola volta, ed ogni dato di ingresso o uscita è disponibile in una sola locazione. Il tempo necessario per trasmettere un segnale tra i due estremi di un filo è $O(1)$, indipendentemente dalla lunghezza del filo. Il numero di unità di elaborazione che un segnale può attraversare in una unità di tempo è 1. Da ciò segue che il numero di unità di tempo necessarie ad un segnale per raggiungere n unità di elaborazione è $\Omega(\log n)$ (tale limitazione è raggiungibile con un circuito il cui grafo di computazione è un albero binario con n foglie).

29.2 Il solido spazio/tempo

Per analizzare la complessità di circuiti e, in particolare, per ricavare limitazioni inferiori di area e tempo, è comodo rifarsi al cosiddetto "solido spazio/tempo" illustrato nella Fig. 29.2, che rappresenta l'evolversi nel tempo della computazione nel circuito.

Intersecando il solido con piani orizzontali o verticali, oppure considerandone il volume, se ne ricavano diverse proprietà che comportano tipi diversi di limitazioni inferiori, le quali possono coinvolgere la "memoria", l'"ingresso/uscita", ed il "flusso di informazione" del circuito.

1. Intersecando il solido con un piano orizzontale (con valore della terza coordinata pari a t), si ottiene un rettangolo corrispondente allo stato del circuito al tempo

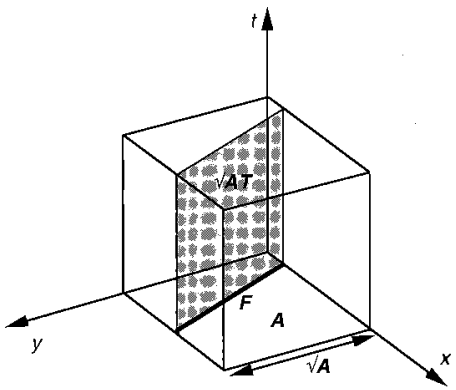


Figura 29.2: Il solido spazio/tempo.

t. Poiché l'area di tale rettangolo non può essere inferiore al numero di bit che il circuito deve memorizzare al tempo t, si deduce che l'area A del circuito non può essere inferiore al massimo numero di bit che il circuito deve ricordare allo stesso istante di tempo, dove tale massimo è calcolato per $1 \leq t \leq T$. In altri termini, se si dimostra che il circuito richiede almeno $f(n)$ celle di memoria, allora risulta $A \geq f(n)$.

2. Ad ogni istante t di tempo, il circuito può leggere un numero di bit al più uguale alla propria area. Pertanto, il volume V del solido non può essere inferiore al numero di bit che il circuito deve ricevere complessivamente in input. Poiché $V = AT$, segue che se il circuito deve leggere n bit di input, allora $AT \geq n$. Considerazioni analoghe valgono per i bit di output prodotti come risultato.
3. Intersecando il solido con un piano verticale, si ottiene un rettangolo di area $\sqrt{A}T$ che partiziona il circuito in due parti, divise da una "frontiera" F. Durante la computazione, le due parti del circuito dovranno scambiarsi dati attraverso F per calcolare il risultato del problema. Pertanto, la "quantità di informazione" che passa complessivamente attraverso la frontiera F non può essere maggiore di $\sqrt{A}T$. Se si dimostra che la quantita d'informazione deve valere almeno I, allora risulta $AT^2 \geq I^2$.

Le limitazioni inferiori per AT sono in genere le più facili da dimostrare. Per esempio, per ordinare n numeri, di k bit ciascuno, occorre leggere nk bit e quindi AT è $\Omega(nk)$, mentre per moltiplicare tra loro due matrici booleane $n \times n$ si ha che AT è $\Omega(n^2)$. Naturalmente, l'area ed il tempo non possono essere tra loro scorrelati. Per esempio, se gli elementi delle matrici da moltiplicare sono forniti tutti insieme, allora T è $\Omega(1)$ ed A è $\Omega(n^2)$, mentre se sono forniti riga per riga allora sia T che A sono $\Omega(n)$. Inoltre, se almeno un bit di uscita dipende da n bit di ingresso, come nel caso delle moltiplicazioni di matrici, allora T è $\Omega(\log n)$. Infine, AT non può essere ovviamente più piccola di una limitazione inferiore del numero di operazioni richieste da un qualsiasi algoritmo sequenziale (come per la limitazione sequenziale

$\Omega(n^2)$, sempre per la moltiplicazione di matrici). Per dimostrare limitazioni inferiori dell'area A, una tecnica diffusa consiste nel determinare un sottoinsieme X' "grande" dell'insieme X dei bit di input, dal quale dipende un sottoinsieme Y' "grande" dell'insieme Y dei bit di output, ovvero tale che, "fissando" i valori di tutti i bit in $X - X'$, variando il valore dei bit di X' varia anche il valore di quelli di Y'. In altri termini, ciò implica che esiste un istante t che separa la lettura di X' dalla produzione di Y'. Successivamente, si dimostra che, "fissando" i valori dei bit di $X - X'$ ad una particolare configurazione, esistono $2^{f(n)}$ configurazioni binarie distinte dei bit in X' che richiedono $2^{f(n)}$ configurazioni binarie distinte dei bit in Y'. In questo modo, il circuito deve avere la possibilità di memorizzare $2^{f(n)}$ stati distinti al tempo t, poiché altrimenti a configurazioni distinte degli input corrisponderebbero gli stessi output. Pertanto, la memoria del circuito deve essere $\Omega(\log 2^{f(n)})$ e quindi A è $\Omega(f(n))$. Vediamo un esempio di applicazione di questa tecnica per il problema dell'ORDINAMENTO.

Esempio 29.1 (Limitazione inferiore di A per ORDINAMENTO). Si consideri il problema di ordinare n numeri di k bit, con $k > \log n$. Siano $X_1, X_2, \ldots, X_n$, dove $X_i = x_{ik-1}x_{ik-2}\cdots x_{i0}$, i numeri in ingresso da ordinare, e si denotino con $Y_1, Y_2, \ldots, Y_n$ i risultati, dove $Y_j = y_{jk-1}y_{jk-2}\cdots y_{j0}$ è il j-esimo valore nella sequenza ordinata. Siano $X' = \{x_{10}, \ldots, x_{n0}\}$ ed $Y' = \{y_{10}, \ldots, y_{n0}\}$. È immediato osservare che per ogni i e j, y_{j0} dipende da x_{i0}. Infatti, si fissino tutti i bit in $X - \{x_{i0}\}$ in modo che $x_{h1} = x_{h2} = \cdots = x_{hk-} = 0$, per $1 \leq h \leq n$, ed inoltre si fissino $j - 1$ bit di $X' - \{x_{i0}\}$ a zero ed $n - j$ bit di $X' - \{x_{i0}\}$ ad 1. Ma allora si ha che $Y_j = X_i$ e, in particolare, $y_{j0} = x_{i0}$. Pertanto, tutti i bit meno significativi degli input devono essere letti prima di produrre i bit meno significativi degli output. Si fissino adesso i valori dei bit in $X - X'$ in modo che i bit $x_{i\lceil \log n \rceil} \cdots x_{i1}$ rappresentino in binario il numero $i - 1$, mentre $x_{ik-1} = x_{ik-2} = \cdots = x_{i\lceil \log n \rceil + 1} = 0$. In questa maniera, risulta che $Y_j = X_j$, prescindendo dai valori dei bit in X' e, in particolare, $y_{j0} = x_{j0}$. Pertanto, esistono 2^n configurazioni distinte dei bit in X' a cui corrispondono 2^n configurazioni distinte dei bit in Y'. Il circuito deve quindi poter memorizzare 2^n stati distinti ed A risulta essere $\Omega(\log 2^n) = \Omega(n)$. □

Infine, per dimostrare limitazioni inferiori di AT^2, la tecnica più usata consiste nel valutare la "quantità di informazione" I che complessivamente deve attraversare la frontiera F di una partizione del circuito. Se il circuito è alto h e largo w, con $h \leq w$, allora ad ogni istante di tempo al più h bit possono attraversare F, e quindi $hT \geq I$. Essendo $A = wh$ e $w \geq h$, segue che $AT^2 \geq I^2$. Con questa tecnica, si può dimostrare che AT^2 risulta essere $\Omega(n^4)$ per la MOLTIPLICAZIONE DI MATRICI e $\Omega(n^2)$ per l'ORDINAMENTO. Vediamo come esempio la dimostrazione di quest'ultimo risultato.

Esempio 29.2 (Limitazione inferiore AT^2 per ORDINAMENTO). Si consideri nuovamente il problema di ordinare n numeri di k bit, con $k > \log n$, introdotto nell'Esempio 29.1, e siano X' ed Y' definiti nello stesso modo. In un circuito arbitrario che risolve il problema, possono presentarsi due casi. Se esiste una porta di uscita dalla quale sono prodotti almeno $n/3$ bit di Y', allora si ha che $T \geq n/3$ e quindi banalmente AT^2 è $\Omega(n^2)$. Se invece ogni porta di uscita produce meno di $n/3$ bit di Y', allora è facile provare che esiste una linea spezzata verticale con al più una deviazione (cioè dalla forma ┌┘) che partiziona il circuito C in due parti C_1 e C_2 tali che il numero dei bit di Y' prodotti in C_1 e il numero dei bit di Y' prodotti in C_2 è per entrambi

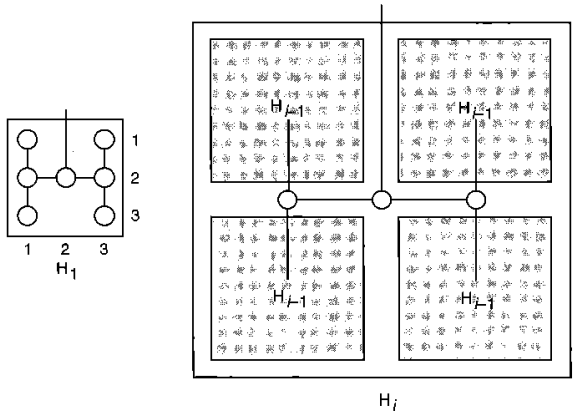


Figura 29.3: Layout ad "H" di area $O(n)$ per un albero binario completo con $n = 2^{2i}$ foglie.

compreso tra $n/3$ e $2n/3$. Infatti, se per assurdo così non fosse, allora cominciando ad isolare il primo incrocio (punto) della griglia in alto a sinistra e procedendo dall'alto in basso e da sinistra verso destra incorporando tutti gli altri incroci, si riuscirebbe ad individuarne uno dal quale sono prodotti più di $n/3$ bit di Y', che è una contraddizione. Almeno uno tra C_1 e C_2 deve leggere almeno $n/3$ bit di X' (senza perdere in generalità, si assuma che sia C_1). A questo punto fissiamo $n/3$ dei bit di X' letti da C_1 ed $n/3$ dei bit di Y' prodotti da C_2. Siano tali bit $x_{s_1 0}, \ldots, x_{s_{n/3} 0}$ e $y_{j_1 0}, \ldots, y_{j_{n/3} 0}$, rispettivamente. È possibile fissare gli altri bit in $X - X'$ in modo che, per $1 \leq i \leq n/3$, sia $x_{s_i 0} = y_{j_i 0}$. Basta infatti porre i primi $\lceil \log n \rceil$ bit di X_{s_i} alla rappresentazione binaria di j_i e gli altri X_p ai valori in $\{1, \ldots, n\} - \{j_1, \ldots, j_{n/3}\}$. A questo punto, tutti gli $n/3$ bit $\{x_{s_i 0}\}$ vanno trasmessi da C_1 a C_2, ma poiché se ne possono inviare al più h per volta (dove h è l'altezza del circuito) si ha $hT \geq n/3$, dalla quale deriva che AT^2 è $\Omega(n^2)$. □

29.3 Layout

Le limitazioni inferiori viste precedentemente si riferivano a circuiti arbitrari per la risoluzione di un problema. Supponiamo adesso di avere già individuato un grafo di computazione per il circuito che risolve il problema e di voler trovare un "layout" minimo di tale grafo, cioè disegnarlo nel modello a griglia in modo da minimizzarne l'area A. In particolare, consideriamo layout ottimi per alberi binari (e loro varianti), butterfly, e shuffle, ed accenniamo ad uno schema generale "divide et impera" per il layout di un grafo qualsiasi con grado massimo limitato da una costante.

Il miglior layout per un albero binario completo con n foglie è il cosiddetto layout quadrato ad "H", che può essere definito ricorsivamente come mostrato nella Fig. 29.3. Sia il numero delle foglie una potenza pari di due, ovvero $n = 2^{2i}$. Per $i = 1$, il layout H_1 per un albero binario con $n = 2^2 = 4$ foglie richiede 3 linee orizzontali e 3 linee verticali di griglia. Per $i > 1$, invece, il layout H_i per un albero

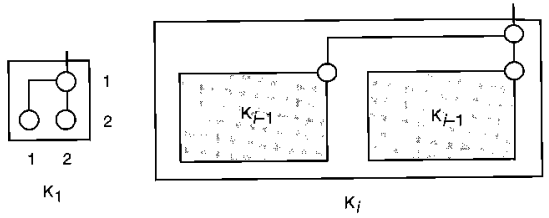


Figura 29.4: Layout K_j di area $O(n \log n)$ per un albero binario completo con tutte le $n = 2^j$ foglie sul bordo inferiore.

binario con $n = 2^{2i}$ foglie può essere costruito a partire da quattro layout H_{i-1} per alberi con $n/4 = 2^{2(i-1)}$ foglie, utilizzando in più una linea orizzontale di griglia per disporre la radice ed i suoi figli, ed una linea di griglia verticale per collegare la radice con l'esterno. Detto $L(i)$ il lato di H_i, si hanno le seguenti relazioni di ricorrenza:

$$\begin{aligned} L(i) &= 3 && \text{se } i = 1, \\ L(i) &= 2L(i-1) + 1 && \text{se } i > 1. \end{aligned}$$

Risolvendo, si ottiene che $L(i)$ è $O(2^i) = O(\sqrt{n})$. Pertanto l'area A del layout è $O(n)$, che ovviamente è ottima. Considerazioni analoghe valgono anche nel caso che il numero delle foglie sia una potenza dispari di due, cioè $n = 2^{2i-1}$.

Il layout ad H, pur richiedendo area ottima, presenta la caratteristica di non avere tutte le foglie sui bordi del circuito. Nei circuiti ad albero, le foglie vengono talvolta usate per l'ingresso/uscita di n dati in parallelo. Poiché per motivi tecnologici è conveniente che i nodi usati per l'ingresso/uscita stiano sui bordi del circuito, si ricorre in tal caso al layout "banale" mostrato nella Fig. 29.4 che, avendo larghezza n ed altezza $\log n$, richiede $O(n \log n)$ area. Tale layout è comunque ottimo tra quelli che presentano tutte le foglie sui bordi, poiché è possibile dimostrare che qualsiasi layout per alberi binari che abbia tutte le foglie sui bordi richiede $\Omega(n \log n)$ area.

Il layout ad H presenta il vantaggio di richiedere area ottima anche nel caso in cui all'albero binario siano aggiunti ulteriori archi. Gli alberi binari presentano la caratteristica che tra ogni coppia di nodi c'è esattamente una catena semplice. In altri termini, cancellando un nodo o un arco l'albero viene disconnesso in due componenti, mentre questo non avviene per altri grafi, come per esempio una mesh. Poiché nei circuiti VLSI non è infrequente la rottura di qualche unità di elaborazione, specialmente durande la fabbricazione stessa del circuito, per non gettare via il circuito vengono introdotti nodi ed archi aggiuntivi che permettano di deviare intorno ad un nodo guasto mantenendo il grafo connesso. Nel caso degli alberi binari, la radice ed i suoi figli vengono di solito collegati tutti e tre con l'esterno, e vengono aggiunti archi, per esempio, tra due "cugini" che sono entrambi figli destri (o sinistri), come proposto da Bertossi e Bonuccelli (1989). La Fig. 29.5 mostra il layout ad H in tale caso, dove i collegamenti tra cugini sono inseriti senza aumentare in ordine di grandezza l'area, che rimane $O(n)$.

Vediamo adesso un layout per una butterfly di rango k (cfr. § 27). Nel disegnare il layout, si adotta la convenzione sulla disposizione dei collegamenti "diagonali" mostrata nella Fig. 29.6, dove è illustrato anche il layout nel caso base $k = 1$.

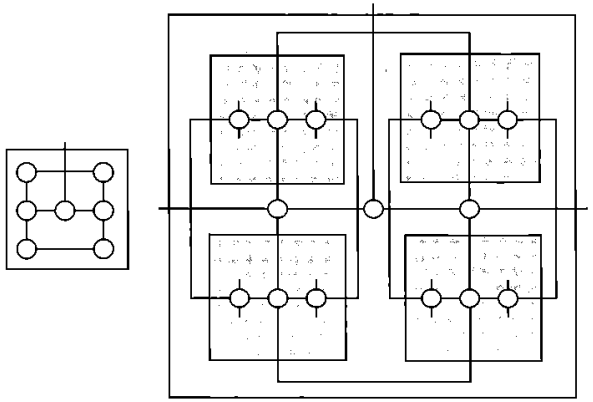


Figura 29.5: Layout ad "H" di area ottima $O(n)$ per un albero binario completo di $n = 2^{2i}$ foglie con collegamenti aggiuntivi tra nodi "cugini".

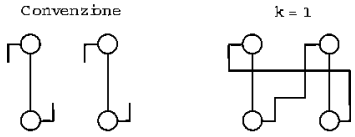


Figura 29.6: Convenzione sulla disposizione dei collegamenti e layout di una butterfly di rango $k = 1$.

In generale, il layout di una butterfly di rango $k > 1$ è costruito a partire dai layout di due butterfly di rango $k - 1$, inserendo gli opportuni collegamenti tra i nodi delle prime due righe, come mostrato nella Fig. 29.7 per $k = 3$ (dove i collegamenti "verticali" sono tratteggiati e quelli "diagonali" sono continui). Per calcolare l'area A del layout, si noti che il numero delle linee di griglia verticali è $3n$, mentre quello delle linee orizzontali è $k + 1 + \sum_{i=1}^{k} 2^i$, che cresce come $O(n)$. Pertanto, l'area A è $O(n^2)$.

Mostriamo adesso il layout delle connessioni shuffle (cfr. § 28) tra due file di n nodi, per $n = 2^i$ con $i = 1, 3$ (la generalizzazione per i arbitrario essendo immediata). Per $i = 1$, si collegano coppie di nodi utilizzando 4 tracce (cioè linee di griglia orizzontali intermedie). Il procedimento si ripete all'aumentare di i raddoppiando il numero di nodi con il conseguente raddoppio di linee orizzontali, come mostrato nella Fig. 29.8, dalla quale è immediato osservare come l'area A sia $O(n^2)$. Richiamiamo l'attenzione sul fatto che si può disegnare un layout con meno di $2n$ linee di griglia orizzontali intermedie, abbassando così la costante moltiplicativa, ma l'area A rimane comunque dello stesso ordine di grandezza.

Concludiamo accennando ad un algoritmo generale di tipo "divide et impera" per trovare il layout di un grafo G di n nodi con grado massimo dei nodi limitato da 4. Si inizia disconnettendo il grafo G in due grafi G_1 e G_2, tramite l'eliminazione di alcuni

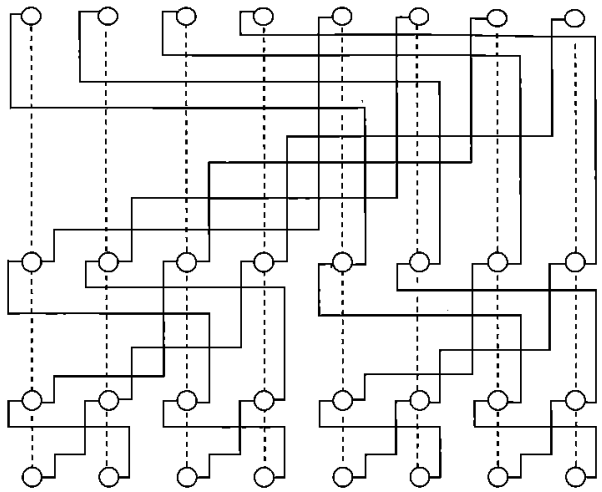


Figura 29.7: Layout di una butterfly di rango $k = 3$.

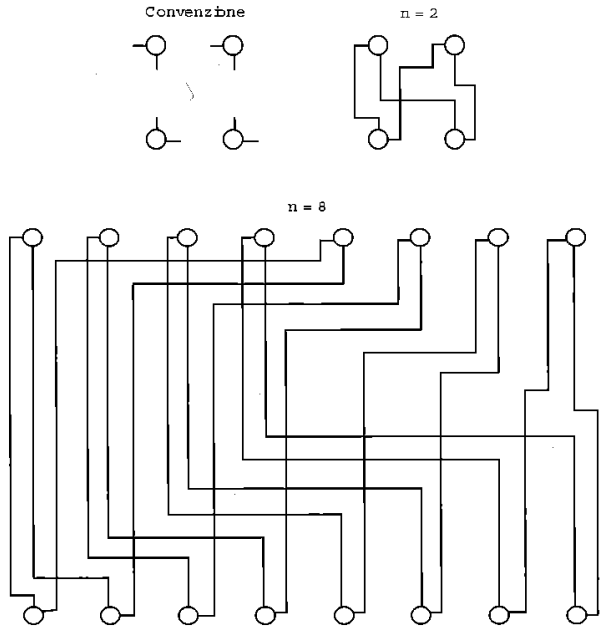


Figura 29.8: Layout delle connessioni shuffle per $n = 2i$, con $i = 1, 3$.

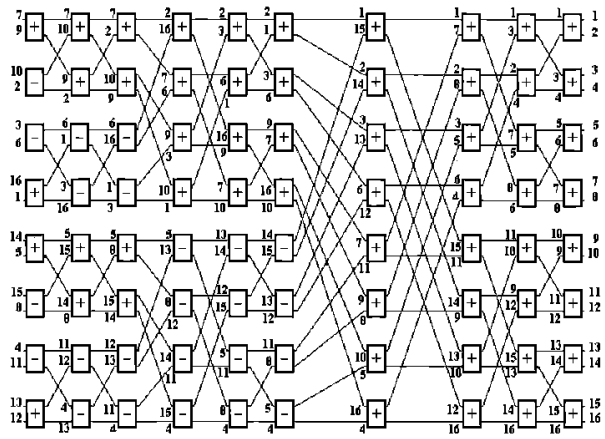


Figura 29.9: Grafo della computazione (di Batcher) corrispondente al diagramma di ordinamento dell'algoritmo "mergesort bitonico" illustrato nella Fig. 28.9.

archi. Ricorsivamente, si forniscono i layout per i due grafi G_1 e G_2, dopodiché si ottiene il layout di G componendo quelli di G_1 e G_2, riaggiungendo sulla griglia i fili corrispondenti agli archi tolti inizialmente. Affinché tale algoritmo produca layout "compatti", occorre che la partizione di G in G_1 e G_2 sia "bilanciata", cioè il numero di nodi di G_1 e G_2 sia circa $n/2$. Inoltre, il numero di archi tolti per disconnettere G deve essere "basso" (possibilmente costante) poiché i fili corrispondenti vanno poi riaggiunti durante la ricombinazione dei due layout. L'area del layout viene ricavata impostando e risolvendo relazioni di ricorrenza.

29.4 Ordinamento

Riconsideriamo l'algoritmo "mergesort bitonico" introdotto nel § 28. In particolare, ridisegnamo il "diagramma" della Fig. 28.9, rappresentando i "confronti" tra coppie di elementi con "nodi" anziché "frecce". In tal caso, anziché avere solo collegamenti ("canali") orizzontali e confrontatori ("frecce") di lunghezza variabile, si hanno confrontatori ("nodi") della stessa dimensione e collegamenti ("archi") possibilmente inclinati. Si ottiene il grafo della computazione illustrato nella Fig. 29.9, detto "grafo di Batcher", dove gli orientamenti delle frecce sono indicati con un "+" o un "−" all'interno dei nodi.

Osservando il grafo della computazione di Batcher, si nota come una sequenza bitonica di lunghezza $m = 2^{h+1}$, con $h \geq 1$, possa essere ordinata mediante una butterfly di rango h preceduta da una fila di connessioni shuffle. Il layout per il grafo di Batcher si può dunque ottenere componendo insieme i layout delle butterfly e delle connessioni shuffle visti nel paragrafo precedente.

Per analizzare la complessità in area e tempo del circuito risultante, consideriamo dapprima il caso in cui ognuno degli n elementi da ordinare sia formato da k bit, con k costante. In tal caso, un confronto tra due elementi può essere ovviamente fatto con

un circuito che richiede area e tempo costanti. Pertanto, ciascuna unità di elaborazione (confrontatore) nel layout del grafo di Batcher richiede $O(1)$ area e $O(1)$ tempo, ed ogni filo ha ampiezza $O(1)$, in quanto trasporta un numero costante di bit. Poiché il grafo di Batcher ha $O(\log^2 n)$ livelli di confrontatori, il tempo T richiesto per l'ordinamento è così $O(\log^2 n)$. Per quanto riguarda l'area, un ragionamento superficiale potrebbe portare alla conclusione che, essendo l'area del layout di una butterfly e di una shuffle $O(n^2)$ ed avendo $O(\log^2 n)$ livelli, si ottenga complessivamente un'area di $O(n^2 \log^2 n)$. In realtà, con un'analisi più attenta della Fig. 29.9, si nota che il numero totale di linee di griglia verticali è $O(n)$ e non $O(n \log^2 n)$. Infatti, posto $n = 2^{h+1}$, il numero delle linee di griglia verticali è dato da quello di h butterfly, rispettivamente di rango $1, 2, \ldots, h$, e da quello di h shuffle che occupano, rispettivamente $4, 8, \ldots, 4h$ linee verticali. Il numero totale di linee verticali è dato pertanto da

$$\sum_{1 \leq j \leq h} \left(j + 1 + \sum_{1 \leq i \leq j} 2^i + 4j \right) = \sum_{1 \leq j \leq \log n - 1} \left(5j + 2^{j+1} - 1 \right)$$

che cresce come $O(n)$. Essendo $O(n)$ anche il numero di linee di griglia orizzontali, l'area A è pertanto $O(n^2)$. Il prodotto AT^2 è così $O(n^2 \log^4 n)$.

Consideriamo adesso il caso in cui il numero k di bit degli n elementi da ordinare non sia una costante. Ogni filo del layout per il grafo di Batcher ha in tal caso ampiezza $O(k)$. Inoltre, ogni confrontatore richiede $O(\log k)$ tempo ed $O(k \log k)$ area. Infatti, il grafo della computazione di un confrontatore è un albero binario completo, che riceve i bit degli elementi da confrontare nelle foglie. La computazione di due elementi X ed Y è effettuata nel modo seguente. La generica foglia confronta i bit x_i ed y_i, e trasmette al padre il valore 2 (in binario), se $x_i < y_i$, il valore 0, se $x_i = y_i$, ed il valore 1, se $x_i > y_i$. Ogni nodo interno trasmette al padre il valore proveniente dal figlio destro, se dal figlio sinistro riceve 0, oppure il valore proveniente dal figlio sinistro, se dal figlio sinistro riceve 1 o 2. Il risultato del confronto, emesso dalla radice, è così 1 se $X < Y$, 0 se $X = Y$, oppure 2, se $X > Y$. A seconda di tale risultato, gli elementi X ed Y sono emessi sugli opportuni fili di uscita.

Esempio 29.3 (Confrontatore a k bit). Un esempio di confronto tra gli elementi $X = 10010011$ ed $Y = 10011010$ è mostrato nella Fig. 29.10. □

In questo modo, il tempo T richiesto per l'ordinamento diviene $O(\log^2 n \log k)$, mentre l'area A è $O(n^2 k \log k)$. Quindi, il prodotto AT^2 diventa $O(n^2 k \log^4 n \log^3 k)$.

Concludiamo notando come i dati attraversino il circuito di Batcher passando ad ogni istante per le unità di elaborazione di un solo, e il medesimo, livello. Pertanto, è possibile immettere nel circuito più sequenze da ordinare, ciascuna ritardata di $O(1)$ tempo rispetto alla precedente. Con questa tecnica, detta pipeline (o "catena di montaggio"), si introduce un'ulteriore forma di parallelismo nel circuito. Infatti, a parte un ritardo iniziale pari al tempo T per attraversare l'intero circuito e produrre la prima sequenza ordinata, le successive sequenze ordinate sono prodotte dal circuito ciascuna dopo $O(1)$ tempo rispetto alla precedente. In questo modo, occorre un tempo complessivo $C = T + m$ per ordinare m sequenze. Pertanto, scegliendo m dello stesso ordine di grandezza di T, occorre C/m tempo ammortizzato per ordinare ciascuna delle m sequenze in ingresso, che è $O(1)$.

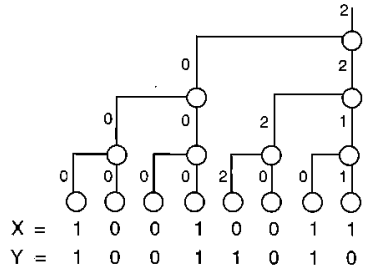


Figura 29.10: Un circuito confrontatore tra due elementi X ed Y di $k = 8$ bit.

29.5 Moltiplicazione di matrici

Consideriamo adesso il problema di moltiplicare due matrici $A = [a_{ij}]$ e $B = [b_{ij}]$ di dimensioni $n \times n$ tali che ciascun elemento sia formato da un numero costante di bit. Vediamo due algoritmi VLSI diversi che, combinati insieme, ne forniscono uno, proposto da Preparata e Vuillemin (1980), con complessità AT^2 ottima pari a $O(n^4)$ per ogni valore di T compreso nell'intervallo $[\log n, n]$.

Poiché ogni elemento è formato da un numero costante di bit, la somma e la moltiplicazione di due elementi richiedono tempo ed area costanti. Un primo circuito banale che ha AT^2 ottima, dove A è $O(n^2)$ mentre T è $O(n)$, è il cosiddetto "moltiplicatore sistolico", il cui grafo della computazione consiste in una mesh quadrata $n \times n$.

Tale algoritmo è analogo a quello visto nel § 28, con la differenza che gli elementi delle due matrici non sono memorizzati e traslati ciclicamente all'interno della mesh, bensì vengono immessi dall'esterno opportunamente traslati e si muovono all'interno del circuito in due direzioni in modo sincrono. Gli elementi di A attraversano il circuito da sinistra verso destra, mentre quelli di B lo attraversano dall'alto verso il basso. Ogni unità di elaborazione della mesh calcola il prodotto dei due elementi che giungono da sinistra e dall'alto, sommandolo al contenuto di un suo registro accumulatore, inizializzato a zero, ed emette quindi i dati ricevuti sui lati opposti a quelli di ingresso. Dopo $O(n)$ tempo, tutti gli elementi delle matrici A e B hanno attraversato la mesh, e vengono emessi i valori degli elementi della matrice C risultante, che sono stati calcolati nei registri accumulatori.

Esempio 29.4 (Moltiplicatore sistolico di matrici). Un esempio di "moltiplicatore sistolico" per $n = 4$, con gli elementi in ingresso traslati opportunamente, è mostrato nella Fig. 29.11. Per meglio comprendere il flusso di attraversamento dei dati nel circuito, è possibile ricalcare gli elementi delle matrici su due fogli trasparenti, e quindi spostare tali fogli in modo sincrono, come descritto nell'algoritmo. □

Consideriamo adesso un "moltiplicatore ricorsivo" che richiede tempo $O(\log n)$. Si assumano $n = 2^k$ e le matrici di ingresso A e B (ed anche la matrice prodotto C) scomposte ciascuna in quattro blocchi di dimensione $n/2 \times n/2$, come descritto nel § 16. In questo modo, dalla usuale formula

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} \qquad 1 \le i \le 2, 1 \le j \le 2,$$

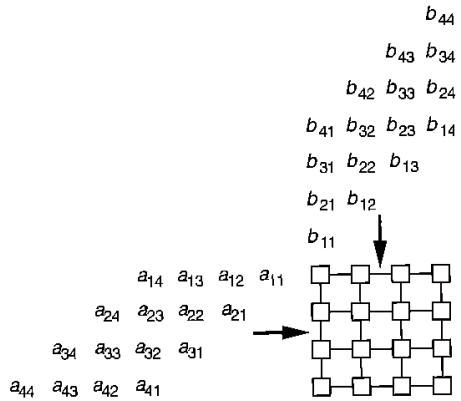


Figura 29.11: Un "moltiplicatore sistolico" per matrici 4×4.

per la moltiplicazione a blocchi, segue che per moltiplicare tra loro due matrici $n \times n$ occorrono 8 moltiplicazioni e 4 addizioni di matrici $n/2 \times n/2$. Si ottiene pertanto un layout rettangolare per la moltiplicazione a blocchi delle matrici $n \times n$ definito ricorsivamente in termini di 8 moltiplicatori per matrici di dimensioni dimezzate. Tale layout è mostrato nella Fig. 29.12, dove ogni filo rappresenta un fascio di $n^2/4$ fili, uno per ciascun elemento di un blocco $n/2 \times n/2$, e contiene al suo interno anche 4 moduli addizionatori di matrici $n/2 \times n/2$, che schematizzano $n^2/4$ addizionatori tra coppie di elementi delle matrici, mentre i pallini neri rappresentano dei "biforcatori", che hanno l'unico scopo di duplicare i dati.

La definizione ricorsiva del moltiplicatore si arresta quando $n = 1$, e in tal caso il moltiplicatore si riduce ad un moltiplicatore tra due elementi, che richiede area e tempo costanti. Ispezionando la Fig. 29.12, ci si rende conto che il layout di un moltiplicatore di matrici $n \times n$ è un rettangolo che contiene al suo interno 8 moltiplicatori per matrici $n/2 \times n/2$, identici al moltiplicatore più grande, a parte una rotazione di 90° ed un ribaltamento orizzontale. Siano $H(n)$ e $W(n)$ l'altezza e la base, rispettivamente, di un moltiplicatore $n \times n$. Si ottengono le seguenti relazioni di ricorrenza "incrociate":

$$\begin{aligned} H(n) &= 1, && \text{per } n = 1, \\ H(n) &= 8W(n/2), && \text{per } n > 1, \end{aligned}$$

$$\begin{aligned} W(n) &= 1, && \text{per } n = 1, \\ W(n) &= H(n/2) + 12(n/2)^2, && \text{per } n > 1. \end{aligned}$$

Per ricavare le soluzioni, si procede sostituendo le definizioni di $W(n/2)$ e $H(n/2)$ in quelle di $H(n)$ e $W(n)$, rispettivamente, ottenendo:

$$\begin{aligned} H(n) &= 8\left(H(n/4) + 12\,(n/4)^2\right) = 8H(n/4) + 6n^2 \\ W(n) &= 8W(n/4) + 12\,(n/2)^2 = 8W(n/4) + 3n^2 \end{aligned}$$

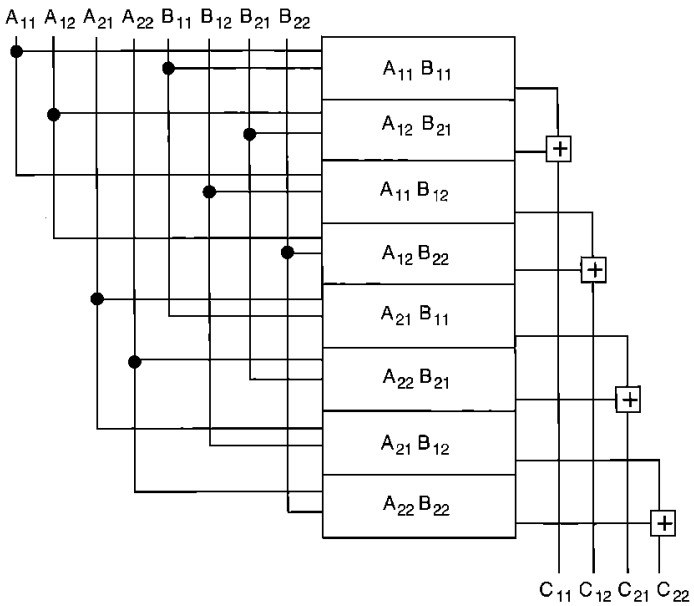


Figura 29.12: Un "moltiplicatore ricorsivo" per matrici a blocchi.

Per entrambe, applicando il teorema delle ricorrenze lineari con partizione bilanciata (cfr. § 16), si ricava $\alpha = \log 8/\log 4 = 3/2 < 2 = \beta$, e quindi sia $H(n)$ che $W(n)$ sono $O(n^2)$. L'area $A = H(n)W(n)$ del moltiplicatore ricorsivo è pertanto $O(n^4)$. Il tempo T del moltiplicatore ricorsivo è invece $O(\log n)$, poiché sviluppando la ricorsione, si vede che il circuito consiste di $2\lceil \log n \rceil + 1$ livelli di unità di elaborazione, dei quali i primi $\lceil \log n \rceil$ sono biforcatori, seguiti da un unico livello di moltiplicatori (tra numeri), seguiti infine da $\lceil \log n \rceil$ livelli di addizionatori (anch'essi tra numeri). Poiché gli elementi delle matrici sono formati da una quantità costante di bit, ogni biforcatore, moltiplicatore, ed addizionatore prende $O(1)$ tempo. Poiché T è $O(\log n)$, il prodotto AT^2 è $O(n^4 \log^2 n)$, che non è ottimo.

Vediamo ora come una combinazione del "moltiplicatore sistolico" e del "moltiplicatore ricorsivo", insieme ad un utilizzo della tecnica "pipeline", permettano di ottenere un circuito con AT^2 ottimo e tempo $O(\log n)$. Innanzitutto, osserviamo come nel moltiplicatore ricorsivo ad ogni istante tutti i segnali si trovino allo stesso livello. Pertanto, introducendo una coppia di matrici diverse ad ogni unità di tempo, è possibile effettuare m moltiplicazioni di matrici $n \times n$ in tempo $O(m + \log n)$. In secondo luogo, le matrici A e B possono essere scomposte in r^2 blocchi, ciascuno formato da una sottomatrice $n/r \times n/r$. A questo punto, è possibile considerare un moltiplicatore sistolico formato da una mesh $r \times r$, dove ognuno degli r^2 nodi consiste di un moltiplicatore ricorsivo per matrici $n/r \times n/r$, utilizzato in pipeline. Ogni nodo della mesh ha così area $O(n^4/r^4)$, e quindi l'area totale A è $O(n^4/r^2)$. Il tempo necessario ad ogni sottomatrice per attraversare la mesh è $O(r)$, ed ogni nodo della mesh deve effettuare r moltiplicazioni di matrici $n/r \times n/r$ che giungono ad istanti di tempo consecutivi. Poiché è utilizzata la tecnica pipeline, queste r moltiplicazioni richiedono tempo $O(r + \log(n/r))$. Pertanto, per ogni valore di r compreso nell'intervallo $[\log n, n]$, risulta che T è $O(r)$. Si ottiene così un prodotto AT^2 che cresce come $O(n^4)$, che è ottimo.

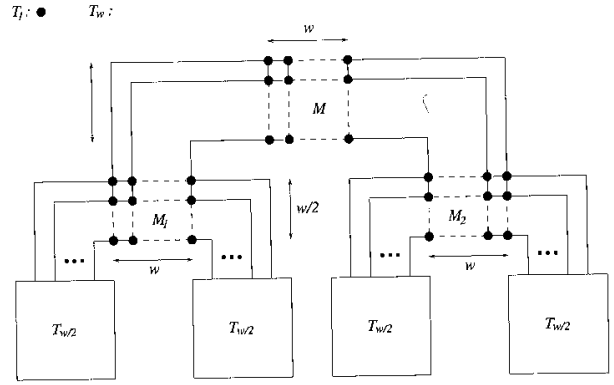
29.6 Esercizi

Esercizio 29.1 (Traslazione ciclica). Si consideri il problema di traslare ciclicamente verso sinistra una sequenza di n bit. Dimostrare che l'area A di un qualsisi circuito che risolva il problema è $\Omega(n)$.

Esercizio 29.2 (Layout alberi binari). Modificare il layout ad H per un albero binario completo con 2^{2i} foglie in modo da connettere tra loro ogni coppia di nodi fratelli. Dimostrare che i nuovi collegamenti possono essere introdotti nel layout ad H aumentandone l'area solo di un fattore costante.

Esercizio 29.3 (Layout alberi binari). Modificare il layout ad H per un albero binario completo con 2^{2i} foglie in modo da connettere tutte le foglie in una lista lineare. Dimostrare che i nuovi collegamenti possono essere introdotti nel layout ad H, aumentandone l'area solo di un fattore costante.

Esercizio 29.4 (Tree of meshes). Per $w = 2^k$, si consideri la famiglia di grafi T_w, detta "tree of meshes", definita in base al seguente schema ricorsivo:



dove M è una mesh di dimensioni $w \times w$ e M_1 e M_2 sono due mesh di dimensioni $w/2 \times w$. Dimostrare che per T_w esiste un layout di area $O(w^2 \log^2 w)$.

Esercizio 29.5 (Mesh of trees). Per $n = 2^k$, una "mesh of trees" $n \times n$ è una usuale mesh quadrata, dove i nodi di ciascuna riga sono le foglie di un albero binario completo, detto "albero-riga", ed i nodi di ciascuna colonna sono le foglie di un albero binario completo, detto "albero-colonna" (si noti che ogni nodo della mesh originale è la foglia sia di un albero-riga che di un albero-colonna). Si dimostri che esiste un layout di area $O(n^2 \log^2 n)$.

Esercizio 29.6 (Layout collineare). Dato un grafo G, un layout per G nel modello a griglia si dice collineare se tutti i nodi di G giacciono su punti di griglia di ascisse consecutive e stessa ordinata. Si dimostri che un albero binario qualsiasi di n nodi (non necessariamente completo né bilanciato) ammette un layout collineare di area $O(n \log n)$.

Esercizio 29.7 (Torneo). Si progetti un algoritmo VLSI per ordinare n interi di k bit (con k costante) in tempo $O(\log n)$ utilizzando l'algoritmo del "torneo" visto nel § 27. (Suggerimento: si impieghi la "mesh of trees" dell'Esercizio 29.5).

Esercizio 29.8 (Bubblesort). Si progetti un algoritmo VLSI per ordinare n interi di k bit (con k costante) in tempo $O(n)$ parallelizzando l'algoritmo "bubblesort" (cfr. Esercizio 15.1).

Capitolo 30

Algoritmi per mesh riconfigurabili

Lo sviluppo della tecnologia VLSI ha offerto la possibilità di costruire circuiti con migliaia di processori integrati su una piastrina di silicio. Per ridurre i costi di produzione, è opportuno che tali circuiti siano caratterizzati da schemi di interconnessione semplici e regolari, come per esempio la mesh. La mesh, però, presenta lo svantaggio di avere un diametro elevato, che rende elevato anche il tempo richiesto per la computazione. Per ridurre il tempo di computazione, pur mantenendo la semplicità e la regolarità della mesh, sono stati introdotti di recente modelli di "mesh con sottobus riconfigurabili dinamicamente". Tali modelli, detti anche più semplicemente "mesh riconfigurabili", sono stati formalizzati da vari autori, tra i quali Ben-Asher, Peleg, Ramaswami e Schuster (1991) e Miller, Prasanna, Reisis e Stout (1993).

Una mesh riconfigurabile consiste di una usuale mesh dove ogni nodo è una unità di elaborazione che possiede una piccola memoria locale ed uno "switch" con capacità di riconfigurazione "dinamica". Lo switch consiste di 4 porte (di solito indicate con le iniziali E, W, N, S dei punti cardinali) sulle quali sono incidenti gli archi provenienti dagli altri nodi. Ogni unità di elaborazione può decidere autonomamente di riconfigurare il proprio switch durante la computazione, stabilendo connessioni fisiche tra coppie di porte. In questo modo, gli archi incidenti sulle due porte tra loro connesse vengono "uniti" in un unico arco, e di fatto si comportano come se formassero un unico filo. Un segnale che viaggia su tale filo non è ritardato dall'attraversamento dei nodi intermedi e può raggiungere in tempo $O(1)$ qualsiasi nodo del filo. Gli archi della mesh sono così utilizzati come se fossero "mattoni" per costruire "catene", dette "sottobus". Riconfigurando tutti gli switch, si ottiene una copertura della mesh con "sottobus disgiunti". Un segnale può viaggiare su un sottobus in tempo $O(1)$, indipendentemente dalla lunghezza del sottobus, e può essere letto da qualsiasi unità di elaborazione facente parte del sottobus.

Ogni singolo passo di una computazione parallela della "mesh riconfigurabile" richiede tempo $O(1)$ ed è suddiviso nelle seguenti fasi:

1. Ogni unità di elaborazione della mesh effettua una computazione locale che richiede tempo $O(1)$;
2. Ogni unità di elaborazione riconfigura autonomamente il proprio switch in tempo $O(1)$;
3. Al più una unità di elaborazione per ciascun sottobus trasmette un segnale sul proprio sottobus;
4. Alcune unità di elaborazione connesse a ciascun sottobus (eventualmente anche tutte) leggono il segnale dopo un tempo $O(1)$.

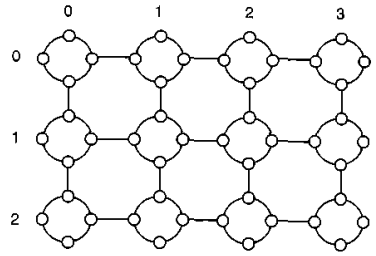


Figura 30.1: Una mesh riconfigurabile con $m = 3$ ed $n = 4$.

L'ipotesi chiave sulla quale si basa tale modello è essenzialmente che il tempo di riconfigurazione degli switch sia dominante in confronto al tempo richiesto per diffondere un segnale ad un numero arbitrario di unità di elaborazione. Questa ipotesi è in realtà troppo forte per la tecnologia elettronica VLSI, dove il tempo per trasmettere un dato su un sottobus che include n unità di elaborazione, benché inferiore ad $O(\log n)$, non è sicuramente indipendente da n, e quindi non può essere $O(1)$. Invece, l'applicazione più promettente per le mesh riconfigurabili è data dalla tecnologia ottica, dove il tempo per riconfigurare gli switch è effettivamente alto in confronto a quello per la diffusione dei dati. Infatti, in un calcolatore ottico i segnali trasmessi sui sottobus consistono di raggi di luce "deviati" dagli switch, che operano come specchi, ed il tempo per diffondere un dato ad n unità di elaborazione è effettivamente $O(1)$.

Nel seguito di questo capitolo, sono dapprima trattate alcune operazione di base sulle mesh riconfigurabili, come sommare e permutare numeri in tempo $O(1)$. Successivamente, sono presentati algoritmi che richiedono tempo $O(1)$ per risolvere i problemi dell'ORDINAMENTO e della MASSIMA SOTTOSEQUENZA COMUNE.

30.1 Operazioni di base

Una mesh riconfigurabile di dimensione $m \times n$ è un modello parallelo sincrono formato da mn unità di elaborazione identiche disposte su m righe ed n colonne, dove ogni unità di elaborazione $P_{i,j}$ è connessa ai suoi (al più) quattro vicini attraverso le quattro porte N (nord), E (est), W (ovest), S (sud) del suo switch, come mostrato nella Fig. 30.1 per $m = 3$ ed $n = 4$.

Le dieci configurazioni possibili per ogni switch sono mostrate nella Fig. 30.2 dove, per esempio, NS indica che le porte N ed S sono tra loro connesse e le porte E e W sono disconnesse, mentre O indica che tutte e quattro le porte sono disconnesse. In questo modo, configurando NS tutti gli switch di una colonna, è possibile distribuire in tempo $O(1)$ un dato a tutte le unità di elaborazione della medesima colonna, mentre configurando EW tutti gli switch di una riga, è possibile distribuirlo a tutte le unità di elaborazione della medesima riga. Ovviamente, ciò può essere fatto in parallelo su tutte le righe e tutte le colonne. Per esempio, configurando EW, NS gli switch di tutte le unità di elaborazione della mesh, è possibile distribuire in tempo $O(1)$ il contenuto di una colonna a tutte le altre colonne ed il contenuto di una riga a tutte le altre righe.

La possibilità di riconfigurare dinamicamente i sottobus di una mesh permette di adattare le interconnessioni in base non solo al "grafo della computazione"

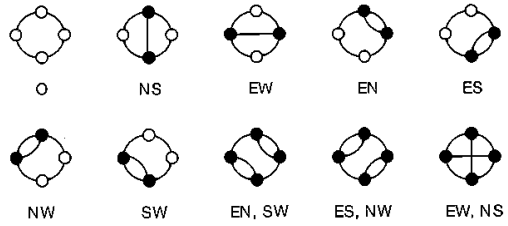


Figura 30.2: Le dieci configurazioni possibili per uno switch.

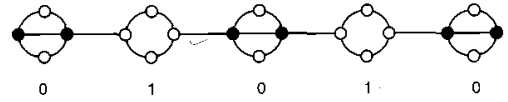


Figura 30.3: Calcolo dell'**or** di $n = 5$ bit.

del problema, ma anche e soprattutto ai particolari valori dei dati di ingresso del problema.

Vediamo come primo esempio come computare l'**or** di n bit $b_0, b_1, \ldots, b_{n-1}$ in tempo $O(1)$ con una mesh $1 \times n$. Si assuma che l'unità di elaborazione P_i della mesh riceva in input il bit b_i, $0 \leq i \leq n-1$. Se $b_i = 1$, allora P_i configura il proprio switch O, se invece $b_i = 0$, allora P_i configura il proprio switch EW. In questo modo, si ottiene un solo sottobus che attraversa tutti i nodi se e solo se tutti i bit in ingresso valgono 0. A questo punto, P_0 trasmette un bit a 1 sulla propria porta W, mentre P_{n-1} legge dalla propria porta E. Se P_{n-1} riceve 1, allora è connesso con P_0 perché tutti i bit in ingresso valgono 0, e quindi P_{n-1} emette come risultato 0. Altrimenti, se P_{n-1} non riceve alcun valore, allora è sconnesso da P_0 perché almeno un bit in ingresso vale 1, e quindi P_{n-1} emette come risultato 1. In modo del tutto analogo si può calcolare l'**and** di n bit.

Esempio 30.1 ("Or" di n bit). Un esempio di riconfigurazione per calcolare l'**or** dei 5 bit 0, 1, 0, 1, 0 è mostrato nella Fig. 30.3. □

Consideriamo un secondo esempio un po' più complicato, dove si computa l'**exor** ("or esclusivo") di n bit $b_0, b_1, \ldots, b_{n-1}$ in tempo $O(1)$ con una mesh $3 \times 2n$. Ricordiamo che l'**exor** di n bit è del tutto equivalente alla loro somma modulo 2, in quanto il risultato è 1 se e solo se il numero di bit che valgono 1 è dispari. Si consideri una mesh $3 \times 2n$ costituita da n "blocchi" 3×2 collegati in serie, dove ogni blocco 3×2 corrisponde ad un singolo bit b_i in ingresso e si configura, come mostrato nella Fig. 30.4a), a seconda del valore di b_i. Se $b_i = 1$, le due righe superiori si "incrociano" ed un eventuale segnale che provenga su una di queste due righe passa sull'altra riga (la riga inferiore serve soltanto per effettuare lo scambio tra le altre due). Se invece $b_i = 0$, allora un eventuale segnale resta sulla stessa riga. Pertanto, se un bit ad 1 è trasmesso da $P_{0,0}$ sulla sua porta W, l'**exor** di tutti i bit in ingresso appare sulla

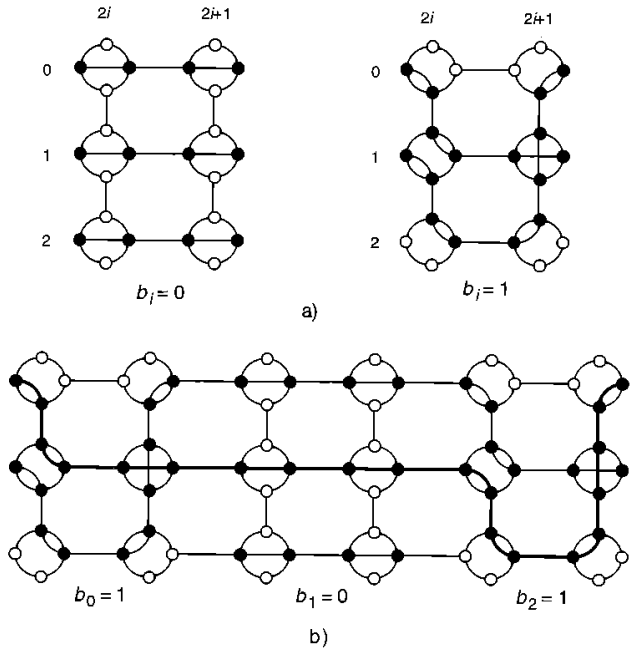


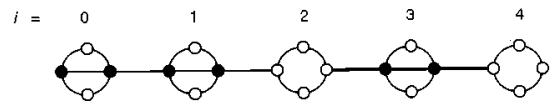
Figura 30.4: Calcolo dell'**exor** di $n = 3$ bit. a) Configurazione degli switch. b) 1 **exor** 0 **exor** 1 = 0.

colonna di destra. Se il bit trasmesso da $P_{0,0}$ raggiunge la porta E di $P_{0,2n-1}$, allora il risultato è 0, mentre se raggiunge la porta E di $P_{1,2n-1}$, allora il risultato è 1.

Esempio 30.2 ("Exor" di n bit). Un esempio di riconfigurazione per calcolare l'**exor** dei 3 bit 1, 0, 1 è mostrato nella Fig. 30.4b). Il risultato è 0, poiché c'è un numero pari di bit in ingresso che valgono 1. □

Con un algoritmo analogo a quello visto per l'**exor**, ma usando una mesh di $m+1$ righe anziché 3, è possibile calcolare in tempo O(1) la somma modulo m (anziché modulo 2) di n bit. Tale algoritmo può essere facilmente adattato anche per calcolare in tempo O(1) la somma $(a + \sum_{0 \le i \le n-1} b_i) \mod m$, dove a è un intero compreso tra 0 ed $m - 1$. Con un algoritmo molto più complicato, dovuto a Bertossi e Mei (1998), è invece possibile calcolare le somme prefisse di n bit in tempo O(1), usando una mesh riconfigurabile di dimensione $2n \times O(\log^2 n / \log\log n)$.

Come terzo esempio di operazione di base, vediamo come effettuare l'addizione di due numeri di n bit. L'algoritmo è simile a quello descritto per l'**or** e richiede tempo O(1) su una mesh $1 \times n$. Siano $a_0 a_1 \cdots a_{n-1}$ e $b_0 b_1 \cdots b_{n-1}$ i due numeri da sommare (dove i bit meno significativi sono a sinistra e quelli più significativi a destra solo per mantenere lo stesso ordine degli indici dei nodi nella mesh). Ciascuna unità di elaborazione P_i configura il proprio switch in base ai valori di a_i e b_i, a seconda che essi inducano una condizione di generazione del riporto o di propagazione del riporto.



i =	0	1	2	3	4
a_i =	0	0	1	1	0
b_i =	1	1	1	0	0
r_i =	0	0	0	1	1
c_i =	1	1	0	0	1

Figura 30.5: Addizione di due numeri di $n = 5$ bit, dove i bit r_i indicano i riporti intermedi, mentre i bit c_i indicano il risultato.

Se $a_i = b_i$, allora P_i configura lo switch O, sconnettendo tutte le porte, altrimenti lo configura EW. P_i genera un riporto trasmettendo un bit ad 1 sulla sua porta E quando $a_i = b_i = 1$. Un "supero" è segnalato da un bit ad 1 che raggiunge la porta E di P_{n-1}. Ogni P_i effettua la somma modulo 2 di a_i, b_i, e dell'eventuale bit di riporto.

Esempio 30.3 (Addizione). Un esempio di addizione tra due numeri di 5 bit è mostrato nella Fig. 30.5, dove i bit r_i indicano i valori dei riporti intermedi, mentre i bit c_i indicano quelli del risultato. Si noti che un riporto è generato da P_2 e che tale riporto si propaga fino a P_4. □

Usando un algoritmo simile a quello per l'addizione, ma costruendo un addizionatore a k stadi, è possibile effettuare la SOMMATORIA di n numeri di k bit in tempo O(1) usando una mesh di dimensioni $n \times nk$. Con tecniche molto più sofisticate, la SOMMATORIA può essere invece computata in tempo O(1) con una mesh $O(k \log n + \log^2 n / \log\log n) \times nk$, mentre le SOMME PREFISSE possono essere computate in tempo O(1) con una mesh $O((k^2 + \log^2 n)/\log(k + \log n)) \times O(nk + n \log n)$. Entrambi questi algoritmi sono stati proposti da Bertossi e Mei (1998) e rappresentano attualmente i migliori algoritmi conosciuti per la SOMMATORIA e le SOMME PREFISSE.

Come ultimo esempio di operazione di base, mostriamo come sia possibile permutare n elementi $a_0, a_1, \ldots, a_{n-1}$ in tempo O(1) con una mesh $n \times n$. Sia π la permutazione desiderata ed assumiamo che gli elementi da permutare e la permutazione π siano contenuti nella riga 0 della mesh, in modo che $P_{0,j}$ contenga a_j e π_j. Dopo aver trasmesso π_j lungo la colonna j con tecniche standard, ogni $P_{i,j}$ configura il proprio switch NW, se $\pi_j = i$, oppure EW, NS, altrimenti. Successivamente, ogni processore della riga 0 trasmette il proprio elemento sulla sua porta N. Il generico dato a_j, trasmesso da $P_{0,j}$, percorre il sottobus lungo la colonna j e "svolta" a sinistra quando incontra $P_{\pi_j,j}$, proseguendo sulla riga π_j fino a raggiungere $P_{\pi_j,0}$, che lo legge dalla sua porta W. Poiché π è una permutazione, vi è un solo switch configurato NW per ogni riga e per ogni colonna. Pertanto, gli elementi $a_0, a_1, \ldots, a_{n-1}$ si ritrovano permutati nella colonna 0 in accordo a π. Rieseguendo "a ritroso" lo stesso algoritmo, ma scegliendo stavolta la permutazione "identità" $\pi_j^{ID} = j$, gli elementi possono essere trasferiti nuovamente dalla colonna 0 alla riga 0 senza cambiarne l'ordine rispetto alla permutazione π specificata.

$$\begin{bmatrix} 21 & 1 & 16 \\ 2 & 27 & 25 \\ 15 & 22 & 3 \\ 4 & 14 & 20 \\ 24 & 13 & 17 \\ 19 & 6 & 5 \\ 26 & 23 & 11 \\ 8 & 7 & 18 \\ 10 & 12 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \\ 9 & 18 & 27 \end{bmatrix}$$

Figura 30.6: Ordinamento per colonne di una matrice A di dimensione 9×3.

$$\begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \\ 9 & 18 & 27 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \\ 19 & 20 & 21 \\ 22 & 23 & 24 \\ 25 & 26 & 27 \end{bmatrix}$$

a) Fase 2

$$\begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \\ 9 & 18 & 27 \end{bmatrix} \rightarrow \begin{bmatrix} -\infty & 6 & 15 & 24 \\ -\infty & 7 & 16 & 25 \\ -\infty & 8 & 17 & 26 \\ -\infty & 9 & 18 & 27 \\ 1 & 10 & 19 & +\infty \\ 2 & 11 & 20 & +\infty \\ 3 & 12 & 21 & +\infty \\ 4 & 13 & 22 & +\infty \\ 5 & 14 & 23 & +\infty \end{bmatrix}$$

b) Fase 6

Figura 30.7: Le fasi 2 e 6 dell'algoritmo "column-sort".

30.2 Ordinamento

Consideriamo adesso il problema dell'ORDINAMENTO di una sequenza di n elementi. Tra i molti algoritmi che risolvono il problema in $O(1)$ tempo, quello migliore è dovuto a Nigam e Sahni (1994) e consiste in un riadattamento per una mesh riconfigurabile $n \times n$ dell'algoritmo "column-sort" ideato da Leighton (1985).

L'algoritmo "column-sort" riceve in ingresso una matrice A di dimensione $r \times s$, dove $r \geq 2(s-1)^2$ ed r **mod** $s = 0$, e la riordina per colonne. In altri termini, l'algoritmo sposta gli $n = rs$ elementi di A in modo che risulti ordinata la sequenza formata dagli elementi della prima colonna, seguiti da quelli della seconda colonna, e così via fino a quelli dell'ultima colonna, come mostrato nella Fig. 30.6 per $r = 9$ ed $s = 3$.

L'algoritmo "column-sort" consta di 8 fasi. Durante le fasi dispari (1, 3, 5 e 7) vengono ordinati gli elementi di ogni colonna. Nella fase 2 gli elementi di A sono presi in ordine di colonna e ridisposti in ordine di riga (si veda la Fig. 30.7a). La fase 4 effettua l'esatto contrario, dove gli elementi di A sono presi in ordine di riga e ridisposti in ordine di colonna. La fase 6 consiste di uno spostamento in avanti di $\lfloor r/2 \rfloor$ posizioni degli elementi della matrice (si veda la Fig. 30.7b). Infine, la fase 8 consiste nell'operazione inversa a quella descritta nella fase 6. Leighton (1985) ha dimostrato che queste 8 fasi sono sufficienti per ordinare la matrice A per colonne.

Per semplicità, illustriamo una versione meno efficiente ma più facile da descrivere dell'algoritmo di Nigam e Sahni, utilizzando una mesh riconfigurabile di dimensione $n \times \frac{1}{2}n^{3/4}$ per ordinare in tempo $O(1)$ una matrice A di $rs = \frac{1}{2}n^{3/4}$ elementi, dove $r = n^{1/2}$ ed $s = \frac{1}{2}n^{1/4}$.

Gli elementi della matrice A sono caricati per colonna sulla prima riga della mesh, dopodiché sono eseguite le 8 fasi del "column-sort".

Le fasi dispari (1, 3, e 5) sono effettuate configurando dapprima tutti gli switch NS, in modo da distribuire il contenuto della prima riga della mesh (cioè gli $\frac{1}{2}n^{3/4}$ elementi da ordinare) a tutte le n righe della mesh. La mesh è partizionata in rs sottomesh quadrate $B_{j,k}$, ciascuna di dimensione $r \times r$, per $1 \leq j \leq r$, $1 \leq k \leq s$. La generica sottomesh $B_{j,k}$ contiene tutti gli elementi della k-esima colonna di A e viene utilizzata per determinare la posizione dell'elemento j-esimo all'interno dell'ordinamento della k-esima colonna di A. Questa computazione è effettuata da $B_{j,k}$ per mezzo di un "torneo", calcolando gli r bit relativi ai confronti "$a_{j,k} \geq a_{i,k}$", $1 \leq i \leq r$, ed effettuandone la sommatoria. Ogni $B_{j,k}$ utilizza i sottobus colonna per spostare gli elementi nella posizione calcolata. Si osservi che possono sorgere conflitti qualora esistano nella stessa colonna di A più elementi con lo stesso valore, poiché tutti questi elementi dovrebbero occupare la medesima posizione. Questi conflitti sono eliminati dando la priorità ai blocchi $B_{j,k}$ con indice di riga inferiore, riempiendo successivamente le posizioni rimaste vuote ricopiando il primo elemento che si trova sulla destra.

Le fasi 2 e 4 sono eseguite banalmente effettuando una permutazione degli elementi della prima riga della mesh. Per quel che riguarda la fase 6, è importante sottolineare come gli elementi $\pm\infty$ aggiunti nel "column-sort" per "aumentare" A di una colonna (si veda la Fig. 30.7b) possano essere ignorati durante l'ordinamento per colonne e lo spostamento di $\lfloor r/2 \rfloor$ posizioni non sia affatto necessario. Infatti, nella fase 6 è sufficiente partizionare la mesh in $r(s+1)$ blocchi in modo che la prima ed ultima colonna della matrice A "aumentata" siano gestite da blocchi di dimensione $r \times \lfloor r/2 \rfloor$, anziché $r \times r$, e quindi eseguire direttamente una fase dispari. Con questo accorgimento, la fase 8 non è più necessaria, poiché gli elementi non si sono spostati nella fase 6, e di fatto le 8 fasi del "column-sort" sono ridotte a 6 soltanto.

30.3 Massima sottosequenza comune

Consideriamo adesso il problema della MASSIMA SOTTOSEQUENZA COMUNE: "Date due stringhe $P = p_1 p_2 \cdots p_m$ e $T = t_1 t_2 \cdots t_n$ di caratteri alfabetici, individuare la lunghezza della più lunga sottosequenza comune tra P e T" (cfr. Esercizio 14.8).

Assumendo che P e T inizino con un carattere "vuoto" λ, che si trova in posizione 0, si può definire una matrice D tale che $D[i, j]$, per $0 \leq i \leq m$ e $0 \leq j \leq n$, è il numero dei caratteri della massima sottosequenza comune tra $p_0 p_1 \cdots p_i$ e $t_0 t_1 \cdots t_j$. Il problema si può allora risolvere tramite le relazioni di "programmazione dinamica" (cfr § 14):

$$D[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ oppure } j = 0, \\ D[i-1, j-1] + 1 & \text{se } i > 0, j > 0, \text{ e } p_i = t_j, \\ \max\{D[i, j-1], D[i-1, j]\} & \text{se } i > 0, j > 0, \text{ e } p_i \neq t_j. \end{cases}$$

Per realizzare questo algoritmo di "programmazione dinamica" su una mesh riconfigurabile, è opportuno considerare una variante del modello visto finora, dove i collegamenti tra unità di elaborazione sono "orientati" (quelli orizzontali da sinistra verso destra e quelli verticali dall'alto verso il basso) e dove, oltre alle 10 configurazioni degli switch mostrate nella Fig. 30.2, sono ammesse le ulteriori 5 configurazioni illustrate nella Fig. 30.8, che permettono di connettere tra loro anche terne di porte oppure tutte e quattro le porte.

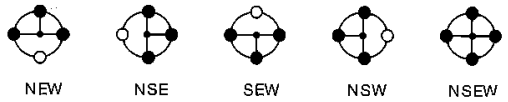


Figura 30.8: Le ulteriori cinque configurazioni ammesse per uno switch.

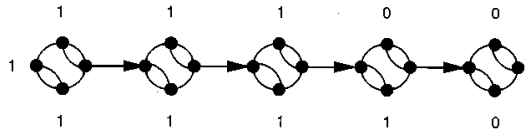


Figura 30.9: Incremento del numero 2 rappresentato su 5 bit.

Con questo modello, è agevole realizzare efficientemente le operazioni di incremento e di massimo, richieste dall'algoritmo di "programmazione dinamica" per la massima sottosequenza comune. Infatti, si rappresenti un intero k, dove $0 \leq k \leq n-1$, con n bit dei quali i primi $k+1$ sono ad 1 ed i rimanenti $n-k-1$ a 0. Allora le operazioni di incremento e massimo possono essere realizzate in tempo O(1) con le reti illustrate nelle Figg. 30.9 e 30.10, rispettivamente. La rete della Fig. 30.9 riceve il numero da incrementare dalle porte N (e la costante 1 da sinistra) e restituisce il risultato sulle porte S.

La rete per il massimo, invece, utilizza la configurazione NSEW sugli switch della diagonale principale, che permette di calcolare l'**or** tra i due bit in ingresso sulle porte N e W, restituendo in uscita il risultato su entrambe le porte S ed E. In questo modo, la rete della Fig. 30.10 restituisce sia dal bordo destro che da quello inferiore il massimo dei due numeri in ingresso dal bordo sinistro e da quello superiore.

Il problema della MASSIMA SOTTOSEQUENZA COMUNE può essere risolto in tempo O(1) su una mesh riconfigurabile di dimensione $O(mh) \times O(mh)$, dove $h = \min\{m, n\}$.

Per ciascuna "casella" $D[i, j]$ della matrice D di "programmazione dinamica" è introdotta una sottomesh $C_{i,j}$ di dimensioni $h \times h$. Il valore $D[i, j]$ è memorizzato nelle unità di elaborazione che stanno sulla diagonale di $C_{i,j}$. Per $i = 0$ oppure $j = 0$, tale valore è la costante 0, altrimenti il valore di $C_{i,j}$ deve essere calcolato in funzione dei valori di $C_{i-1,j-1}$, $C_{i-1,j}$, e $C_{i,j-1}$, in accordo alle relazioni di "programmazione dinamica" viste precedentemente. Le appropriate connessioni tra $C_{i-1,j-1}$, $C_{i-1,j}$, $C_{i,j-1}$ e $C_{i,j}$ sono mostrate nella Fig. 30.11, dove le sottomesh $C_{i,j}$, $S_{i,j}$ ed R hanno la stessa struttura e configurazione della sottomesh illustrata nella Fig. 30.10, e sono pertanto capaci di ricevere in ingresso una coppia di interi dal bordo sinistro e da quello superiore e di restituirne il massimo sia sul bordo destro che su quello inferiore. Le sottomesh "ON/OFF" hanno dimensione $1 \times h$ ed il loro ruolo è soltanto quello di connettere o disconnettere gli archi tra le sottomesh. Ciò è effettuato configurando tutti i loro switch NS oppure O, rispettivamente. La sottomesh "OFF/+1" ha anch'essa dimensione $1 \times h$ ed è capace di disconnettere gli archi o di effettuare un incremento.

La computazione procede nel modo seguente. I caratteri p_i e t_j sono letti da $S_{i,0}$ ed $S_{0,j}$, rispettivamente, e distribuiti con tecniche standard su righe e colonne in

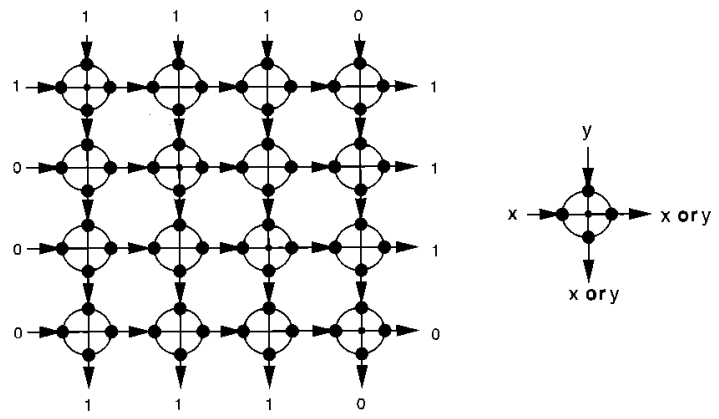


Figura 30.10: Massimo tra 0 (a sinistra) e 2 (in alto), rappresentati su 4 bit. Il risultato è 2, disponibile sia in basso che a destra.

modo che $S_{i,j}$ memorizzi sia p_i che t_j. A seconda del valore di questi due caratteri, $S_{i,j}$ controlla le configurazioni della sottomesh "OFF/+1" e delle due sottomesh "ON/OFF" connesse con archi tratteggiati nella Fig. 30.11. Se $p_i = t_j$, allora queste tre sottomesh sono configurate "+1", "OFF", e "OFF", rispettivamente, permettendo a $C_{i,j}$ di ricevere il valore di $C_{i-1,j-1}$ aumentato di uno. Se invece $p_i \neq t_j$, allora le tre sottomesh sono configurate "OFF", "ON" e "ON", rispettivamente, permettendo a $C_{i,j}$ di ricevere il massimo tra i valori di $C_{i-1,j}$, e $C_{i,j-1}$. Le sottomesh $C_{i,0}$ e $C_{0,j}$ trasmettono la rappresentazione dell'intero 0, e per costruzione i valori che attraversano gli switch nelle diagonali dei $C_{i,j}$ rappresentano proprio i valori di $D[i, j]$. Il risultato è ottenuto in tempo O(1) nella sottomesh $C_{m,n}$.

Con tecniche simili a quelle appena viste per la MASSIMA SOTTOSEQUENZA COMUNE, proposte da Bertossi e Mei (1999), si possono eseguire in tempo O(1) su mesh riconfigurabili anche altri algoritmi di "programmazione dinamica" che risolvono ulteriori problemi computazionali, come per esempio quello dei CAMMINI MINIMI da un singolo nodo a tutti gli altri nodi di un grafo orientato (con lunghezze anche negative sugli archi).

30.4 Esercizi

Esercizio 30.1 (Or prefissi). Si generalizzi l'algoritmo per il calcolo dell'**or** di n bit per calcolarne gli "**or** prefissi" in tempo O(1) con una mesh riconfigurabile di dimensione $1 \times n$.

Esercizio 30.2 (Somma di n bit modulo m). Si generalizzi l'algoritmo per il calcolo dell'**exor** di n bit per calcolare la somma modulo m di n bit in tempo O(1) con una mesh riconfigurabile di dimensione $(m+1) \times n$.

Esercizio 30.3 (Torneo). Si realizzi l'algoritmo del "torneo" per risolvere il problema dell'ORDINAMENTO in tempo O(1) con una mesh riconfigurabile avente $O(n^3)$ unità di elaborazione.

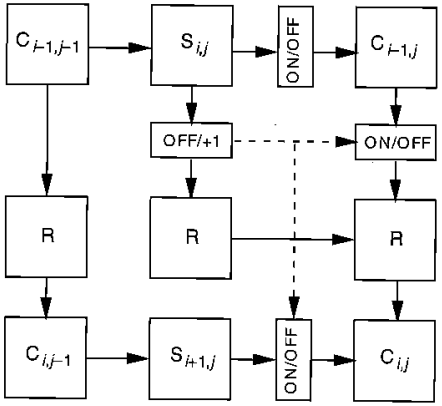


Figura 30.11: Schema di interconnessione tra le sottomesh $C_{i-1,j-1}$, $C_{i-1,j}$, $C_{i,j-1}$, e $C_{i,j}$.

Esercizio 30.4 (Column-sort). Si esegua (a mano) l'algoritmo "column-sort" per ordinare la matrice A mostrata nella Figura 30.6.

Esercizio 30.5 (String-matching k-approssimato). Si modifichi la mesh riconfigurabile per la "massima sottosequenza comune" per risolvere in tempo O(1) il problema dello "string-matching k-approssimato" (cfr. § 13).

Capitolo 31

Algoritmi concorrenti

Nei precedenti quattro capitoli, sono stati considerati algoritmi paralleli per modelli sincroni, dove tutti i processori eseguono lo stesso programma, iniziando e terminando contemporaneamente l'esecuzione della medesima istruzione su dati uguali o diversi. In questo e nel prossimo capitolo, invece, si considerano algoritmi paralleli per modelli asincroni, dove i processori eseguono in generale programmi diversi, iniziando e terminando in istanti differenti l'esecuzione di istruzioni diverse su dati diversi. In particolare, in questo capitolo sono trattati algoritmi per modelli concorrenti, in cui c'è un'unica memoria globale condivisa tra tutti i processori.

In un modello concorrente, ogni processore esegue un programma sequenziale, detto comunemente "processo", che può essere uguale o diverso da quelli degli altri processori. I processi si possono scambiare dati attraverso la memoria comune, modificando e leggendo i valori di variabili condivise. Poiché i processi sono asincroni, non vi è alcuna garanzia che le istruzioni per scambiare i dati tra processi siano eseguite contemporaneamente. Pertanto, le variabili comuni a più processi devono essere accedute in modo mutuamente esclusivo, garantendo che ogni processo possa acquisire la variabile comune esclusivamente per sé per la porzione di codice (detta "sezione critica") comprendente tutte le istruzioni necessarie per l'elaborazione della variabile stessa, senza però precludere la possibilità che altri processi possano godere successivamente del medesimo privilegio.

Esempio 31.1 (Mutua esclusione). Si supponga che due processi concorrenti P e Q debbano entrambi incrementare di 1 il valore di una variabile condivisa x, che inizialmente vale 5:

```
process P                    process Q
  begin                        begin
    ⋮                            ⋮
    p := x;                      q := x;
    ⋮                            ⋮
    p := p + 1;                  q := q + 1;
    ⋮                            ⋮
    x := p;                      x := q;
    ⋮                            ⋮
  end;                         end;
```

Le tre istruzioni esplicitate per ciascun processo formano una "sezione critica" su x. Garantendo la mutua esclusione sulle sezioni critiche, il valore finale di x sarà correttamente 7, come nel caso sequenziale in cui P fosse eseguito prima di Q oppure Q fosse eseguito prima di P. Se invece non è garantita la mutua esclusione su x, le istruzioni dei due processi potrebbero essere eseguite nel seguente ordine:

$p := x;$
$q := x;$
$p := p + 1;$
$q := q + 1;$
$x := p;$
$x := q;$

In tal caso, sia P che Q leggerebbero il valore 5 prima che l'altro processo l'abbia aggiornato a 6, ed il valore finale di x sarebbe scorrettamente 6, come se fosse stato eseguito soltanto uno dei due processi. □

Per garantire la mutua esclusione sugli accessi, i processi utilizzano opportune istruzioni, dette primitive di sincronizzazione. Per semplicità, verranno usate in questo capitolo le primitive **lock** e **unlock**. Se un processo P esegue **lock**(x) e la variabile x è libera, allora P acquisisce x e nessun altro processo potrà accedere ad x finché P non effettuerà una **unlock**(x). Se P esegue **lock**(x) ed x è già stata acquisita da un altro processo Q, allora P resta in attesa finché Q non libererà x. Se ci sono più processi in attesa su x, allora nell'istante in cui x è liberata uno ed uno solo dei processi in attesa acquisirà la variabile x.

Esempio 31.2 (Lock e unlock). Usando le primitive **lock** ed **unlock**, la mutua esclusione sulle sezioni critiche dei processi P e Q dell'Esempio 31.1 può essere realizzata come segue:

```
process P                    process Q
  begin                        begin
    ⋮                            ⋮
    lock(x);                     lock(x);
    p := x;                      q := x;
    ⋮                            ⋮
    p := p + 1;                  q := q + 1;
    ⋮                            ⋮
    x := p;                      x := q;
    unlock(x);                   unlock(x);
    ⋮                            ⋮
  end;                         end;
```

□

Mentre i modelli sincroni sono più adatti per progettare algoritmi paralleli efficienti che risolvono problemi caratterizzati da una struttura molto regolare, i modelli concorrenti si prestano ad esplicitare un potenziale parallelismo insito anche in molti problemi non caratterizzati da una struttura regolare. A differenza dei modelli sincroni, però, i modelli concorrenti presentano alcuni inconvenienti dovuti principalmente proprio alla loro natura asincrona.

In primo luogo, uno scorretto uso delle primitive di sincronizzazione da parte del programmatore può provocare fenomeni di "stallo", cioè di attesa infinita da parte dei processi (tali fenomeni possono comunque essere prevenuti o rilevati con opportune tecniche).

Esempio 31.3 (Stallo). Si considerino i seguenti processi P e Q:

```
process P                    process Q
  begin                        begin
    ⋮                            ⋮
    lock(x);                     lock(y);
    ⋮                            ⋮
    lock(y);                     lock(x);
    ⋮                            ⋮
    unlock(x);                   unlock(x);
    ⋮                            ⋮
    unlock(y);                   unlock(y);
    ⋮                            ⋮
  end;                         end;
```

Si assuma che P esegua **lock**(x), trovi x libera e quindi acquisisca x, e immediatamente dopo Q esegua **lock**(y), trovi y libera, e quindi acquisisca y. Quando P eseguirà **lock**(y), P si porrà in attesa che Q liberi y, e quando Q eseguirà **lock**(x), Q si porrà in attesa che P liberi x. In questo modo, i due processi P e Q saranno bloccati per sempre. □

In secondo luogo, un'analisi teorica delle prestazioni si può discostare parecchio (per eccesso) dalle prestazioni reali. Come misura delle prestazioni è usato il cosiddetto "speedup", ovvero il rapporto tra il tempo richiesto dall'algoritmo sequenziale e quello richiesto dalla sua versione concorrente con p processori. Si può giungere al paradosso che aumentando il numero dei processi si ottengono speedup peggiori, perché aumentano le attese dei processi dovute sia a sincronizzazioni su variabili condivise, sia a conflitti sul circuito d'indirizzamento della memoria per accessi a variabili non condivise! Pertanto, è necessario provare sperimentalmente ciascun algoritmo concorrente proposto su macchine asincrone reali, variandone il numero p di processi, e trovare così il valore di p, che di solito non supera le poche unità, che dà uno speedup ottimale.

Per esprimere l'esecuzione concorrente di p processi identici su p processori, si userà nel programma principale il costrutto

" **for all** i **where** $1 \leq i \leq p$ **do concurrently** PROCESSO(i); "

dove PROCESSO(i) è il nome del processo da eseguire sul processore i. L'effetto dell'esecuzione è analogo a quello di p chiamate simultanee di procedure, che vengono eseguite contemporaneamente ma in modo asincrono. L'istruzione successiva al "**for all** " nel programma principale non è eseguita finché tutti i p processi asincroni non sono terminati. Nel seguito di questo capitolo, sono presentati algoritmi concorrenti per risolvere i problemi del MASSIMO, dell'ORDINAMENTO, dei CAMMINI MINIMI, e del COMMESSO VIAGGIATORE.

31.1 Massimo

Consideriamo un algoritmo concorrente efficiente per trovare il MASSIMO tra n numeri usando p processori ($p \leq n$). Si assuma per semplicità che gli elementi da elaborare siano indicati con $a_1, a_2, \ldots, a_n$. Si abbiano p processori $P_1, P_2, \ldots, P_p$. Ciascun processore P_i esegue lo stesso processo MAX(i), che effettua una scansione su n/p elementi, memorizzandone il massimo in una variabile locale *max-locale*$_i$. Tale risultato sarà successivamente confrontato col contenuto di una variabile globale *max-globale* condivisa tra i p processi, che è inizializzata a $+\infty$ nel programma principale.

```
procedure MASSIMO(a_1, a_2, ... , a_n);
    begin
        max-globale := +∞;
        for all i where 1 ≤ i ≤ p do concurrently MAX(i);
    end;

process MAX(i);
    begin
        max-locale_i := +∞;
        for j := i to n by p do
            if a_j < max-locale_i then max-locale_i := a_j;
        lock(max-globale);
        if max-locale_i < max-globale then
            max-globale := max-locale_i;
        unlock(max-globale);
    end;
```

Il ciclo **for** è eseguito $O(n/p)$ volte in parallelo da ciascuno dei p processi. La sezione critica compresa tra **lock** e **unlock** richiede l'accesso sequenziale alla variabile condivisa *max-globale* da parte di p processi e quindi costa $O(p)$ tempo. Pertanto, la complessità è $O(n/p + p)$. Tale complessità è minimizzata se $p = \sqrt{n}$. In tal caso, il tempo richiesto diventa $O(\sqrt{n})$ e lo speedup è anch'esso $O(n/\sqrt{n}) = O(\sqrt{n})$. Tale valutazione è comunque teorica e ottimista, perché non tiene conto dei conflitti sugli accessi alla memoria comune da parte dei p processori per leggere i dati $a_0, a_1, \ldots, a_{n-1}$.

31.2 Ordinamento

Vediamo adesso come risolvere il problema dell'ORDINAMENTO di una sequenza di n numeri con una versione concorrente dell'algoritmo QUICKSORT che utilizza p processori ($p \ll n$). Si ricorda che il QUICKSORT, su una sequenza $a_1, \ldots, a_n$, sceglie un elemento x, detto perno, riarrangia gli elementi della sequenza in modo che $a_i \leq x$, per $1 \leq i \leq j$, e $a_i \geq x$, per $j \leq i \leq n$, e successivamente riapplica il procedimento alle due porzioni $a_1, \ldots, a_j$ e $a_{j+1}, \ldots, a_n$ (cfr. § 10).

Con p processori e gli elementi da ordinare memorizzati nella memoria globale, può essere mantenuta una pila comune in cui sono memorizzati gli estremi di ciascuna porzione da ordinare. Per accedere alla pila, ogni processo dovrà usare le primitive **lock** e **unlock**.

Ogni processore può eseguire lo stesso processo ciclico, che estrae gli estremi della porzione da ordinare dalla testa della pila, la partiziona sequenzialmente con l'operazione "perno", reinserisce gli estremi delle due porzioni nella pila, e ricomincia daccapo.

Per agevolare l'analisi della complessità, si assumano $n = 2^k - 1$ e $p = 2^m$ con $m < k$. Si indichi con $S(n)$ il numero medio di confronti tra elementi della sequenza $a_1, \ldots, a_n$. Una ragionevole stima di $S(n)$ è data dalla soluzione delle seguenti relazioni di ricorrenza:

$$\begin{aligned} S(n) &= 2 && n = 3 \\ S(n) &= n - 1 + 2S((n-1)/2) && n = 7, 15, 31, \ldots \end{aligned}$$

Anziché considerare la soluzione in ordine di grandezza, è più conveniente considerarne la soluzione esatta. È possibile dimostrare che la soluzione esatta $S(n)$ è la seguente:

$$S(n) = (n+1)\log(n+1) - 2n,$$

come si può verificare sostituendo i valori di $S(n)$ ed $S((n-1)/2)$ nelle relazioni di ricorrenza stesse.

Nella versione concorrente, si distinguono due fasi esecutive. Nella prima fase, ci sono più processi che porzioni da ordinare (e quindi molti processi saranno inattivi), mentre nella seconda fase ci sono più porzioni da ordinare che processi.

Nella tabella seguente sono riportate le iterazioni della prima fase, il numero 2^i di porzioni da ordinare ad ogni iterazione i, il numero c_i di confronti che deve effettuare ciascun processo all'iterazione i, ed il numero $2^i c_i$ di confronti effettuati complessivamente da tutti i processi. In altri termini, il valore della quarta colonna è dato dal prodotto tra i valori della seconda e della terza:

Iterazione	Porzioni	Confronti per processo	Confronti totali
0	1	$c_0 = n - 1 = n/1 - 1$	$2^0 c_0 = n - 1$
1	2	$c_1 = c_0/2 - 1 = n/2 - (1 + 1/2)$	$2^1 c_1 = n - 3$
2	4	$c_2 = c_1/2 - 1 = n/4 - (1 + 1/2 + 1/4)$	$2^2 c_2 = n - 7$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
m	$p = 2^m$	$c_m = n/p - (1 + 1/2 + 1/4 + \cdots + 1/p)$	$2^m c_m = n - p + 1$

Il tempo richiesto dalla prima fase è dato dalla somma dei valori della terza colonna della tabella:

$$T_1(n) = c_0 + c_1 + \cdots + c_m = 2(n-1)(1 - 1/p) - \log n.$$

Il numero di confronti effettuati complessivamente durante la prima fase è dato dalla somma dei valori della quarta colonna della tabella:

$$C_1(n) = 2^0 c_0 + 2^1 c_1 + \cdots + 2^{\log p} c_m = (n+1)\log p - 2(p-1).$$

Nella seconda fase, assumendo un carico uniforme tra i p processi, che sono tutti attivi, il tempo richiesto è pari al numero di confronti che ancora rimangono da essere effettuati diviso il numero di processi:

$$T_2(n) = (S(n) - C_1(n))/p.$$

Pertanto, il tempo medio richiesto dalla versione concorrente è:

$$T(n) = T_1(n) + T_2(n).$$

Lo speedup teorico risultante si ottiene dividendo il tempo della versione sequenziale per quello della versione concorrente:

$$\text{speedup} = S(n)/T(n).$$

Esempio 31.4 (Speedup del Quicksort). Per $n = 1023$ e $p = 8$, si ottiene uno speedup che non è neanche la metà del numero di processori:

$$\begin{aligned} S(n) &= 1023 \log 1024 - 2046 = 10240 - 2046 \cong 8200, \\ T_1(n) &= 2044(1 - 1/8) \cong 1776, \\ C_1(n) &= 1024(3) - 14 \cong 3058, \\ T_2(n) &= (8200 - 3058)/8 \cong 643, \\ T(n) &= T_1(n) + T_2(n) \cong 2400, \\ \text{speedup} &\cong 8200/2400 \cong 3.4. \end{aligned}$$

□

Il basso speedup è dovuto principalmente alla prima fase, dove molti processi sono inattivi perché ci sono poche porzioni da ordinare. Purtroppo, lo speedup reale è ancora più basso di quello teorico $S(n)/T(n)$, a causa dei conflitti da parte dei processi sugli accessi alla pila e ai dati $a_1, \ldots, a_n$. Per esempio, sono state effettuate delle prove sperimentali eseguendo la versione concorrente del QUICKSORT con $n = 20480$ al variare del numero di processi. Fino a 8 processi, lo speedup reale è di pochissimo inferiore allo speedup teorico (entrambi circa 3 per $p = 4$, e poco meno di 4 per $p = 8$). Già per $p = 10$, $S(n)/T(n)$ è poco più di 4, mentre lo speedup reale è circa 3, e per $p > 10$ lo speedup teorico cresce molto lentamente mentre quello reale addirittura diminuisce all'aumentare del numero di processi!

31.3 Cammini minimi

Vediamo adesso come modificare l'algoritmo di Pape-D'Esopo (cfr. § 10) per risolvere il problema dei CAMMINI MINIMI da un singolo nodo r a tutti gli altri nodi in un grafo orientato pesato di n nodi, rendendolo concorrente usando p processori.

Si ricorda che l'algoritmo di Pape-D'Esopo costruisce un albero T dei cammini minimi radicato in r effettuando una visita del grafo per mezzo di una dequeue Q, che è una combinazione di una pila e di una coda. Si parte con Q contenente solo r e con distanze d_u infinite per ogni nodo del grafo eccetto r, che ha d_r uguale a 0. Finché Q non si è svuotata, viene estratto il nodo u in testa a Q e sono verificate le condizioni di ottimalità di Bellman ($d_u + c_{uv} \geq d_v$) su ogni arco (u, v) uscente da u. Se le condizioni non sono verificate ($d_u + c_{uv} < d_v$), d_v assume il nuovo valore $d_u + c_{uv}$ e v è inserito in Q. Se v è inserito per la prima volta (cioè la sua vecchia distanza d_v da r in T era infinita), allora v viene posto in fondo a Q, mentre quando v è inserito in Q le volte successive, allora v viene posto in testa a Q.

Nella versione concorrente, si definiscono p processi. Il processo RICERCA(i) effettua ciclicamente una "ricerca", che consiste nell'estrazione di un nodo u da Q, nel

verificare le condizioni di Bellman sugli archi (u, v), e nell'aggiornare le distanze ed inserire i nodi v in Q. Per controllare la terminazione, ogni processo ha una variabile *attesa*(i). Tale variabile è posta a **true**, se il processo non ha niente da fare (perché ha trovato Q vuota), mentre è posta a **false**, altrimenti. La condizione di terminazione dell'intera procedura è data dal valore della variabile globale *stop* = *attesa*(1) **and** *attesa*(2) **and** $\cdots$ **and** *attesa*(p). In altri termini, la computazione termina quando *stop* = **true**, cioè quando tutti i processi non hanno niente da fare perché hanno tutti trovato Q vuota.

Vengono usate le primitive **lock** e **unlock** per bloccare Q durante estrazioni ed inserimenti, e per bloccare la distanza d_v all'interno di una ricerca. Il valore della variabile globale *stop* può essere modificato dal solo processo RICERCA(1) e quindi *stop* non è bloccata. Si assume che T sia stato inizializzato ad un albero "fittizio", come spiegato nel § 10, per mezzo dei processi INIZIALIZZA(i), per $1 \leq i \leq p$. Nel programma principale, Q è inizializzata in modo che contenga il nodo r, e *stop* è inizializzata a **false**.

```
process RICERCA(i);
  begin
    attesa(i) := false;
    while not stop do begin
      lock(Q);
      if Q = ∅ then begin
        attesa(i) := true;
        if i = 1 then stop := attesa(1) and attesa(2) and ··· and attesa(p);
        unlock(Q);
      end else begin
        estrai u dalla testa di Q;
        attesa(i) := false;
        unlock(Q);
        for each (u, v) ∈ A do begin
          distanza := d_u + c_uv;
          lock(d_v);
          if distanza < d_v then begin
            if d_v = +∞
              then prima := true
              else prima := false;
            d_v := distanza;
            p_v := u;
            unlock(d_v);
            lock(Q);
            if v ∉ Q then begin
              if prima
                then inserisci v in fondo a Q
                else inserisci v in testa a Q
              unlock(Q)
            end else
              unlock(d_v)
          end
        end
      end
  end;
```

```
process INIZIALIZZA(i);
    begin
        for j := i to n by p do
            if j = r
                then begin d_j := 0; p_j := 0 end
                else begin d_j := +∞; p_j := r end;
    end;

procedure CAMMINI-MINIMI(G, r);
    begin
        for all i where 1 ≤ i ≤ p do concurrently INIZIALIZZA(i);
        Q := {r};
        stop := false;
        for all i where 1 ≤ i ≤ p do concurrently RICERCA(i)
    end;
```

Prove sperimentali del precedente algoritmo concorrente, effettuate per grafi di $n = 30$ nodi con $1 \leq p \leq 8$ processi, hanno fatto registrare i seguenti speedup: 1.70 per $p = 2$; 2.1 per $p = 3$; 2.5 per $p = 4$; 3 per $p = 5$; e 3.5 per $p = 8$.

31.4 Branch-&-Bound

Un utilizzo molto importante degli algoritmi concorrenti si ha nel progetto di algoritmi branch-&-bound. Si ricorda che il branch-&-bound è una tecnica enumerativa per risolvere problemi di ottimizzazione, che elenca le soluzioni del problema effettuando delle scelte (cfr. § 23). Viene generato un albero delle scelte, tenendo conto di: (1) una regola di branch, cioè di un criterio per determinare la scelta successiva; (2) una funzione di bound, che dà una limitazione della funzione costo per tutte le soluzioni generabili a partire dalle scelte finora effettuate; (3) un criterio di visita dell'albero delle scelte (per esempio LIFO, FIFO, con priorità).

Gli algoritmi branch-&-bound, pur avendo complessità esponenziale nel caso pessimo, sono molto usati per risolvere problemi computazionalmente intrattabili di rilevante interesse pratico. Pertanto, può essere molto importante ottenere, per una versione concorrente, anche uno speedup uguale a poche unità soltanto. Per esempio, supponiamo che occorrano 7 giorni per risolvere un certo problema su input di dimensione 30 con un algoritmo branch-&-bound sequenziale (come accade in realtà per alcuni input del COMMESSO VIAGGIATORE relativi al traffico aereo). Se si ottiene uno speedup c per la versione concorrente dell'algoritmo branch-&-bound, allora lo stesso problema sugli stessi input può essere risolto impiegando soltanto $7/c$ giorni!

Consideriamo una versione concorrente di un algoritmo branch-&-bound proprio per il problema del COMMESSO VIAGGIATORE. Ricordiamo che in tale problema sono date n città e le distanze d_{ij} tra tutte le possibili coppie i e j di città distinte, e si vuole determinare un percorso ciclico che includa ogni città una e una sola volta percorrendo complessivamente la minor distanza possibile. L'input può essere visto come un grafo non orientato completo di n nodi con costi sugli archi pari alle distanze, memorizzate sotto forma di matrice, e il problema consiste nel trovare un circuito Hamiltoniano di costo minimo.

Un buon algoritmo branch-&-bound sequenziale si può basare sulle seguenti regole (cfr. Esercizio 23.3):

1. *Scelta*: inclusione/esclusione di un arco (i, j);

2. *Regola di branch*: considerare l'arco la cui esclusione massimizza il lower bound;
3. *Visita*: considera prima il nodo dell'albero delle scelte con lower bound minimo;
4. *Lower bound*: metodo di riduzione sulla matrice delle distanze.

Una buona versione concorrente considera p processi, dove ciascun processo effettua le operazioni seguenti:

a. seleziona da una lista L di nodi, condivisa tra tutti i processi, il nodo dell'albero delle scelte non ancora esaminato che ha lower bound più piccolo;
b. considera l'arco la cui esclusione massimizza il lower bound e genera i due figli del nodo dell'albero delle scelte corrispondenti ai due casi in cui l'arco selezionato è incluso/escluso dalla soluzione parziale finora costruita;
c. inserisce i due nodi generati nella lista L dei nodi non ancora esaminati e torna ad (a).

Ovviamente la lista L deve essere bloccata quando i processi vi accedono, ma il tempo per garantire tale mutua esclusione su L è comunque trascurabile rispetto al tempo richiesto da ciascun processo per effettuare la propria computazione.

Prove sperimentali del precedente algoritmo concorrente, effettuate per $n = 30$ città con $1 \leq p \leq 16$ processi, hanno fatto registrare i seguenti speedup: 1.80 per $p = 2$; 3.5 per $p = 4$; 6 per $p = 8$; ed 8 per $p = 16$.

31.5 Esercizi

Esercizio 31.1 (Prevenzione stallo). Si dimostri che non può avvenire stallo qualora tutti i processi eseguano i **lock** rispettando l'ordine alfabetico dei nomi delle variabili condivise (ad esempio ogni processo deve eseguire **lock**(x) prima di **lock**(y) poiché x precede y nell'ordine alfabetico).

Esercizio 31.2 (Rilevazione stallo). Si consideri un grafo G i cui nodi corrispondono ai processi e alle variabili condivise e dove esiste un arco $[P, x]$ se e solo se il processo P ha acquisito la variabile x oppure è in attesa della variabile x. Si dimostri che c'è una situazione di stallo se e solo se esiste un circuito nel grafo G.

Esercizio 31.3 (Sommatoria). Si scriva una procedura concorrente per calcolare la SOMMATORIA tra n numeri usando p processori.

Esercizio 31.4 (Moltiplicazione matrici). Si scriva una procedura concorrente per la moltiplicazione di due matrici $n \times n$ usando p processori.

Esercizio 31.5 (Previsita). Si scriva una procedura concorrente per effettuare la "visita anticipata" (o "previsita", cfr § 4) di un albero ordinato di n nodi usando p processori.

Esercizio 31.6 (Tappabuco). Si scriva una procedura "branch-&-bound" concorrente per il gioco del TAPPABUCO di lato n (cfr. Esercizio 23.6) usando p processori.

Capitolo 32

Algoritmi distribuiti

Nel capitolo precedente, sono stati considerati algoritmi concorrenti per un singolo calcolatore fornito di una memoria comune condivisa da più processori che eseguono processi in modo asincrono. In questo capitolo, invece, si considerano algoritmi distribuiti per reti di calcolatori, in cui ci sono più calcolatori distinti connessi in rete (geograficamente anche molto distanti tra loro), ognuno provvisto della propria memoria, che cooperano in modo asincrono per la risoluzione di un problema computazionale.

Come nel caso concorrente, ogni calcolatore esegue un processo sequenziale, che può essere uguale o diverso dai processi eseguiti dagli altri calcolatori. A differenza del caso concorrente, però, non c'è alcuna memoria comune, non c'è un "programma principale" che controlla la computazione dell'algoritmo eseguito dai processi, e i dati del problema da risolvere sono partizionati tra i calcolatori.

Non essendoci alcuna memoria comune, lo scambio di dati tra processi non è controllabile con primitive di sincronizzazione. Invece, ogni calcolatore può comunicare con altri calcolatori inviando e ricevendo messaggi lungo canali di comunicazione che dipendono dalla topologia della rete. Il modello di calcolo può essere visto come un grafo, dove i nodi sono i calcolatori e gli archi sono collegamenti tra coppie di calcolatori. Ogni calcolatore può inviare messaggi solo ai "vicini", cioè con i calcolatori con i quali è collegato tramite archi. Ovviamente, per comunicare con calcolatori non vicini, i messaggi dovranno essere ritrasmessi dai calcolatori intermedi lungo un cammino tra il calcolatore mittente e quello destinatario.

I messaggi sono diversi per algoritmi diversi. Gli unici eventi rilevanti per un processo sono, oltre a quelli dell'algoritmo che il processo stesso sta eseguendo, l'invio e la ricezione di un messaggio. Non c'è uno stato globale della rete visibile ad un osservatore esterno, pertanto gli eventi riguardanti i messaggi sono ordinabili solo parzialmente (se si vuole un ordinamento totale si possono usare delle marche temporali, dette "timestamp"). La ricezione di un messaggio segue l'invio dello stesso messaggio e, per ogni coppia di processi, i messaggi sono ricevuti nello stesso ordine in cui sono stati spediti. Ogni messaggio inviato viene ricevuto dopo un tempo ignoto a priori ma finito, ed i messaggi ricevuti da un processo sono accodati ed elaborati in ordine FIFO. Non c'è perdita, alterazione, duplicazione o scambio di messaggi. La misura di complessità più usata per un algoritmo distribuito è il numero totale di messaggi. Si preferiscono pertanto algoritmi che richiedono meno messaggi, poiché un numero elevato di messaggi tende ad intasare la rete e quindi ad aumentare considerevolmente i tempi di computazione.

Non essendoci un "programma principale" che controlla la computazione dell'algoritmo eseguito dai processi, non esiste in generale alcuna gerarchia tra i processi. Pertanto, una decisione comune va presa ottenendo un consenso tra tutti i processi. Se si vuole un processo "coordinatore" (o "leader"), allora ne va designato uno eseguendo un algoritmo apposito.

Essendo i dati partizionati[1] tra i vari calcolatori, ogni processo non conosce né lo stato globale della computazione nella rete, né la totalità dei dati. In altri termini, nel risolvere un problema, ogni processo ha una conoscenza solo parziale dei dati e della computazione, di solito relativa soltanto a se stesso ed al più ai processi in esecuzione sui calcolatori "vicini". Un singolo processo non conosce neanche la topologia della rete né il numero totale di processi!

La struttura di controllo di ciascun processo è "guidata da eventi", ovvero permette di attendere il verificarsi di un particolare evento tra un insieme di possibilità e di effettuare conseguentemente azioni opportune. Si assuma che in corrispondenza di tre condizioni $C1$, $C2$ e $C3$, occorra effettuare le azioni $A1$, $A2$ e $A3$, rispettivamente. Si utilizza allora il costrutto seguente:

when $C1$ **do** $A1$;
when $C2$ **do** $A2$;
when $C3$ **do** $A3$;

In questo modo, al verificarsi dell'evento $C2$, che, per esempio, consiste nell'estrazione di un certo tipo di messaggio dalla coda dei messaggi in ingresso, verrà eseguita l'istruzione $A2$, dopodiché il processo si porrà in attesa del verificarsi di un altro evento.

Le caratteristiche peculiari dei modelli distribuiti (processi asincroni, controllo guidato da eventi, mancanza di ordinamento globale tra eventi, partizionamento dei dati, scambio di messaggi tra processi, ecc.) rendono piuttosto complicato il progetto di algoritmi distribuiti. In particolare, occorre prestare attenzione nel dimostrare che gli algoritmi termineranno correttamente dopo un tempo finito, evitando situazioni di stallo in cui si ha un circuito di processi ognuno in attesa del successivo. Comunque, si possono individuare tre tecniche principali di progetto di algoritmi distribuiti:

1. *Diffusione* Un particolare processo inizialmente può solo spedire messaggi, mentre gli altri processi possono spedire un messaggio solo dopo averne ricevuto uno (come avviene per trasmettere un dato dalla radice a tutte le foglie di un albero binario).
2. *Token* Un "gettone" (o "token") rappresenta un privilegio che circola nella rete, e il processo che riceve il token effettua l'elaborazione e poi passa il token ad un altro processo (come accade quando viene passato un microfono tra i relatori di una conferenza).
3. *Timestamping* Viene mantenuto un clock globale logico per mezzo di "marche temporali" che permettono di ordinare totalmente gli eventi (per esempio per evitare la "starvation", cioè che un processo, pur non essendo in stallo, "muoia di fame" poiché non arriva mai il suo turno per "mangiare").

Esempio 32.1 (Clock globale). Vediamo come definire un clock logico globale con la tecnica del timestamping. Ogni processo P_i ha una variabile locale intera h_i, inizializzata a zero. Ogni messaggio m spedito da P_i è marcato con il valore di h_i e col nome i del processo, ed ha la forma (m, h_i, i). Il processo P_i elabora h_i come segue:

[1] Talvolta i dati sono replicati tra i vari processi, per esempio per motivi di affidabilità. In tal caso, ci sono più copie $x_1, x_2, \ldots, x_k$ dello stesso dato x, una per processo, ma sorge allora il problema di mantenere tali copie "consistenti", in modo che ad ogni istante valga $x_1 = x_2 = \cdots = x_k$. In questo capitolo la replicazione dei dati non è considerata.

if received (m, h_j, j) **then** $h_i := \max(h_i, h_j) + 1$;

if devi inviare m **then**
begin
$h_i := h_i + 1$;
send(m, h_i, i)
end;

In questo modo, gli h_i sono sempre crescenti per ciascun i, e l'invio di un messaggio m ha un tempo più piccolo di quello della ricezione dello stesso messaggio. L'ordinamento che si ottiene tra gli eventi è totale, perché l'invio di (m, h_i, i) precede quello di (m', h_j, j) se $h_i < h_j$ oppure se $h_i = h_j$ e $i < j$. □

Nel resto di questo capitolo, dopo aver definito i grafi che rappresentano le principali topologie delle reti di calcolatori (come "anelli", "stelle", "grafi completi", ecc.), saranno presentati algoritmi distribuiti per la MUTUA ESCLUSIONE (in un grafo completo), l'ELEZIONE DEL LEADER (in un anello), e il MINIMO ALBERO DI COPERTURA (in un grafo arbitrario).

32.1 Topologie di reti

Per descrivere la topologia di una rete di calcolatori, sono considerati principalmente i seguenti grafi (di solito non orientati, con una rilevante eccezione): anello, stella, albero, grafo, grafo completo.

In un anello bidirezionale, gli n calcolatori sono disposti in un singolo "circuito", così che ogni processo può comunicare direttamente con entrambi i vicini (detti comunemente il vicino di destra e il vicino di sinistra). In un anello unidirezionale, invece, gli n calcolatori sono disposti in un singolo "ciclo", così che ogni processo può trasmettere direttamente ad un vicino (per esempio quello di destra, assumendo che il ciclo sia orientato secondo il verso orario) e ricevere dall'altro vicino (quello di sinistra). La stella è un caso speciale di albero (cfr. § 9), dove un particolare calcolatore è connesso con tutti i rimanenti $n - 1$, mentre ciascuno di questi è connesso con uno soltanto. Infine, oltre ad un usuale albero o grafo non orientato, è usato talvolta il grafo completo, dove ogni calcolatore è connesso con tutti i rimanenti $n - 1$.

Esempio 32.2 (Topologie di reti). La Fig. 32.1 mostra esempi di anello (bidirezionale e unidirezionale), stella, albero, grafo, e grafo completo, per reti di $n = 8$ calcolatori ciascuno. □

32.2 Mutua esclusione

Nel problema della MUTUA ESCLUSIONE, occorre assegnare equamente e temporaneamente un privilegio ad un processo (per esempio per accedere ad una "sezione critica" e poter modificare una variabile comune: cfr § 31). In altri termini, il privilegio va assegnato in modo che nessun particolare processo sia favorito e che il processo che ottiene il privilegio non lo mantenga per sempre.

In alcune topologie, il problema della mutua esclusione è risolvibile banalmente. Per esempio, in un anello unidirezionale basta far circolare un token:

wait(*token*) **from** vicino di sinistra;
··· "sezione critica" ···
send(*token*) **to** vicino di destra;

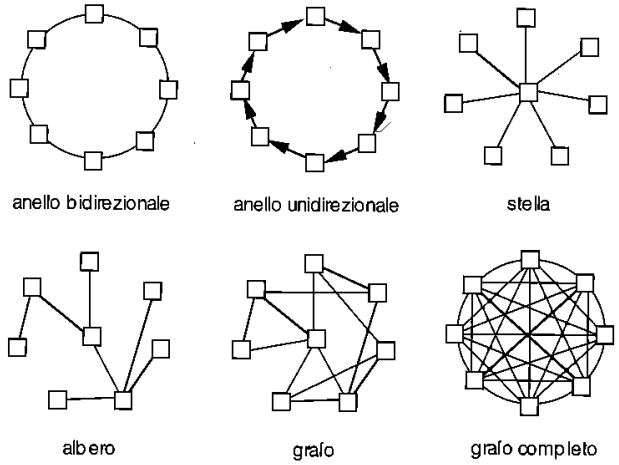


Figura 32.1: Esempi di topologie per reti di $n = 8$ calcolatori.

In un grafo completo con n processi, invece, è facile risolvere il problema con $3(n - 1)$ messaggi. Infatti, con $n - 1$ messaggi un processo può comunicare agli altri la sua intenzione di entrare in una sezione critica, con ulteriori $n - 1$ messaggi tutti gli altri processi gli possono dare il permesso, e con $n - 1$ messaggi finali il processo può informare gli altri che è uscito dalla sezione critica.

Vediamo invece un algoritmo più efficiente, inventato da Ricart e Agrawala, (1981, 1983) che usa soltanto n messaggi per un grafo completo. L'algoritmo richiede $n - 1$ messaggi perché un processo P_i che vuole entrare in una sezione critica informi tutti gli altri processi della sua intenzione, ed un solo altro messaggio perché P_i sia abilitato all'accesso. In particolare, non sono spediti messaggi per informare i processi quando P_i esce dalla sezione critica!

Viene utilizzato un "token", che inizialmente è assegnato ad un processo arbitrario, e solo chi possiede il token può accedere alla sezione critica. Non si sa chi possiede il token, ed un processo che intende accedere alla sezione critica manda una richiesta del token a tutti gli altri processi per mezzo di un messaggio che è marcato con un timestamp e distribuito a tutti, per un totale di $n - 1$ messaggi.

Il token consiste in un vettore di timestamp, dove l'i-esimo elemento *token*[i] del vettore registra quando il token è stato assegnato per l'ultima volta al processo P_i. Viene utilizzato un secondo vettore, locale a ciascun processo, dove l'i-esimo elemento *richiesta*[i] del vettore registra quando il token è stato richiesto per l'ultima volta dal processo P_i. Tutti i vettori sono inizializzati a zero. Si assuma che il processo P_i detenga il token. Quando esce dalla sezione critica, P_i scandisce circolarmente il suo vettore locale delle richieste nell'ordine $i + 1, i + 2, \ldots, n, 1, 2, \ldots, i - 1$. La scansione si arresta sul primo indice j tale che *richiesta*[j] > *token*[j] (cioè il timestamp della richiesta di P_j è maggiore del timestamp dell'ultimo assegnamento a P_j), e P_i trasmette il token a P_j.

Nel codice seguente la variabile locale *token-presente* è assunta inizializzata a

false per ogni processo, tranne quello che detiene inizialmente il token. La primitiva **broadcast** (*richiesta*, *orologio*, *i*) trasmette la richiesta del token da parte di P_i a tutti gli altri processi, specificando il timestamp con la variabile *orologio*. La primitiva **wait**(*accesso*, *token*) mette il processo P_i in attesa di un messaggio di tipo "accesso", che consiste nella ricezione del token, che è poi copiato nell'array *token*[1..*n*]. È usata una variabile booleana *token-in-uso*, posta a **true** durante l'accesso alla sezione critica. L'algoritmo consta di due parti, la prima riguarda l'accesso alla sezione critica, mentre la seconda tratta l'elaborazione dei messaggi di richiesta del token.

```
if not token-presente then
    begin
        orologio := orologio + 1;
        broadcast (richiesta, orologio, i);
        wait (accesso, token);
        token-presente := true;
    end;
token-in-uso := true;
··· "sezione critica" ···
token[i] := orologio;
token-in-uso := false;
for j := i mod n + 1 to (i − 2) mod n + 1 do
    if richiesta[j] > token[j] and token-presente then
        begin
            token-presente := false;
            send (accesso, token) to j;
        end;

when received (richiesta, k, j) do
    begin
        richiesta[j] := max{richiesta[j], k};
        if token-presente and not token-in-uso then
            begin
                token[i] := orologio;
                for j := i mod n + 1 to (i − 2) mod n + 1 do
                    if richiesta[j] > token[j] and token-presente then
                        begin
                            token-presente := false;
                            send (accesso, token) to j;
                        end
            end
    end;
```

La mutua esclusione è garantita dal fatto che ad ogni istante esiste al più una variabile *token-presente* che vale **true**, perché si cambia il valore da **false** a **true** solo alla ricezione del token, che è unico, e si ricambia da **true** a **false** prima di riinviare il token. L'assenza di stallo si prova facilmente per assurdo, poiché se tutti i processi fossero bloccati, allora il token sarebbe in transito, ed alla ricezione del token un processo ripartirebbe. Infine, l'equità deriva dalla tecnica di scansione ciclica del vettore delle richieste, che garantisce che ogni richiesta verrà soddisfatta. Sono utilizzati $n - 1$ messaggi "corti" (di un solo timestamp) per diffondere la richiesta, ed un solo messaggio "lungo" (di $n - 1$ timestamp) per passare il token, per un totale di n messaggi.

32.3 Elezione del leader

Nel problema dell'ELEZIONE DEL LEADER, occorre assegnare permanentemente un privilegio ad un processo (per esempio per coordinare le attività dei rimanenti processi, oppure per "rigenerare" un token che è andato perduto). Un qualsiasi algoritmo distribuito dove il controllo passa attraverso il leader è di fatto centralizzato, ed un guasto al calcolatore che esegue il processo leader fa fallire l'intera computazione. Quando un tale guasto è rilevato, comunque, i restanti processi si possono accordare per eleggere un nuovo leader.

Consideriamo due algoritmi distribuiti per eleggere un leader in una rete ad anello unidirezionale e bidirezionale, rispettivamente, in cui ciascun processo P_i non conosce né l'identità degli altri processi né il numero totale n dei processi. Entrambi gli algoritmi consistono nel trovare il MASSIMO tra gli identificatori dei processi, che ovviamente devono essere tutti distinti, e assumono soltanto che ogni processo P_i conosca il proprio identificatore i.

Vediamo dapprima l'algoritmo per l'anello unidirezionale, ideato da Chang e Roberts (1979). L'elezione inizia con un qualsiasi processo che si marca "partecipante" e trasmette il proprio identificatore al suo vicino di destra. Quando un processo P_i riceve un identificatore j, confronta i con j e ritrasmette a destra il massimo tra i e j. Se P_i non è marcato e riceve $j < i$, allora si marca anch'esso "partecipante". Quando un processo marcato P_i riceve il proprio identificatore i, esso sa di essere il massimo e quindi si proclama leader. Se è necessario che gli altri processi conoscano la sua identità, il leader può trasmettere il proprio identificatore lungo l'anello.

Nel codice seguente si usano due messaggi: *elezione*, per trasmettere l'identificatore durante l'elezione, ed *eletto*, per far conoscere l'identità del leader. Ogni processo ha il proprio identificatore memorizzato nella variabile *mio-numero* e la marca nella variabile booleana *partecipante*, inizializzata a **false**.

```
when voglio indire l'elezione do
    begin
        partecipante := true;
        send-right (elezione, mio-numero);
    end;

when received (elezione, j) do
    begin
        if j > mio-numero then
            begin
                send-right (elezione, j);
                partecipante := false
            end;
        if j < mio-numero and not partecipante then
            begin
                send-right (elezione, mio-numero);
                partecipante := true
            end;
        if j = mio-numero then
            send-right (eletto, mio-numero);
    end;
```

```
when received (eletto, j) do
    begin
        leader := j;
        partecipante := false;
        if j ≠ mio-numero then
            send-right (eletto, j);
    end;
```

La dimostrazione di correttezza è banale qualora un solo processo indica l'elezione. Comunque, l'algoritmo è corretto anche se più processi indicono più elezioni simultaneamente. Se un processo partecipante ad un'elezione riceve dal vicino un identificatore minore del proprio (nel caso **if** $j <$ *mio-numero* **and** *partecipante*), è applicato il cosiddetto principio di estinzione selettiva ed il messaggio ricevuto viene ignorato e non ritrasmesso.

Il tempo richiesto dall'algoritmo è $O(n)$, assumendo che ogni messaggio giunga ad un vicino dopo $O(1)$ tempo. Infatti, ci sono due casi estremi. Se tutti i processi indicono un'elezione contemporaneamente, allora il massimo riceverà il proprio identificatore dopo un giro nell'anello. Se invece l'elezione è indetta soltanto dal processo a destra del massimo, allora occorreranno 2 giri dell'anello. In entrambi i casi, il tempo è quindi $O(n)$.

Il numero di messaggi, invece, varia a seconda dei casi. Nel caso migliore, i processi sono ordinati per identificativi crescenti e l'elezione è indetta dal massimo. In tal caso, il numero di messaggi è $O(n)$. Nel caso peggiore, i processi sono ordinati per identificativi decrescenti e tutti i processi indicono un'elezione contemporaneamente. Siccome il messaggio di elezione indetta da P_i è trasferito i volte prima di essere estinto, il numero totale di messaggi è $O(\sum_{i=1}^{n} i)$, che cresce come $O(n^2)$.

Nel caso medio, assumendo che gli identificatori dei processi siano uniformemente distribuiti lungo l'anello, è possibile dimostrare che il numero totale di messaggi richiesto dall'algoritmo di Chang e Roberts è $O(n \log n)$. Esiste comunque un altro algoritmo, ideato da Dolev, Klawe e Rodeh (1982), che richiede $O(n \log n)$ messaggi nel caso pessimo per eleggere il leader in un anello unidirezionale.

Vediamo adesso l'algoritmo per eleggere il leader in un anello bidirezionale, dovuto a Hirschberg e Sinclair (1980), che richiede $O(n \log n)$ messaggi nel caso pessimo. Stavolta i messaggi di candidatura viaggiano in entrambe le direzioni fino ad allontanarsi per una certa distanza prefissata (in modo che il mittente si aspetta come risultato due risposte affermative alla propria candidatura). Tali messaggi sono eventualmente ritrasmessi ripetutamente, raddoppiando ogni volta la distanza che devono percorrere.

L'elezione inizia con un qualsiasi processo P_j che si marca "candidato", e procede in "fasi" $0, 1, 2, \ldots$. Nella fase h, P_j trasmette j ai suoi due vicini, con l'intenzione che ciascun messaggio giunga fino a distanza 2^h da P_j e poi torni indietro una risposta affermativa. Quando un processo P_i riceve un identificatore j, confronta j con il proprio identificatore i. Se i è maggiore di j, allora P_i sostituisce P_j come "candidato", altrimenti P_j resta "candidato" e P_i è "sconfitto". Se P_i non riceve entrambe le risposte affermative alla propria candidatura, allora si dichiara "sconfitto", altrimenti consulta un numero doppio di processi. Un processo sconfitto si limita a ritrasmettere i messaggi che riceve da un vicino all'altro vicino. Se P_j riceve il messaggio di candidatura col proprio identificatore, allora si proclama leader. Ogni processo ha una variabile *stato* che registra lo stato del processo riguardo all'elezione, che può assumere i valori "non-coinvolto", "candidato", "sconfitto", "eletto". Sono usate tre primitive di comunicazione:

1. **send-left-right**, per trasmettere un messaggio ad entrambi i vicini,
2. **pass**, per passare il messaggio ricevuto da un vicino all'altro vicino,
3. **respond**, per rispondere al vicino che ha trasmesso il messaggio.

I messaggi sono di due tipi. Quelli del primo tipo servono per dichiarare una candidatura, utilizzano sia **send-left-right** che **pass**, ed hanno la forma:

(*candidatura*, *numero*, *lung*, *maxlung*)

dove *numero* è il nome del processo candidato che spedisce il messaggio, *lung* è la distanza che il messaggio ha già percorso nell'anello, e *maxlung* è la massima distanza che il messaggio deve percorrere.

I messaggi del secondo tipo servono per rispondere ad una dichiarazione di candidatura, utilizzano sia **pass** che **respond**, ed hanno la forma:

(*risposta*, *esito*, *numero*)

dove *esito* vale **true** se la risposta è favorevole, **false** altrimenti, e *numero* è il nome del processo destinatario del messaggio.

La primitiva **wait** è usata per attendere entrambi i messaggi di risposta. Il numero delle risposte è accumulato in *numrisp*, mentre l'**and** dei due valori di *esito* è accumulato nella variabile booleana *ok*.

```
when received (risposta, esito, numero) do
    begin
        if numero = mio-numero then begin
            numrisp := numrisp + 1;
            ok := ok and esito
        end else
            pass (risposta, esito, numero);
    end;

when received (candidatura, numero, lung, maxlung) do
    begin
        if numero < mio-numero then begin
            respond (risposta, false, numero);
            if stato = "non-coinvolto"
                then voglio indire l'elezione;
        end;
        if numero > mio-numero then begin
            stato := "sconfitto";
            lung := lung + 1;
            if lung < maxlung
                then pass (candidatura, numero, lung, maxlung)
                else respond (risposta, true, numero);
        end;
        if numero = mio-numero then
            if stato ≠ "eletto" then begin
                stato := "eletto";
                vincitore := mio-numero;
                pass (eletto, mio-numero)
            end
    end;
```

```
when received (eletto, numero) do
    begin
        if vincitore ≠ numero then begin
            pass (eletto, numero);
            vincitore := numero;
            stato := "non-coinvolto"
        end
    end;

when voglio indire l'elezione do
    begin
        stato := "candidato"; maxlung := 1;
        while stato = "candidato" do
            begin
                numrisp := 0;
                ok := true;
                send-left-right (candidatura, mio-numero, 0, maxlung);
                wait (numrisp = 2);
                if ok
                    then maxlung := maxlung * 2
                    else stato := "sconfitto"
            end
    end;
```

Valutiamo il numero di messaggi richiesti dall'algoritmo. Nella generica fase h, la lunghezza della catena attraversata da un messaggio di candidatura è 2^h. Per $h > 0$, un processo lancia la sua candidatura su una catena di lunghezza raddoppiata (pari a $2 * 2^{h-1}$) solo se non è stato sconfitto nella consultazione precedente da un processo che dista da lui al più 2^{h-1} in qualsiasi delle due direzioni dell'anello. Ciò implica che, in un arbitrario gruppo di $2^{h-1} + 1$ processi consecutivi, al più un solo processo può lanciare la propria candidatura nella fase h. Pertanto, nel caso pessimo, nella fase h ci sono al più $\lfloor n/(2^{h-1} + 1) \rfloor$ candidati con catene di lunghezza 2^h. Per ogni candidato ci sono 2 messaggi (candidatura e risposta) in ogni direzione, per un totale di 4 messaggi, lungo le catene di lunghezza appropriata. Pertanto, il numero totale di messaggi della fase h è limitato da $4 \cdot 2^h \lfloor n/(2^{h-1} + 1) \rfloor \leq 8n$. Ma il numero totale di fasi che sono eseguite prima che venga eletto il leader è $1 + \lceil \log n \rceil$, e quindi il numero di messaggi è $O(n \log n)$.

32.4 Minimo albero di copertura

Progettiamo adesso un algoritmo distribuito efficiente per trovare il MINIMO ALBERO DI COPERTURA in un grafo generico connesso, dove ciascun nodo (processo) è a conoscenza della propria identità, che è distinta da quella degli altri nodi, e del peso degli archi (collegamenti) ad esso incidenti.

Ricordiamo che, dato un grafo non orientato pesato $G = (N, A)$, con n nodi ed m archi, si vuole trovare un albero $T = (N, A')$, dove A' è un sottoinsieme di $n - 1$ archi di A, tale che la somma dei pesi degli archi in A' sia minima [cfr § 13].

Il metodo generale di risoluzione consiste nel mantenere una foresta di copertura di frammenti, che sono componenti connesse (cioè sottoalberi) dell'albero minimo.

Si parte con una foresta di n frammenti, ciascuno contenente un solo nodo e nessun arco e si uniscono ripetutamente coppie di frammenti per mezzo dell'arco di peso minimo che li collega, finché non si ottiene un solo frammento, che è il minimo albero di copertura. L'ordine con il quale sono effettuate le unioni tra due frammenti non è rilevante ai fini della correttezza, purché sia sempre scelto il "miglior arco uscente" per ogni frammento F, cioè l'arco di peso minimo che connette un nodo del frammento F con un nodo esterno al frammento. Infatti è facile provare che, se F è un frammento del minimo albero di copertura ed a è il miglior arco uscente di F, allora anche $F \cup \{a\}$ è un frammento del minimo albero di copertura. In particolare, entrambi gli algoritmi sequenziali di Kruskal e di Prim sono basati su questa strategia. Infatti, l'algoritmo di Kruskal [cfr. § 13.1] aggiunge ripetutamente alla foresta di copertura l'arco di peso minimo tra tutti quelli che connettono coppie di frammenti distinti, mentre l'algoritmo di Prim [cfr. Esercizio 13.4] si concentra sempre sullo stesso frammento F e lo "ingrossa" ripetutamente aggiungendovi il miglior arco uscente di F.

Per adottare il precedente metodo generale di risoluzione in ambito distribuito, bisogna che sia possibile estendere la foresta di copertura con più archi trovati in parallelo. Occorre quindi che più frammenti determinino il proprio minimo arco uscente contemporaneamente e che siano eseguite più unioni simultanee di coppie di frammenti. Se il minimo arco uscente da ogni frammento non è unico, però, può accadere che l'unione simultanea di più frammenti provochi la formazione di un circuito. Per evitare tale inconveniente, si assume che i pesi degli archi siano tutti distinti. In questo modo, il minimo arco uscente da ogni frammento è unico, e quindi anche il minimo albero di copertura è unico. Se i pesi non sono distinti, è sufficiente concatenare ad ogni peso $c[a]$ di un arco $a = [u, v]$ gli identificatori u e v dei due nodi dell'arco, con l'identificatore minore che precede il maggiore, ottenendo così pesi $c'[a]$ tutti distinti senza alterare la soluzione del problema.

Un secondo inconveniente che può sorgere in ambito distribuito è che, nel caso pessimo, un frammento grande si può ripetutamente fondere con uno piccolo, aumentando considerevolmente il numero totale di messaggi (p.e, un frammento di $n/2$ nodi si può fondere $O(n)$ volte con un frammento di 1 nodo, per un totale di $O(n^2)$ messaggi). Per rimediare a questo inconveniente, viene usata la nozione di "livello", come stima della dimensione di un frammento F. Si assuma che ogni nodo (processo) sia a conoscenza della propria identità, che è distinta da quella degli altri nodi, e soltanto dei pesi degli archi in esso incidenti (e quindi non conosce né il numero n di nodi, né l'identità dei vicini, né altre proprietà del grafo). Uno schema generale di algoritmo distribuito per il minimo albero di copertura è il seguente.

I frammenti del minimo albero di copertura crescono "per livelli". Ciascun frammento di livello L ha almeno 2^L nodi. Inizialmente, ci sono n frammenti di livello 0. Per ogni frammento F, ciascun nodo u di F conosce sia l'identità del frammento sia quali archi incidenti in u sono o non sono in F. Per formare frammenti di livello $L + 1$, ciascun frammento di livello L effettua una ricerca del minimo arco uscente. Su richiesta della radice, ciascun nodo del frammento cerca, tra gli archi in esso incidenti, l'arco di peso minimo uscente da F (se esiste), chiedendo all'altro nodo dell'arco se appartiene allo stesso frammento oppure no. Successivamente, tutte le foglie di F spediscono questa informazione verso i nodi interni, che ritrasmettono durante il tragitto sempre il minimo. In questo modo, il minimo ottenuto dalla radice è proprio il minimo arco uscente dal frammento. Per unire due frammenti di livello L in un unico frammento di livello $L + 1$, la radice di ciascun frammento comunica coi nodi incidenti sul minimo arco uscente perché marchino l'arco come appartenente al

nuovo frammento. Poiché i pesi degli archi sono tutti distinti, è facile verificare che esiste un unico arco a che è il minimo arco uscente comune a due frammenti. Successivamente, è stabilita l'identità del nuovo frammento (per esempio, scegliendo come identificativo del nuovo frammento proprio l'arco a e cambiando conseguentemente la radice). Poiché i due frammenti di livello L avevano ciascuno almeno 2^L nodi, il nuovo frammento risultante di livello $L+1$ ha almeno 2^{L+1} nodi.

Essendo il modello asincrono, non vi è alcuna garanzia che nello schema precedente tutti i frammenti crescano in modo "bilanciato". In altri termini, nello stesso istante, alcuni frammenti possono avere un livello più piccolo (e quindi meno nodi) ed altri un livello più grande (e quindi più nodi). Per abbassare la complessità, qualora si cerchi di unire due frammenti di livello diverso, si impone che il frammento di livello più piccolo sia "assorbito" da quello di livello più grande, ereditandone sia identità che livello. In questo modo, l'intera operazione può essere pensata come un aggiornamento del frammento di livello più grande già esistente piuttosto che una creazione di un nuovo frammento e quindi occorre fare modifiche solo nel frammento più piccolo. Indicato con $L(F)$ il livello L del frammento F, si ha così:

$L(F) = 0$, se F include un solo nodo,

$L(F) = L(F) + 1$, se F si fonde con F' tale che $L(F) = L(F')$,

$L(F) = L(F')$, se F è assorbito da F' tale che $L(F) < L(F')$.

L'assorbimento presenta un ulteriore vantaggio. Infatti se F scopre che il suo minimo arco uscente raggiunge F', ed $L(F) < L(F')$, allora F può essere direttamente assorbito da F' senza attendere informazioni sul minimo arco uscente da F'. Nel caso che $L(F) = L(F')$, invece, poiché è creato un nuovo frammento, prima di fondere occorre attendere che sia stato individuato il minimo arco uscente comune sia di F che di F'.

Si noti che sorge una difficoltà quando un nodo u chiede ad un altro nodo v se è nello stesso frammento oppure no. Infatti, essendo il modello asincrono, v potrebbe essere in realtà nello stesso frammento di u, ma ignorarlo perché non ha ancora ricevuto il messaggio con l'aggiornamento dell'identità del frammento. Questa difficoltà è superata confrontando i livelli dei frammenti di u e v. Se il livello di v è inferiore a quello di u, allora la risposta è ritardata finché il livello di v non diventa maggiore o uguale a quello di u.

Le considerazioni precedenti sono alla base dell'algoritmo di Gallager, Humblet e Spira (1983), che presentiamo ora in dettaglio. In questo algoritmo, un frammento F è univocamente determinato dal peso di un suo arco, detto nucleo. Ciascun nodo di F memorizza il livello, il peso del nucleo, e l'arco che lo connette al padre nel frammento. Ciascuna sequenza di padri porta ad uno dei due nodi estremi del nucleo, ciascuno dei quali ha come padre l'altro. Per comodità, entrambi questi nodi sono i coordinatori di F. Si usano i seguenti sette tipi di messaggi:

- *inizia*, che è diffuso dai coordinatori a tutti i nodi del frammento sia per iniziare la ricerca del minimo arco uscente dal nuovo frammento che per diffondere l'identità (peso del nucleo) ed il livello del nuovo frammento;
- *verifica*, che un nodo, mentre cerca il suo minimo arco uscente, invia ad un vicino per sapere se fa parte dello stesso frammento oppure no;
- *accetta* o *rifiuta*, che sono spediti in risposta ad un messaggio di verifica per negare o affermare, rispettivamente, l'appartenenza allo stesso frammento;

- *minimo*, che è diretto dai nodi che stanno al bordo di un frammento verso i coordinatori per calcolare il minimo degli archi uscenti trovati dai singoli nodi del frammento;
- *cambia*, che è spedito dai coordinatori di un frammento verso il nodo che ha trovato il minimo arco uscente affinché sia tentata l'unione con il frammento all'altro estremo del minimo arco uscente;
- *connetti*, che è spedito lungo il minimo arco uscente di un frammento F quando questo tenta di unirsi col frammento all'estremo opposto; il risultato è una fusione, qualora due messaggi *connetti* siano stati spediti in entrambi i versi dello stesso arco, oppure un assorbimento, qualora un solo messaggio *connetti* sia stato spedito sull'arco ed il livello del destinatario sia maggiore del livello del mittente.

Il processo di ciascun nodo u utilizza le seguenti variabili:

- *statonodo*, lo stato del nodo u, che può essere "addormentato", se il nodo non ha ancora ricevuto alcun messaggio, "trova", se è in cerca del minimo arco uscente da F, e "trovato", altrimenti;
- *statoarco*[a], lo stato dell'arco a incidente in u, che può essere "ramo", se appartiene al minimo albero di copertura, "rifiutato", se non appartiene al minimo albero di copertura, e "base", altrimenti (inizialmente *statoarco*[a] = "base"; durante l'esecuzione dell'algoritmo può temporaneamente accadere che lo stato dello stesso arco $a = [u, v]$ sia diverso per i nodi u e v);
- *mpu*, il peso del minimo arco uscente nel sottoalbero di F radicato in u (che può assumere anche il particolare valore ∞);
- *migliore*, l'arco incidente in u dal quale proviene *mpu* (col particolare valore λ per indicarne la non esistenza);
- *padre*, l'arco incidente in u che lo connette al padre nel frammento F;
- *candidato*, l'arco incidente in u dal quale u si aspetta risposta ad un messaggio di *verifica*;
- F ed L, l'identificativo ed il livello, rispettivamente, del frammento al quale appartiene u;
- *conta-trova*, che conta quante risposte mancano a messaggi di *inizio* con parametro "trova" inviati lungo archi incidenti in u per la ricerca del minimo arco uscente.

Inoltre, il processo del nodo u utilizza la sua coda dei messaggi in ingresso per "ritardare" la risposta ad un messaggio, semplicemente reinserendo il messaggio in fondo alla coda. Vediamo il codice del processo relativo al generico nodo u, che appartiene al frammento F di livello L. In genere, l'arco attraverso il quale u comunica è $a = [u, v]$ con v appartenente al frammento F' di livello L' (tranne per la procedura SVEGLIA). La computazione inizia quando un qualsiasi processo u si sveglia spontaneamente oppure perché ha ricevuto un messaggio da un altro processo.

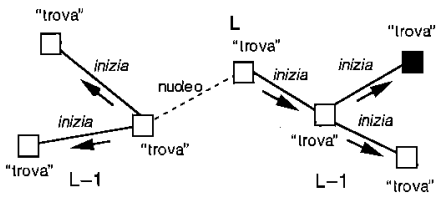


Figura 32.2: Diffusione dei messaggi *inizia*, a partire dai coordinatori, dopo una fusione (il nodo nero indica il processo che troverà il minimo arco uscente dal frammento).

when ti sei svegliato **do** SVEGLIA;

procedure SVEGLIA;
  begin
    e := arco $[u, v]$ incidente in u con peso $c[e]$ minimo;
    statoarco$[e]$:= "ramo";
    $L := 0$;
    statonodo := "trovato";
    conta-trova := 0;
    send (*connetti*, L) **on** e;
  end;

I messaggi *connetti* lanciano l'attività di fusione o assorbimento dei due frammenti F ed F'. Se $L' < L$, allora il nodo u ritrasmette il proprio valore di *statonodo* (che può essere "trova" oppure "trovato") lungo l'arco $a = [u, v]$ per non fare attendere inutilmente il frammento più piccolo. Infatti, se u non ha ancora inviato il messaggio di *minimo* nel suo frammento F (*statonodo* = "trova"), allora anche v può collaborare per cercare il minimo arco uscente del frammento F ingrandito con l'assorbimento di F' da parte di F. Se invece u ha già inviato il messaggio di minimo su F (*statonodo* = "trovato"), allora il minimo arco uscente dal nodo u ha sicuramente un peso minore del minimo arco uscente dal frammento F', ed è quindi inutile che v collabori per cercare il minimo arco uscente di F.

when received (*connetti*, L') **on** a **do**
  begin
    if *statonodo* = "addormentato" **then** SVEGLIA;
    if $L' < L$ **then begin**
      statoarco$[a]$:= "ramo";
      send (*inizia*, L, F, *statonodo*) **on** a;
      if *statonodo* = "trova"**then** *conta-trova* := *conta-trova* + 1
    end else
      if *statoarco*$[a]$:= "base"
        then reinserisci il messaggio in fondo alla coda
        else send (*inizia*, $L + 1$, $c[a]$, "trova") **on** a;
  end;

Come mostrato nella Fig. 32.2, i messaggi *inizia* sono trasmessi in origine dai coordinatori e distribuiti all'interno del frammento, dopo una fusione o un assorbimento, insieme con il livello e l'identità del nuovo frammento (L ed F, nel caso di

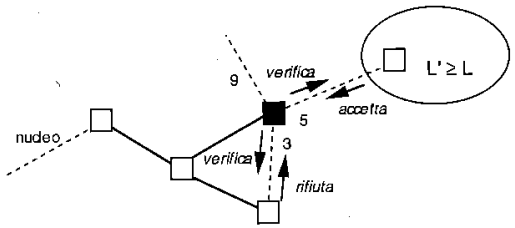


Figura 32.3: Verifica sugli archi incidenti in un nodo per ricercare il minimo arco uscente.

assorbimento, oppure $L + 1$ e $c[a]$, nel caso di fusione) ed eventualmente con lo stato "trova" per ricercare il minimo arco uscente dal nuovo frammento. Dopo aver aggiornato il proprio livello, l'identità del frammento e l'arco che lo connette al padre, il nodo u ritrasmette il messaggio *inizia* su tutti gli archi del frammento (tranne su quello dove ha ricevuto il messaggio, che porta al padre) e, se lo stato ricevuto è "trova", ricerca il minimo arco uscente da u con la procedura TEST.

when received (*inizia*, L', F', S) **on** a **do**
  begin
    $L := L'$; $F := F'$;
    statonodo := S;
    padre := a;
    migliore := λ; *mpu* := ∞;
    for each e **such that** $e \neq a$ **and** *statoarco*$[e]$ = "ramo" **do begin**
      send (*inizia*, F', L', S) **on** e;
      if S = "trova" **then** *conta-trova* := *conta-trova* + 1;
    end;
    if S = "trova" **then** TEST
  end;

Nella procedura TEST, il nodo u cerca il proprio minimo arco uscente spedendo messaggi di *verifica* sugli archi "base". È considerato un arco "base" per volta, in ordine di peso crescente, come mostrato nella Fig. 32.3, ottenendo come risposta messaggi *rifiuta* o *accetta*. Se u non ha trovato alcun candidato come proprio minimo arco uscente, allora esegue la procedura TROVATO-MINIMO.

procedure TEST;
  begin
    if esiste un arco incidente in u con stato "base" **then begin**
      candidato := arco incidente in u con stato "base" e peso minimo;
      send (*verifica*, L, F) **on** *candidato*
    end else begin
      candidato := λ;
      TROVATO-MINIMO
    end
  end;

Quando u riceve un messaggio di *verifica*, se il suo livello è minore di quello ricevuto, allora ritarda la risposta. Altrimenti, se le identità dei frammenti sono diverse,

risponde al mittente con un messaggio *accetta*, mentre se le identità dei frammenti sono le stesse allora (a parte un'eccezione) u risponde con *rifiuta* (si veda ancora la Fig. 32.3) e classifica l'arco "rifiutato" (lo stesso farà poi anche l'altro estremo v dell'arco). L'eccezione menzionata si ha quando u ha trasmesso una *verifica* ed ora riceve una *verifica* sullo stesso arco e per lo stesso frammento. In tal caso, u può rifiutare direttamente l'arco senza bisogno di inviare esplicitamente il messaggio *rifiuta*, e passa a rieseguire la procedura TEST.

```
when received (verifica, L', F') on a do
    begin
        if statonodo = "addormentato" then SVEGLIA;
        if L < L' then
            reinserisci il messaggio in fondo alla coda
        else if F ≠ F' then send (accetta) on a
            else begin
                if statoarco[a] = "base" then statoarco[a] := "rifiutato";
                if candidato ≠ a
                    then send (rifiuta) on a
                    else TEST
            end
    end;

when received (accetta) on a do
    begin
        candidato := λ;
        if c[a] < mpu then begin
            migliore := a; mpu := c[a];
        end;
        TROVATO-MINIMO;
    end;

when received (rifiuta) on a do
    begin
        if statoarco[a] = "base" then statoarco[a] := "rifiutato";
        TEST
    end;
```

Come illustrato nella Fig. 32.4, la computazione del minimo arco uscente dell'intero frammento parte dalle "foglie", cioè dai nodi che hanno padre come solo arco nello stato "ramo". Dopo che la "foglia" u ha ricevuto un messaggio *accetta* oppure ha esaurito la verifica su tutti gli archi "base", invia un messaggio di *minimo* al padre. Ciò avviene nella procedura TROVATO-MINIMO, che come abbiamo visto può essere chiamata in risposta ad un messaggio *accetta* oppure al termine della procedura TEST.

```
procedure TROVATO-MINIMO;
    begin
        if conta-trova = 0 and candidato = λ then begin
            statonodo := "trovato"
            send (minimo, mpu) on padre
        end
    end;
```

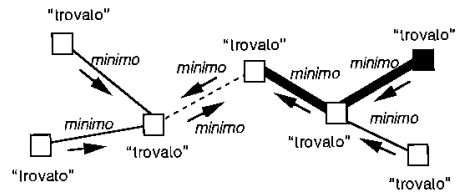



Figura 32.4: Computazione del minimo arco uscente dell'intero frammento (tale arco è trovato dal nodo nero e segue il percorso in grassetto fino a pervenire ai coordinatori).

Un nodo "interno" (cioè non "foglia") u aspetta finché non ha calcolato il minimo arco uscente dal nodo u stesso e non ha ricevuto messaggi di *minimo* da tutti i figli, dopodiché trasmette *mpu* al padre e passa nello stato "trovato" con una chiamata a TROVATO-MINIMO. Quando u riceve un messaggio *minimo* salva in *mpu* il minimo tra il *peso* ricevuto col messaggio ed il precedente valore di *mpu*, mentre salva in *migliore* l'identità dell'arco sul quale ha ricevuto il minimo. Nell'eventualità che u sia uno dei due coordinatori, il messaggio di minimo può essere ricevuto dall'altro coordinatore sull'arco $a = padre$.

```
when received (minimo, peso) on a do
    begin
        if a ≠ padre then begin
            conta-trova := conta-trova − 1;
            if peso < mpu then begin
                mpu := peso;
                migliore := a;
            end;
            TROVATO-MINIMO
        end else if statonodo = "trova"
            then reinserisci il messaggio in fondo alla coda
            else if peso > mpu
                then AGGIORNA-FRAMMENTI
                else if peso = mpu = ∞
                    then termina l'esecuzione;
    end;
```

Dopo che i due coordinatori del frammento si sono scambiati messaggi di *minimo* (si veda ancora la Fig. 32.4), ciascun coordinatore può determinare sia il peso del minimo arco uscente dal frammento sia in quale "metà" del frammento si trova tale arco. Infatti, gli archi *migliore* salvati in ogni nodo durante l'elaborazione dei messaggi di *minimo* permettono di trasmettere messaggi di *cambia* lungo il percorso dai coordinatori verso il nodo sul quale è incidente il minimo arco uscente dal frammento (come esemplificato nella Fig. 32.5 con gli archi in grassetto). Tale nodo spedirà un messaggio di *connetti*, che rilancerà l'attività di fusione o assorbimento. Con i successivi messaggi di *inizia* i due nodi estremi del minimo arco uscente diverranno coordinatori del nuovo frammento e saranno conseguentemente aggiornate in ogni nodo le informazioni relative al frammento (in particolare, verranno scambiati i ruoli padre/figlio tra i nodi estremi di ogni arco migliore salvato precedentemente).

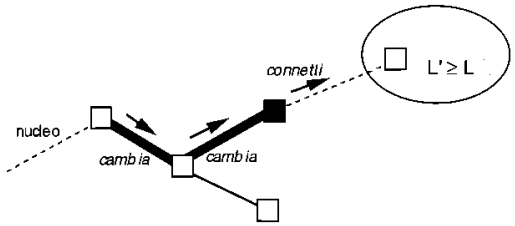


Figura 32.5: Richiesta di connessione al nodo che ha trovato il minimo arco uscente.

procedure AGGIORNA-FRAMMENTI;
  begin
    if *statoarco*[*migliore*] = "ramo" **then**
      send (*cambia*) **on** *migliore*
    else begin
      send (*connetti*, *L*) **on** *migliore*;
      statoarco[*migliore*] := "ramo"
    end
  end;

when received (*cambia*) **do** AGGIORNA-FRAMMENTI;

La correttezza dell'algoritmo deriva dal fatto che i frammenti si fondono o sono assorbiti lungo i loro minimi archi uscenti e che non ci possono essere attese infinite. Infatti, gli unici messaggi che possono causare attese sono *verifica* e *connetti*. Un messaggio *verifica* o sveglia un frammento a livello zero o avrà risposta senza attesa. Un messaggio *connetti* o sveglia un frammento a livello zero o va ad un frammento di livello più alto (con risposta immediata) o va ad un frammento dello stesso livello con lo stesso minimo arco uscente (creando un frammento di livello più alto). In ogni caso, un'azione è intrapresa senza attesa.

Valutiamo il numero totale di messaggi. Un arco può essere scartato solo una volta, poiché dopo essere stato scartato entrambi i suoi nodi estremi lo marcano "rifiutato", al prezzo di 2 messaggi. Pertanto il numero totale di messaggi di *verifica* e *rifiuta* è $O(m)$. Quando un nodo è ad un certo livello L (escluso $L = 0$ e l'ultimo), può ricevere al più un messaggio *inizia* ed al più un messaggio *accetta*, mentre può trasmettere al più un messaggio *verifica* (con successo), un *minimo*, ed un *cambia* o *connetti*, per un totale di 5 messaggi. Poiché $L \leq \log n$, come si può agevolmente dimostrare per induzione su L, ogni nodo può passare per $O(\log n)$ livelli. Pertanto il numero di messaggi per ogni nodo è $O(\log n)$, per un totale di $O(n \log n)$ messaggi. Quando un nodo è al livello $L = 0$, può ricevere al più un messaggio *inizia* e trasmettere al più un messaggio *connetti*, mentre quando è all'ultimo livello può spedire al più un messaggio *minimo*, per un totale di $O(n)$ messaggi. Pertanto, il numero totale di messaggi è $O(m + n \log n)$.

Consideriamo adesso la complessità in tempo, assumendo che un qualsiasi messaggio percorra un arco in tempo $O(1)$. Si può dimostrare che il tempo necessario per la computazione è $O(n^2)$ nel caso pessimo. Se però inizialmente vengono svegliati tutti i nodi (la qual cosa è fattibile in tempo $O(n)$), allora è possibile dimostrare che il tempo richiesto per trovare il minimo albero di copertura si riduce ad $O(n \log n)$.

Entrambe le dimostrazioni possono essere trovate nell'articolo originale di Gallager, Humblet e Spira (1983).

32.5 Esercizi

Esercizio 32.1 (Algoritmo di Lamport). Si progetti un algoritmo distribuito per la mutua esclusione che richieda $3(n-1)$ messaggi in un grafo completo con n processi.

Esercizio 32.2 (Algoritmo di Misra). Definire un algoritmo distribuito per un anello unidirezionale che sia in grado di diagnosticare l'eventuale perdita del token e di rigenerare il token perduto (suggerimento: si usino due token, *ping* e *pong*, e due variabili, *nping* e *npong*, di segno opposto ma stesso valore assoluto, che registrano il numero di volte che *ping* e *pong* si sono incontrati in un processo, in modo che valga l'invariante $nping + npong = 0$; se lo stesso token transita due volte di seguito dallo stesso processo, allora l'altro token è stato perso).

Esercizio 32.3 (Algoritmo di Hirshberg e Sinclair). Si valuti il tempo richiesto dall'algoritmo di Hirshberg e Sinclair per l'elezione del leader in un anello bidirezionale, assumendo che ogni messaggio inviato da un processo impieghi $O(1)$ tempo per essere ricevuto da un processo vicino.

Esercizio 32.4 (Colorazione). Si progetti una euristica "greedy" distribuita per colorare i nodi (processi) di un grafo generico, in modo che nodi adiacenti abbiano colori diversi e si usino "pochi" colori (suggerimento: assumendo che ogni processo conosca i nodi vicini con identificatori minori di esso, ogni processo aspetta che tali nodi si siano colorati e poi si colora col minimo colore disponibile).

Esercizio 32.5 (Colorazione in un anello). Si progetti un algoritmo distribuito efficiente per colorare i nodi (processi) di un anello unidirezionale col numero minimo di colori, assumendo che i processi ignorino il numero totale dei nodi.

Esercizio 32.6 (Algoritmo di Gallager, Humblet e Spira). Si dimostri con un esempio che l'algoritmo di Gallager, Humblet e Spira richiede $O(n^2)$ tempo nel caso pessimo, assumendo che ogni messaggio percorra un arco in tempo $O(1)$.

Esercizio 32.7 (Albero di copertura). Si progetti un algoritmo distribuito efficiente che, dato un nodo (processo) arbitrario di un grafo generico, trova un albero di copertura (non necessariamente minimo) radicato in tale nodo.

Esercizio 32.8 (Elezione del leader con albero di copertura). Si progetti un algoritmo distribuito che, dato un albero di copertura (non necessariamente minimo) di un grafo generico di n nodi, permette di eleggere un leader con $O(n)$ messaggi.

Conclusione

Giunti alla fine di questo testo, cerchiamo di ricavarne "la morale". Nel cercare di risolvere un problema, lo si affronta da due lati. Da una parte, si cerca di trovare costruttivamente un algoritmo che lo risolva, fornendo così una limitazione superiore alla complessità del problema. D'altro lato, si cerca di dimostrare matematicamente una limitazione inferiore alla complessità intrinseca del problema, che valga per qualsiasi algoritmo. Infine, se possibile, si cerca di individuare un algoritmo ottimo, dimostrando limitazioni inferiori via via più elevate e/o progettando algoritmi via via più veloci. In quest'ottica, le dimostrazioni di NP-completezza (oppure di indecidibilità) servono a provare rigorosamente limitazioni inferiori superpolinomiali (o addirittura infinite) della complessità.

In pratica, i due aspetti interagiscono e vengono entrambi affrontati dal progettista. Infatti, dati di ingresso particolarmente sfavorevoli per il comportamento di un certo algoritmo possono suggerire la strada da seguire in una dimostrazione di NP-completezza. D'altro canto, una prova di NP-completezza fallita può aiutare ad individuare particolari situazioni da affrontare nel progetto di un algoritmo.

Una caratterizzazione matematica della soluzione spesso suggerisce un semplice algoritmo di risoluzione, di solito non molto efficiente. Applicando opportune tecniche di progetto, delle quali le più usate sono il divide et impera e la programmazione dinamica, e organizzando i dati in strutture di dati appropriate, si può riuscire a migliorare notevolmente l'efficienza dell'algoritmo proposto. Queste tre fasi (caratterizzazione della soluzione, applicazione di tecniche di progetto, strutturazione dei dati) possono essere riconsiderate più di una volta, e in un ordine qualsiasi, ottenendo versioni sempre più efficienti dell'algoritmo, la cui complessità può magari raggiungere in ordine di grandezza una limitazione inferiore dimostrata precedentemente. Trovato un algoritmo efficiente in ordine di grandezza, si prova a modificarlo "limando" se possibile le costanti coinvolte nella funzione di complessità ed effettuando prove sperimentali su dati "reali", in modo da ottenere una versione dell'algoritmo che sia efficiente non solo in teoria ma anche in pratica. Per esempio, è ben noto che il "quicksort" e l'algoritmo del "simplesso", pur avendo complessità superiore in ordine di grandezza ad altri algoritmi che risolvono i problemi dell'ORDINAMENTO e della PROGRAMMAZIONE LINEARE, sono efficientissimi in pratica.

Se invece il problema è "intrattabile", si possono seguire diverse strade. Innanzitutto, si può vedere se c'è qualche caso speciale del problema che sia risolubile in tempo polinomiale. Se il problema generale è di ottimizzazione, si può tentare di individuare un algoritmo di approssimazione assoluta che abbia complessità polinomiale e fornisca una soluzione vicina a quella ottima. Se il problema non è "fortemente"

NP-arduo, si può cercare un algoritmo esatto di complessità pseudo-polinomiale oppure uno ϵ-approssimato di complessità polinomiale. Si possono progettare algoritmi esponenziali di tipo branch-&-bound, o ricercare algoritmi euristici, di solito basati su tecniche greedy o di ricerca locale, che siano particolarmente facili da programmare, abbiano se possibile complessità polinomiale, e forniscano sperimentalmente "buone" soluzioni su dati "reali". A tal fine, sono spesso disponibili via "internet" sia particolari dati di ingresso significativi del problema, sia il codice eseguibile e/o i risultati di vari algoritmi per lo stesso problema. È così possibile valutare la "bontà" del proprio algoritmo confrontandone, sugli stessi dati di ingresso, sia il valore della soluzione sia il tempo di calcolo con quelli di altri algoritmi già noti. In alcuni casi, si possono individuare algoritmi probabilistici che forniscono la soluzione corretta con probabilità molto vicina ad 1, richiedendo sempre tempo polinomiale (algoritmi Monte Carlo), oppure che richiedono tempo polinomiale con probabilità molto vicina ad 1, non fornendo mai una soluzione scorretta (algoritmi Las Vegas).

Molto importante è l'analisi dei "casi speciali" del problema. Spesso si è vittime di una inutile generalizzazione, che magari rende NP-arduo un problema altrimenti risolubile in tempo polinomiale! Se si è interessati solo ad alcuni casi speciali, l'intrattabilità del problema generalizzato è praticamente irrilevante.

Per esempio, la SODDISFATTIBILITÀ è NP-completa se in ogni clausola compaiono al più tre variabili (affermate o negate), ma è risolubile in tempo polinomiale se ne compaiono al più due! La CRICCA, l'INSIEME INDIPENDENTE, la COLORAZIONE, e il CIRCUITO HAMILTONIANO diventano polinomiali se formulati su un grafo di intervalli, i cui nodi rappresentano intervalli della retta reale e i cui archi rappresentano l'intersezione di due intervalli [cfr. § 9]. Viceversa, il MINIMO ALBERO DI COPERTURA diventa NP-completo se si richiede che ogni nodo dell'albero sia collegato ad al più k nodi, con k dato di input del problema. La PROGRAMMAZIONE LINEARE INTERA (anche solo 0/1) è NP-completa, ma la PROGRAMMAZIONE LINEARE, in cui le variabili x_i sono reali, è risolubile in tempo polinomiale. La 3-COLORAZIONE è NP-completa, ma la 2-COLORAZIONE è risolubile in tempo polinomiale. L'ABBINAMENTO 3-DIMENSIONALE è NP-completo, ma quello 2-DIMENSIONALE (che è equivalente all'usuale ABBINAMENTO) è risolubile in tempo polinomiale.

È istruttivo osservare come molti problemi, solo in apparenza simili, abbiano complessità diversa. Per esempio, il CIRCUITO HAMILTONIANO, in cui si chiede se esiste una catena chiusa che includa ogni nodo una e una sola volta, è NP-completo, mentre il CIRCUITO EULERIANO, in cui si chiede se esiste una catena chiusa che includa ogni arco una e una sola volta, è risolubile in tempo polinomiale. Analogamente, l'INSIEME INDIPENDENTE, dove si cerca un insieme di nodi a due a due non connessi da archi, è NP-arduo, mentre l'ABBINAMENTO, dove si cerca un insieme di archi a due a due non incidenti su uno stesso nodo, è risolubile in tempo polinomiale.

La Tabella C.1 seguente riassume 11 coppie di problemi apparentemente simili: quelli nella colonna di sinistra sono "facili" (risolubili in tempo polinomiale), mentre quelli nella colonna di destra sono "difficili" (NP-ardui).

Invece, nella Tabella C.2 è riportata la complessità dei migliori algoritmi conosciuti finora per risolvere gran parte dei problemi che sono stati considerati in questo testo, insieme agli autori degli algoritmi.

È bene rimarcare che l'impiego di tecniche euristiche oppure probabilistiche può risultare molto utile anche per progettare algoritmi per problemi che sono già noti essere trattabili. Per esempio, un algoritmo euristico dovuto a Boyer e Moore (1977)

	Problemi "facili" (polinomiali)	*Problemi "difficili" (NP-ardui)*
1.	DOMINO LINEARE	DOMINO LIMITATO
2.	2-SODDISFATTIBILITÀ	3-SODDISFATTIBILITÀ
3.	PROGRAMMAZIONE LINEARE	PROGRAMMAZIONE LINEARE 0/1
4.	ABBINAMENTO 2-DIMENSIONALE	ABBINAMENTO 3-DIMENSIONALE
5.	CIRCUITO EULERIANO	CIRCUITO HAMILTONIANO
6.	ZAINO (REALE)	ZAINO (INTERO)
7.	MINIMO ALBERO DI COPERTURA	MINIMO ALBERO DI COPERTURA CON k CONNESSIONI
8.	CAMMINI MINIMI	CAMMINI MINIMI SENZA NODI RIPETUTI
9.	2-COLORAZIONE	3-COLORAZIONE
10.	PROGRAMMI IN RITARDO	SCHEDULING MULTIPROCESSORE
11.	EQUAZIONI DIOFANTINE $ax^k = c$ e/o $\sum_i a_i x_i = c$ x_i intero positivo $\forall\, i$	EQUAZIONI DIOFANTINE $ax^2 + by = c$ x e y interi positivi

Tabella C.1: Coppie di problemi apparentemente simili ma con complessità diversa.

ed un algoritmo probabilistico dovuto a Karp e Rabin (1981) sono gli algoritmi più veloci in pratica per risolvere il problema dello STRING MATCHING.

Per abbassare ulteriormente il tempo richiesto per risolvere problemi trattabili, può essere opportuna l'introduzione di qualche forma di parallelismo. Mentre la scelta di un modello asincrono (concorrente e/o distribuito) è forzata dalla natura stessa del problema, quella di un modello sincrono (in pratica, una opportuna rete a grado limitato) è invece una libera scelta del progettista. I modelli sincroni sono in genere più adatti ad esplicitare il massimo parallelismo insito in un problema, poiché permettono di sfruttare appieno l'eventuale regolarità strutturale del problema stesso. L'utilizzo di modelli VLSI è invece di solito ristretto alla realizzazione di circuiti per pochi problemi fondamentali, come l'ORDINAMENTO[1] o la MOLTIPLICAZIONE DI MATRICI. È possibile così arricchire la "libreria" di un usuale calcolatore sequenziale (che esegue programmi "software") con due o tre circuiti VLSI paralleli molto veloci utilizzati come sottoprogrammi "hardware" di libreria per la risoluzione dei summenzionati problemi. Infine, allo stato attuale della tecnologia, il modello con "mesh riconfigurabili" è ancora avveniristico, ma potrebbe diventare il modello parallelo vincente se la tecnologia permetterà in un prossimo futuro di produrre calcolatori ottici a basso costo ed in grandi quantità.

[1] Ricordiamo che in francese il calcolatore è chiamato non a caso "ordinateur" (cioè ordinatore). Infatti, da studi statistici effettuati, risulta che un calcolatore trascorre la maggior parte della sua "vita" computazionale ad ordinare dati.

	Problemi	*Complessità e autori*
1.	PROGRAMMAZIONE LINEARE	$O(n^{3.5}L^2)$ $L = \log \lvert \prod a_{ij} \prod b_i \rvert$ Karmarkar (1984)
2.	FLUSSO MASSIMO	$O(mn \log(n^2/m))$ Goldberg-Tarjan (1986)
3.	ABBINAMENTO	$O(m\sqrt{n})$ Micali-Vazirani (1980)
4.	MINIMO ALBERO DI COPERTURA	$O(min\{m + n \log n, m\beta\})$ $\beta = \min\{i : \log^{(i)} n \leq m/n\}$ Fredman-Tarjan (1987)
5.	CAMMINI MINIMI (TUTTE LE COPPIE)	$O(mn + n^2 \log n)$ Fredman-Tarjan (1987)
6.	TRASFORMATA DI FOURIER	$O(n \log n)$ Cooley-Tukey (1965)
7.	MOLTIPLICAZIONE DI MATRICI	$O(n^{2.376})$ Coppersmith-Winograd (1987)
8.	INVILUPPO CONVESSO	$O(n \log n)$ Graham (1972)
9.	STRING MATCHING	$O(n + m)$ Knuth-Morris-Pratt (1977)
10.	STRING MATCHING k-APPROSSIMATO	$O(kn)$ Landau-Vishkin (1986)
11.	ORDINAMENTO	$O(n \log n)$ Shell (1959), Hoare (1962), Williams (1964)

Tabella C.2: Complessità dei migliori algoritmi conosciuti per alcuni problemi.

Appendici

Appendice A

Numeri casuali

Una caratteristica importante dei calcolatori consiste nella possibilità di generare numeri "casuali". Tali numeri sono utilissimi sia nel progetto di algoritmi probabilistici (cfr. § 20), sia per generare dati arbitrari di input al fine di provare sperimentalmente il comportamento di algoritmi euristici (cfr. § 24).

Un numero casuale è un numero che ha una certa probabilità di occorrere all'interno di un dato intervallo di variabilità. Per semplicità, consideriamo probabilità uniformi, dove ciascun numero all'interno dell'intervallo ha la stessa probabilità di occorrere di tutti gli altri, ed intervalli finiti, come per esempio l'intervallo [0, 1] dei numeri reali oppure un generico intervallo $[s, d]$ dei numeri interi, con $s < d$. Nella maggior parte dei casi, non si richiede un singolo numero casuale soltanto, ma piuttosto una sequenza di numeri casuali.

È bene rimarcare che, utilizzando un algoritmo deterministico per generare numeri casuali, qualsiasi sequenza prodotta può essere prevista anticipatamente una volta che sia noto l'algoritmo stesso. In tal caso, i numeri prodotti dall'algoritmo (detto "generatore") non sono affatto casuali! Se però le sequenze di numeri generati superano alcuni test statistici di casualità, come per esempio il famoso test χ^2 ("chi-quadro"), si hanno sequenze di numeri "pseudo-casuali". Tali numeri, pur non essendo casuali, si comportano dal punto di vista statistico come se lo fossero veramente, e possono essere pertanto utilizzati come "buona approssimazione" dei numeri casuali.

Sulla generazione di numeri pseudo-casuali e sui relativi test statistici di casualità sono stati scritti volumi su volumi. Quasi tutti i sistemi di calcolo ed i libri di testo di informatica contengono programmi per generare (supposti) numeri pseudo-casuali. Purtroppo, la maggior parte di tali programmi non è affatto soddisfacente. Come suggerito dallo stesso Knuth (1969-1973), la prima cosa da fare quando si vogliono usare numeri casuali consiste nel sostituire il generatore fornito dalla "libreria" di sistema con uno proprio, del quale sia garantita la bontà!

In questo testo, consideriamo un generatore molto semplice, quello "congruente lineare moltiplicativo" introdotto da Lehmer nel 1951, e ne vedremo diverse realizzazioni Pascal, proposte da Park e Miller (1988), che possono essere installate virtualmente su qualsiasi sistema di calcolo.

A.1 Generatore di Lehmer

Il generatore congruente lineare moltiplicativo proposto da Lehmer è estremamente semplice. Se due parametri interi (il "modulo" ed il "moltiplicatore") sono scelti in maniera opportuna e se la sua realizzazione software (p.e. con un programma Pascal) è corretta, allora il generatore dà la garanzia di fornire una sequenza virtualmente infinita di numeri che superano qualsiasi test statistico di casualità.

Il generatore di Lehmer dipende dai seguenti parametri:

1. il *modulo* m, che deve essere un numero primo piuttosto "grande";
2. il *moltiplicatore* a, che deve essere un numero intero compreso tra 2 ed $m - 1$.

Scelto un intero iniziale z_1, detto seme, compreso tra 1 ed $m - 1$, l'algoritmo genera una sequenza di interi $z_1, z_2, z_3, \ldots$ per mezzo della formula iterativa

3. $z_{i+1} = f(z_i)$, per $i = 1, 2, \ldots$

dove $f(z)$ è definita per ogni z compreso tra 1 ed $m - 1$ come segue:

4. $f(z) = (az) \bmod m$.

Oltre alla sequenza di interi $z_1, z_2, z_3, \ldots$ può essere prodotta anche una sequenza di reali $r_1, r_2, r_3, \ldots$ appartenenti all'intervallo (0, 1), dove

5. $r_i = z_i/m$, per $i = 1, 2, \ldots$

Poiché m è primo, $f(z) \neq 0$ per ogni z compreso tra 1 ed $m - 1$, la qual cosa evita che da un certo punto in poi vengano generati soltanto degli zeri nella sequenza delle z. Inoltre, i valori reali $r = 0$ ed $r = 1$ non sono mai generati, mentre il più piccolo reale r generato è $1/m$ ed il più grande è $1 - 1/m$. Se a ed m sono scelti oculatamente, è stato dimostrato che la sequenza delle z è in effetti pseudo-casuale, e quindi statisticamente indistinguibile da una sequenza di interi estratti casualmente dall'insieme $\{1, 2, \ldots, m - 1\}$. Inoltre, se la sequenza delle z è pseudo-casuale, allora anche quella delle r lo è. A tal fine, è opportuno che a ed m siano scelti in modo che il generatore sia caratterizzato da un periodo massimo, ovvero non generi due volte lo stesso valore senza aver generato nel frattempo tutti gli altri $m - 2$ valori possibili.

Esempio A.1 (Generatore di Lehmer). Se $f(z) = (6z) \bmod 13$ e $z_1 = 1$, allora la sequenza delle z è:

$$\ldots 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \ldots$$

dove la stessa sottosequenza si ripete ciclicamente all'infinito, con periodo massimo $m - 1 = 12$. Si noti che, essendo il periodo massimo, un qualsiasi seme compreso tra 1 e 12 va bene. Infatti, ciascun seme produce una sequenza dove la sottosequenza periodica non è che una traslazione ciclica di quella con seme uguale ad 1. Se invece $f(z) = (7z) \bmod 13$, allora la sequenza delle z è:

$$\ldots 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 \ldots$$

che, pur avendo periodo massimo, è però considerata "meno casuale" della precedente, a causa della configurazione evidente nella seconda metà della sottosequenza periodica. Infine, se $f(z) = (5z) \bmod 13$, allora non si ha neanche periodo massimo, poiché la sequenza è ... 1, 5, 12, 8, 1 ... oppure ... 2, 10, 11, 3, 2 ... oppure ... 4, 7, 9, 6, 4 ... a seconda del seme. □

È possibile dimostrare che per ogni modulo m primo ($m \geq 3$) una buona percentuale delle $m - 2$ possibili scelte del moltiplicatore a garantisce periodo massimo. Per $m = 13$, i moltiplicatori $a = 2, 6, 7, 11$ forniscono un periodo massimo, mentre per $m = 2^{31} - 1 = 2147483647$ il moltiplicatore $a = 7^5 = 16807$ è uno dei 534 milioni di moltiplicatori che danno periodo massimo.

In particolare, proprio i valori $m = 2^{31} - 1$ ed $a = 7^5$ sono caldamente raccomandati da Park e Miller (1988) per ottenere un generatore di numeri pseudo-casuali che superi praticamente ogni test statistico di casualità. Oltre a garantire casualità, tale scelta è utile perché lo standard in quasi tutti i calcolatori odierni prevede di rappresentare un intero su una parola di memoria di 32 bit. Pertanto, la scelta di $m = 2^{31} - 1$ permette di eseguire molto velocemente l'operazione modulo m mentre quella di $a = 7^5$ garantisce un periodo massimo molto lungo (oltre due miliardi).

A.2 Realizzazione in Pascal

Il generatore di Lehmer con $m = 2^{31} - 1$ ed $a = 7^5$ ha praticamente superato tutti i test statistici di casualità. Nel realizzarlo con una procedura, per esempio in Pascal, bisogna però dimostrare di ottenerne una realizzazione corretta, cioè che genera effettivamente i numeri che ci si aspetta, poiché durante il calcolo potrebbe verificarsi una situazione di "overflow", cioè di supero della capacità di rappresentazione di un numero. In altri termini, la stessa procedura Pascal potrebbe funzionare su alcuni calcolatori ma non su altri, a seconda del numero di bit usati dalla macchina per rappresentare numeri interi (o reali) e del valore che assume la costante Pascal *maxint* (il massimo intero rappresentabile, che dipende anch'esso dalla macchina).

La più semplice realizzazione Pascal è data dalla seguente funzione RANDOM, che restituisce un reale nell'intervallo (0, 1). La funzione utilizza una variabile globale intera *seme*, che contiene il valore corrente di z e che deve essere inizializzata una volta per tutte ad un valore compreso tra 1 e 2147483646 (per esempio la codifica su 8 cifre delle propria data di nascita, in modo da avere sequenze diverse per utenti diversi).

```
var seme: integer;

function RANDOM: real;
    const a = 16807; m = 2147483647;
    begin
        seme := (a * seme) mod m;
        RANDOM := seme / m
    end;
```

Poiché il prodotto $a * seme$ può crescere fino a $16807 \cdot 2147483647 \cong 1.03 \cdot 2^{45}$, questa versione del generatore è corretta solo se $maxint \geq 2^{46} - 1$. Se *maxint* è minore di $2^{46} - 1$, come avviene nella maggior parte dei sistemi, si hanno condizioni di "overflow" su interi, che producono errori.

Una semplice verifica di correttezza del generatore consiste nel partire con $z_1 = 1$, generare 10000 numeri, e poi constatare se $z_{10001} = 1043618065$ oppure no:

```
seme := 1;
for i := 1 to 10000 do u := RANDOM;
writeln(seme);
```

Infatti, se deve avvenire un "overflow", che quindi rende scorretti tutti i valori prodotti da tale istante in poi oppure blocca l'esecuzione con un messaggio di errore, allora sicuramente l'"overflow" occorrà in tale sequenza di 10000 numeri, poiché vi deve essere prodotto un valore intermedio di *seme* maggiore o uguale a 2147483531.

Una seconda versione della funzione Pascal RANDOM utilizza la rappresentazione dei numeri reali per effettuare i calcoli sugli interi effettuando l'operazione in modulo tramite la funzione di libreria *trunc*, che restituisce il valore dell'argomento privato delle cifre decimali:

```
var seme: real;

function RANDOM: real;
    const a = 16807.0; m = 2147483647.0;
    var temp: real;
    begin
        temp := a * seme;
        seme := temp − m * trunc(temp / m);
        RANDOM := seme / m
    end;
```

La seconda versione del generatore è corretta se i reali sono rappresentati con una mantissa di almeno 46 bit (escluso il bit del segno) e se $maxint \geq 16806$, che è il valore massimo che *trunc(temp/m)* può assumere (p.e. esempio, questa versione del generatore è corretta sui sistemi che impiegano la rappresentazione dei reali su 64 bit in accordo allo standard dell'IEEE).

Se nessuna delle due precedenti realizzazioni del generatore risulta corretta sul sistema di calcolo a disposizione, allora si può impiegare una tra due ulteriori versioni, la correttezza delle quali è dimostrata da Park e Miller (1988) e non è qui riportata. Entrambe queste versioni sono funzionalmente equivalenti e utilizzano due costanti p e q, che valgono m **mod** a ed m **div** a, rispettivamente. Come le due versioni precedenti, differiscono solo per l'utilizzo della rappresentazione dei numeri interi oppure reali per effettuare i calcoli sugli interi, e la variabile globale *seme* va di conseguenza dichiarata intera oppure reale, rispettivamente.

```
function RANDOM: real;
    const a = 16807; m = 2147483647; q = 127773; p = 2836;
    var low, high, test: integer;
    begin
        high := seme div q;
        low := seme mod q;
        test := a * low − p * high;
        if test > 0
            then seme := test
            else seme := test + m;
        RANDOM := seme / m
    end;
```

Poiché la variabile *test* assume sempre valori interi rappresentabili su 32 bit (incluso il bit del segno), la terza versione del generatore è corretta qualora $maxint \geq 2^{31} - 1$.


```
function RANDOM: real;
    const a = 16807.0; m = 2147483647.0; q = 127773.0; p = 2836.0;
    var low, high, test: real;
    begin
        high := trunc(seme / q);
        low := seme − q * high;
        test := a * low − p * high;
        if test > 0.0
            then seme := test
            else seme := test + m;
        RANDOM := seme / m
    end;
```

La quarta versione del generatore è quella da usare quando *maxint* è minore di $2^{31} - 1$ (come avviene per esempio su alcuni microprocessori). Questa versione è corretta se i reali sono rappresentati con una mantissa di almeno 32 bit (incluso il bit di segno) e $maxint > 16807$.

A.3 Prove sperimentali di algoritmi

Come accennato precedentemente, i generatori di numeri pseudo-casuali sono utilissimi, oltre che nel progetto di algoritmi probabilistici che richiedono esplicitamente la funzione RANDOM, anche per provare sperimentalmente il comportamento di algoritmi deterministici su dati arbitrari di ingresso.

Ovviamente, non si è vincolati a considerare sempre e solo numeri reali nell'intervallo $(0, 1)$ oppure numeri interi nell'intervallo $[1, m - 1]$. Infatti, per generare un reale x compreso in un intervallo arbitrario (s, d), è sufficiente generare un reale r in $(0, 1)$ e quindi porre $x = s + r(d - s)$. Analogamente, per generare un intero n appartenente all'intervallo $[s, d)$, con s e d interi, basta porre $n = s + \lfloor r(d - s) \rfloor$. Inoltre, se occorre effettuare una certa azione con una data probabilità $p \neq 0$, è sufficiente generare r e quindi effettuare l'azione solo se $r \leq p$. Grazie a queste regole, è possibile generare casualmente (strutture di) dati di ingresso, anche piuttosto complicate e di grandi dimensioni, sulle quali provare ad eseguire gli algoritmi.

Esempio A.2 (Prove sperimentali su grafi). Si assuma di volere sperimentare il comportamento di un algoritmo per grafi non orientati. Le prove sperimentali possono essere effettuate su grafi generati casualmente, al variare del numero n dei nodi. Tali valori di n dipendono in genere dalla natura del problema da risolvere e possono variare da poche centinaia a parecchie centinaia di migliaia (p.e. gli algoritmi per la BISEZIONE trattati nel § 24 sono stati provati su grafi di oltre 200000 nodi e 500000 archi). Fissato n, e fissati eventuali altri parametri, vengono generati casualmente parecchi grafi di ingresso con le stesse caratteristiche ed i risultati prodotti dall'algoritmo sono poi mediati tra tutti i grafi generati. Per motivi statistici, il numero di grafi generati con le stesse caratteristiche è di solito 30. Per generare casualmente un grafo di n nodi, si possono seguire svariati criteri, dei quali i due principali sono i seguenti:

1. Fissato n e data una probabilità p in $(0, 1)$, un arco tra ciascuna coppia di nodi i e j, con $1 \leq i \neq j \leq n$, è inserito con probabilità p;
2. Fissato n e dato un reale δ in $(0, 1)$, ciascun nodo è associato ad un punto di un quadrato di lato unitario, cioè è caratterizzato da una ascissa ed una ordinata, entrambe generate casualmente in $(0, 1)$; un arco tra ciascuna coppia di nodi i e

j, con $1 \leq i \neq j \leq n$, è inserito se la distanza Euclidea tra i punti i e j nel quadrato è al più δ.

Si noti come nel primo caso il numero atteso di archi incidenti in ciascun nodo sia $p(n-1)$, mentre nel secondo caso tale numero sia $n\pi\delta^2$. Pertanto, variando i parametri p e δ si può agire sul numero atteso di archi incidenti in ciascun nodo, controllando così la densità del grafo. Oltre al variare di n, le prove sperimentali possono quindi essere effettuate, per un dato n, al variare di p oppure di δ. □

A.4 Esercizi

Esercizio A.1. Si consideri il generatore definito dalla funzione $f(z) = (9806z + 1) \bmod 131071$. Si discuta cosa accade qualora $z = 37911$.

Esercizio A.2. Si consideri il generatore definito dalla funzione $f(z) = (25173z + 13849) \bmod 216$. Si verifichi che nella rappresentazione binaria di z il bit meno significativo cicla con periodo 2, quello immediatamente precedente cicla con periodo 4, e così via.

Esercizio A.3. La seguente funzione Pascal è un buon generatore di interi nell'intervallo $[0, c-1]$?

```
function PIPPO(c: integer): integer;
    var u: real;
    begin
        u := RANDOM;
        PIPPO := seme mod c
    end;
```

Esercizio A.4. Si scriva una procedura Pascal per generare casualmente stringhe alfabetiche di 6 lettere.

Esercizio A.5. Si scriva una procedura Pascal per generare casualmente una matrice binaria di dimensione 10×10.

Esercizio A.6. Si scriva una procedura Pascal per generare casualmente un grafo orientato pesato di 100 nodi.

Appendice B

Il linguaggio C

Data l'enorme diffusione del linguaggio C, se ne fornisce in questa appendice una breve introduzione, mostrando differenze ed analogie col linguaggio Pascal.

Per descrivere i costrutti principali del C, si considerano soltanto alcuni semplici esempi, cominciando dalla procedura ORDINA introdotta nell'Esempio I.10 per ordinare un vettore di n interi con complessità $O(n^2)$, che richiama al suo interno la funzione MIN.

B.1 Confronto tra Pascal e C

A parte la diversa sintassi, si nota una forte corrispondenza tra le istruzioni dalla versione C di ORDINA e quelle dalla sua controparte Pascal:

```
void ORDINA(int *A, int n)
    {
    int j, pos, temp;
    int MIN(int *, int, int, int *);

    for (j = 0; j < n; j++)
        {
        temp = A[j]; A[j] = MIN(A, j, n, &pos); A[pos] = temp;
        }
    }

int MIN(int *B, int j, int k, int *pos)
    {
    int m, i;

    m = B[j]; *pos = j;
    for (i = j + 1; i <= k; i++)
        if (B[i] < m)
            {
            m = B[i]; *pos = i;
            }
    return(m);
    }
```

Le differenze principali che balzano subito all'occhio sono le seguenti. Innanzitutto in C si usano i delimitatori { e }, anziché **begin** ed **end**, che in più rispetto al Pascal individuano un blocco all'inizio del quale si possono dichiarare variabili locali solo all'interno di quel blocco. Nelle dichiarazioni di variabili e procedure si antepone il tipo al nome della variabile o della procedura, anziché postporlo, e non si usano parole chiave equivalenti a quelle Pascal **var** o **procedure**. Inoltre, in C si hanno solo funzioni che restituiscono un valore e quindi una procedura è vista come una funzione particolare di tipo "void", cioè che restituisce il valore "vuoto". Non si usa assegnare un valore al nome della funzione (come con "MIN := *m*" in Pascal), ma si utilizza l'istruzione apposita **return**, seguita da un'espressione racchiusa tra parentesi tonde. Anche una funzione di tipo void può usare **return**, purché la **return** non abbia parametri. Negli assegnamenti si utilizza l'operatore "=" anziché ":=", mentre l'operatore di uguaglianza è "==" anziché "="; l'operatore di disuguaglianza è "!=", che in Pascal è indicato invece con "<>" (anche se nel presente testo è stato utilizzato "≠" per chiarezza). Non si scrive esplicitamente il **then** dopo la condizione di un **if** (però si specifica l'eventuale **else**, come in Pascal). Il costrutto di iterazione limitata ha la forma seguente:

" **for** (indice = limiteinferiore; condizione; incremento) istruzione"

dove l'istruzione che forma il corpo del **for** è ripetuta fino a quando la condizione non è più verificata. Un incremento di uno è abbreviato con "*i*++", che è l'equivalente dell'assegnamento Pascal "$i := i + 1$". Riguardo al passaggio dei parametri, in C esiste solo la modalità di passaggio di un parametro per valore. Per ottenere il passaggio per "variabile" (**var** in Pascal), in C si deve ricorrere ai puntatori e passare il valore del puntatore alla variabile, cioè l'indirizzo della variabile stessa. Si indica con $*pos$ una variabile puntata dal puntatore *pos*, così che l'assegnamento "$*pos = i$" modifica il valore della variabile, mentre nella chiamata della procedura si usa $\&pos$, che indica il valore del puntatore *pos*, cioè l'indirizzo della variabile puntata. Infine, in C gli indici degli elementi dei vettori partono da 0, e per selezionare un elemento di una matrice si usa la notazione $A[i][j][k]$ anziché quella $A[i, j, k]$ del Pascal.

Si consideri adesso un esempio un po' più complesso, dove si fornisce in C la realizzazione bidirezionale circolare con sentinella di una lista, che è stata introdotta nel § 2. Le opportune dichiarazioni sono le seguenti.

```
#include <stdlib.h>

typedef int tipoelem;

struct cella
    {
    tipoelem elemento;
    struct cella *precedente, *successivo;
    };

typedef struct cella *lista, *posizione;

void CREALISTA(lista *);
char LISTAVUOTA(lista);
posizione PRIMOLISTA(lista);
posizione ULTIMOLISTA(lista);
posizione SUCCLISTA(posizione, lista);
posizione PREDLISTA(posizione, lista);
```



```
char FINELISTA(posizione, lista);
tipoelem LEGGILISTA(posizione, lista);
void SCRIVILISTA(tipoelem, posizione, lista);
void INSLISTA(tipoelem, posizione *, lista);
void CANCLISTA(posizione *, lista);
```

In C, il tipo record si individua con la parola chiave **struct**. Per accedere ad un campo del record si usa lo stesso operatore "." del Pascal, ma per accedere ad un campo di un record puntato da un puntatore, in C si usa l'operatore "→" anziché la combinazione dei due operatori "↑." del Pascal. **typedef** indica la definizione di un tipo, e l'espressione "**struct** cella **precedente*" indica un puntatore ad un record "cella". In C non è prevista la possibilità che una funzione restituisca un tipo booleano e una funzione booleana è "simulata" con una che restituisce un carattere (char) oppure un intero (int). Per allocare una nuova porzione di memoria, in C si usa la funzione di libreria *malloc*, che è analoga alla *new* Pascal, ma che richiede come parametro il numero di byte della porzione di memoria da allocare. Per utilizzarla, occorre l'istruzione di inclusione delle funzioni di libreria **#include**. Per ottenere la dimensione di una struttura, c'è un'altra funzione di libreria *sizeof*, che restituisce il numero di byte del suo argomento. Per liberare memoria, si usa la funzione *free*, che è la controparte C della *dispose* Pascal.

La realizzazione completa degli operatori delle liste è la seguente.

```
void CREALISTA(lista *L)
    {
    (*L) = (lista) malloc(sizeof(struct cella));
    (*L)→precedente = *L;
    (*L)→successivo = *L;
    }

char LISTAVUOTA(lista L)
    {
    return(L→successivo == L);
    }

posizione PRIMOLISTA(lista L)
    {
    return(L→successivo);
    }

posizione ULTIMOLISTA(lista L)
    {
    return(L→precedente);
    }

posizione SUCCLISTA(posizione p, lista L)
    {
    return(p→successivo);
    }
```

```
posizione PREDLISTA(posizione p, lista L)
    {
    return(p→precedente);
    }

char FINELISTA(posizione p, lista L)
    {
    return(p == L);
    }

tipoelem LEGGILISTA(posizione p, lista L)
    {
    return(p→elemento);
    }

void SCRIVILISTA(tipoelem a, posizione p, lista L)
    {
    p→elemento = a;
    }

void INSLISTA(tipoelem a, posizione *p, lista L)
    {
    struct cella *temp;

    temp = (struct cella *)malloc(sizeof(struct cella));
    temp→precedente = (*p)→precedente;
    temp→successivo = *p;
    temp→precedente→successivo = temp;
    (*p)→precedente = temp;
    temp→elemento = a;
    (*p) = temp;
    }

void CANCLISTA(posizione *p, lista L)
    {
    posizione temp;

    (*p)→successivo→precedente = (*p)→precedente;
    (*p)→precedente→successivo = (*p)→successivo;
    temp = (*p)→successivo;
    free(*p);
    *p = temp;
    }
```

Vediamo infine come impiegare gli operatori delle liste per trovare l'intersezione C di due insiemi A e B, dove gli insiemi sono realizzati con liste ordinate, come spiegato nel § 5.


```
typedef lista insieme;

void INTERSEZIONE(insieme A, insieme B, insieme *C)
    {
    posizione p, q, r;

    CREALISTA(C);
    p = PRIMOLISTA(A); q = PRIMOLISTA(B); r = PRIMOLISTA(*C);
    while (!FINELISTA(p, A) && !FINELISTA(q, B))
        {
        if (LEGGILISTA(p, A) == LEGGILISTA(q, B))
            {
            INSLISTA(LEGGILISTA(p, A), &r, *C);
            p = SUCCLISTA(p, A);
            q = SUCCLISTA(q, B);
            r = SUCCLISTA(r, *C);
            }
        else if (LEGGILISTA(p, A) < LEGGILISTA(q, B))
                p = SUCCLISTA(p, A);
            else
                q = SUCCLISTA(q, B);
        }
    }
```

L'iterazione illimitata in C si indica con "**while** (condizione) istruzione" che è analogo al Pascal (a parte la mancanza del "**do**" fra la condizione e l'istruzione). Gli operatori booleani **!**, **&&**, e || del C sono analoghi, rispettivamente, agli operatori booleani **not**, **and**, e **or** del Pascal.

Per concludere questa breve introduzione, la Tabella B.1 riassume i principali operatori, tipi, parole chiave, e funzioni di libreria di C e Pascal.

B.2 Esercizi

Esercizio B.1 (Pile). Si scrivano le procedure C per realizzare una pila con un vettore, come spiegato nel § 3.

```
#define maxlung 1024
typedef int tipoelem;
typedef struct
    {
    int testa;
    tipoelem elementi[maxlung];
    } pila;

void CREAPILA(pila *);
char PILAVUOTA(pila *);
tipoelem LEGGIPILA(pila *);
void FUORIPILA(pila *);
void INPILA(tipoelem, pila *);
```

	Pascal	C
PAROLA CHIAVE		
dichiarazione di array	**array**	[]
dichiarazione di record	**record**	**struct**
dichiarazione di variabili	**var**	
definizione di tipo	**type**	**typedef**
definizione di costante	**const**	**const**
inizio/fine di un blocco	**begin** ... **end**	{ ... }
istruzione condizionale	**if** ... **then** ... **else** ...	**if** (...) ... **else** ...
iterazione *limitata*	***for*** ... ***to*** ... ***do*** ...	***for*** (... ; ... ; ...) ...
iterazione illimitata	**while** ... **do** ...	**while** (...) ...
funzione	**function**	
procedura	**procedure**	void
ritorno da procedura	**exit**	**return**
salto incondizionato	**goto**	**goto**
puntatore nullo	**nil**	NULL
inserimento di librerie	**unit, uses**	**#include**
tipo intero	integer	*int*
tipo carattere	character	char
tipi reale, reale lungo	real, double	float, double
tipo booleano	boolean	char, int
OPERATORE		
assegnamento	:=	=
addizione	+	+
incremento		++
sottrazione	−	−
decremento		−−
moltiplicazione	*	*
divisione	/	/
divisione intera	**div**	/
resto della divisione intera	***mod***	%
prodotto logico	**and**	**&&**
somma logica	**or**	||
negazione logica	**not**	!
uguaglianza	=	==
non uguaglianza	<>	!=
minore	<	<
maggiore	>	>
minore o uguale	<=	<=
maggiore o uguale	>=	>=
indice di un vettore	[ ]	[ ]
indici di una matrice	[, , . . . ,]	[ ] [ ] . . . [ ]
intervallo	..	
variabile puntata	↑	*
indirizzo	@	&
selezione campo record	.	.
campo record puntato	↑.	→
FUNZIONE DI LIBRERIA		
lettura	*read*(), *readln*()	*scanf*()
scrittura	*write*(), *writeln*()	*printf*()
allocazione memoria	*new*()	*malloc*()
rilascio memoria	*dispose*()	*free*()
dimensione memoria		*sizeof*()

Tabella B.1: I principali operatori, tipi, parole chiave, e funzioni di libreria di C e Pascal.


```
void CREAPILA(pila *P)
    {
    P→testa = 0;
    }

char PILAVUOTA(pila *P)
    {
    return(P→testa == 0);
    }

tipoelem LEGGIPILA(pila *P)
    {
    if (!PILAVUOTA(P))
        return(P→elementi[(P→testa) − 1]);
    }

void FUORIPILA(pila *P)
    {
    if (!PILAVUOTA(P)) P→testa−−;
    }

void INPILA(tipoelem a, pila *P)
    {
    if (p→testa != maxlung)
        p→elementi[p→testa++] = a;
    else
        printf("Errore: pila piena.\n");
    }
```

Esercizio B.2 (Code). Si scrivano le procedure C per realizzare una coda con un vettore, come spiegato nel § 3.

```
#define maxlung 1024
typedef int tipoelem;
typedef struct
    {
    int testa;
    int lung;
    tipoelem elementi[maxlung];
    } coda;

void CREACODA(coda *);
char CODAVUOTA(coda *);
tipoelem LEGGICODA(coda *);
void FUORICODA(coda *);
void INCODA(tipoelem, coda *);

void CREACODA(coda *Q)
    {
    Q→testa = 0; Q→lung = 0;
    }
```

```
char CODAVUOTA(coda *Q)
    {
    return(Q→lung == 0);
    }

tipoelem LEGGICODA(coda *Q)
    {
    if (!CODAVUOTA(Q))
        return(Q→elementi[Q→testa]);
    }

void FUORICODA(coda *Q)
    {
    if (!CODAVUOTA(Q))
        {
        Q→testa = (Q→testa + 1) % maxlung;
        Q→lung--;
        }
    }

void INCODA(tipoelem a, coda *Q)
    {
    if (Q→lung < maxlung)
        Q→elementi[(Q→testa + Q→lung++) % maxlung] = a;
    }
```

Esercizio B.3 (Insiemi). Si scrivano le procedure C per realizzare un insieme con un vettore booleano, come spiegato nel § 5.

```
#define N 1024
typedef int tipoelem;
typedef char * insieme;

void CREAINSIEME(insieme);
char INSIEMEVUOTO(insieme);
char APPARTIENE(tipoelem, insieme);
void INSERISCI(tipoelem, insieme);
void CANCELLA(tipoelem, insieme);
void UNIONE(insieme, insieme, insieme);
void INTERSEZIONE(insieme, insieme, insieme);
void DIFFERENZA(insieme, insieme, insieme);

void CREAINSIEME(insieme A)
    {
    int i;

    A = (insieme)malloc(N * sizeof(char));
    for (i = 0; i < N; i++) A[i] = 0;
    }
```



```
char INSIEMEVUOTO(insieme A)
    {
    int i;

    i = 0;
    while (i < N && !A[i]) i++;
    return(i == N);
    }

char APPARTIENE(tipoelem i, insieme A)
    {
    return(A[i]);
    }

void INSERISCI(tipoelem i, insieme A)
    {
    A[i] = 1;
    }

void CANCELLA(tipoelem i, insieme A)
    {
    A[i] = 0;
    }

void UNIONE(insieme A, insieme B, insieme C)
    {
    int i;

    for (i = 0; i < N; i++)
        C[i] = A[i] || B[i];
    }

void INTERSEZIONE(insieme A, insieme B, insieme C)
    {
    int i;

    for (i = 0; i < N; i++)
        C[i] = A[i] && B[i];
    }

void DIFFERENZA(insieme A, insieme B, insieme C)
    {
    int i;

    for (i = 0; i < N; i++)
        C[i] = A[i] && !B[i];
    }
```

Esercizio B.4 (Alberi binari). Si scrivano le procedure C per realizzare un albero binario con puntatori, come spiegato nel § 4.

```
typedef int tiponome;

struct bincella
    {
    struct bincella *sinistro, *destro, *genitore;
    tiponome nome;
    };

typedef struct bincella *binalbero, *nodo;

void CREABINALBERO(binalbero *);
char BINALBEROVUOTO(binalbero);
nodo BINRADICE(binalbero);
nodo BINPADRE(nodo, binalbero);
nodo FIGLIOSINISTRO(nodo, binalbero);
nodo FIGLIODESTRO(nodo, binalbero);
char SINISTROVUOTO(nodo, binalbero);
char DESTROVUOTO(nodo, binalbero);
void COSTRBINALBERO(binalbero *, binalbero);
void CANCSOTTOBINALBERO(nodo, binalbero *);
tiponome LEGGINODO(nodo, binalbero);
void SCRIVINODO(tiponome, nodo, binalbero);

void CREABINALBERO(binalbero *T)
    {
    *T = NULL;
    }

char BINALBEROVUOTO(binalbero T)
    {
    return(T == NULL);
    }

nodo BINRADICE(binalbero T)
    {
    return(T);
    }

nodo BINPADRE(nodo u, binalbero T)
    {
    return(u→genitore);
    }

nodo FIGLIOSINISTRO(nodo u, binalbero T)
    {
    return(u→sinistro);
    }
```



```
nodo FIGLIODESTRO(nodo u, binalbero T)
    {
    return(u→destro);
    }

char SINISTROVUOTO(nodo u, binalbero T)
    {
    return(u→sinistro == NULL);
    }

char DESTROVUOTO(nodo u, binalbero T)
    {
    return(u→destro == NULL);
    }

void COSTRBINALBERO(binalbero *T1, binalbero T2)
    {
    nodo u;
    u = (nodo) malloc(sizeof(struct bincella));
    u→sinistro = *T1;
    u→destro = T2;
    u→genitore = NULL;
    if ((*T1) != NULL)
        (*T1)→genitore = u;
    if (T2 != NULL)
        T2→genitore = u;
    *T1 = u;
    }

void CANCSOTTOBINALBERO(nodo u, binalbero *T)
    {
    if (!SINISTROVUOTO(u, *T))
        CANCSOTTOBINALBERO(FIGLIOSINISTRO(u, *T), T);
    if (!DESTROVUOTO(u, *T))
        CANCSOTTOBINALBERO(FIGLIODESTRO(u, *T), T);
    if (u→genitore == NULL)
        *T = NULL;
    else if (u→genitore→sinistro == u)
            u→genitore→sinistro = NULL;
        else
            u→genitore→destro = NULL;
    free(u);
    }

tiponome LEGGINODO(nodo u, binalbero T)
    {
    return(u→nome);
    }
```

```
void SCRIVINODO(tiponome e, nodo u, binalbero T)
    {
    u→nome = e;
    }
```

Esercizio B.5 (Ricerca binaria). Si scriva la versione C della ricerca binaria descritta nel § 6.

```
char RICBIN(tipoelem *V, tipoelem k, int i, int j)
    {
    int m;

    if (i > j)
        return(0);
    else
        {
        m = (i + j)/2;
        if (k == V[m])
            return(1);
        else if (k < V[m])
                return(RICBIN(V, k, i, m − 1));
            else
                return(RICBIN(V, k, m + 1, j));
        }
    }
```

Esercizio B.6 (Quicksort). Si scriva un programma C che legge n interi, li memorizza in un vettore A, lo ordina con l'algoritmo "quicksort" descritto nel § 10, e stampa il vettore ordinato.

```
#include <stdio.h>

void QUICKSORT(int *A, int primo, int ultimo)
    {
    void PERNO(int *, int, int, int *);
    int j;

    if (primo < ultimo)
        {
        PERNO(A, primo, ultimo, &j);
        QUICKSORT(A, primo, j − 1);
        QUICKSORT(A, j + 1, ultimo);
        }
    }
```



```
void PERNO(int *A, int primo, int ultimo, int *j)
    {
    int x, temp, i;

    x = A[primo]; *j = primo;
    for (i = primo + 1; i <= ultimo; i++)
        if (A[i] < x)
            {
            (*j)++;
            temp = A[*j]; A[*j] = A[i]; A[i] = temp;
            }
    A[primo] = A[*j];
    A[*j] = x;
    }

void main()
    {
    void QUICKSORT(int *, int, int);
    int i, A[n];

    for (i = 0; i < n; i++)
        scanf("%d", &A[i]);
    QUICKSORT(A, 0, n − 1);
    for (i = 0; i < n; i++)
        printf("%d ", A[i]);
    printf("\n");
    }
```

Esercizio B.7 (Tutte le permutazioni). Si scriva una procedura C che stampa tutte le $n!$ permutazioni di n elementi interi contenuti in un vettore (cfr. Esercizio 12.4).

```
void PERMUTA(int *A, int primo, int n)
    {
    void SCAMBIA(int *, int *);
    int i;

    if (primo == n)
        {
        for (i = 0; i < n; i++)
            printf("%d", A[i]);
        printf("\n");
        }
    else
        for (i = primo; i < n; i++)
            {
            SCAMBIA(&A[primo], &A[i]);
            PERMUTA(A, primo + 1, n);
            SCAMBIA(&A[primo], &A[i]);
            }
    }
```

```
void SCAMBIA(int *x, int *y)
    {
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
    }
```

Bibliografia

La presente bibliografia è suddivisa per capitoli. Per ogni capitolo, sono elencati sia libri di testo, specificando se si tratta di testi introduttivi o specialistici, sia articoli di riviste scientifiche. I testi introduttivi sono adatti sia a studenti del Corso di Laurea in Informatica che a lettori che abbiano una preparazione matematica a livello di scuola superiore. I testi specialistici, alcuni dei quali sono adottati come libri di testo nell'ambito di corsi universitari avanzati, sono invece adatti a studenti del successivo Corso di Laurea Specialistica in Informatica, o comunque a lettori più maturi. Gli articoli di riviste scientifiche sono articoli di ricerca che rappresentano la fonte originale degli argomenti trattati nel presente testo oppure che forniscono materiale per ulteriori approfondimenti e complementi. Non essendo stati generalmente scritti a fini didattici, presuppongono spesso la conoscenza di risultati specifici, e sono per questo più adatti a studenti del Dottorato di Ricerca. Tali articoli sono stati inseriti ugualmente perché è opinione dell'autore che tutti si debbano pian piano abituare a leggere direttamente le fonti originali, con spirito critico e costruttivo. Inoltre, sono stati inclusi anche alcuni articoli di ricerca dell'autore del presente testo, affinché gli studenti interessati possano orientarsi sugli argomenti per possibili tesi di laurea.

Introduzione

A. Aho, J. Hopcroft e J. Ullman (1983), *Data Structures and Algorithms*, Addison-Wesley, Reading, MA. Ottimo testo introduttivo dedicato alle strutture di dati e al progetto e all'analisi di algoritmi.

S. Baase (1988), *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, MA (seconda edizione). Buon testo introduttivo, breve e chiaro, su progetto e analisi di algoritmi.

T.H. Cormen, C.E. Leiserson e R.L. Rivest (1990), *Introduction to Algorithms*, MIT Press / McGraw-Hill, New York. Testo enciclopedico rivolto al progetto e all'analisi di algoritmi ed alle strutture di dati. Descrive e analizza anche gli "heap binomiali" e gli "heap di Fibonacci", atti ad effettuare l'operazione di UNIONE tra due heap.

D. Harel (1987), *Algorithmics: The Spirit of Computing*, Addison-Wesley, Reading, MA. Eccellente opera introduttiva che affronta tutti gli aspetti dell'algoritmica.

E. Horowitz, S. Sahni e D. Mehta (1995), *Fundamental of Data Structures in C++*, Computer Science Press, New York. Opera specialistica a carattere enciclopedico sulle strutture di dati, con procedure in C++.

E. Horowitz, S. Sahni e S. Rajasekaran (1997), *Computer Algorithms*, Computer Science Press, New York. Opera enciclopedica su progetto e analisi di algoritmi.

D. Knuth (1968, 1969, 1973), *The Art of Computer Programming* (3 volumi), Addison-Wesley, Reading, MA. Un'opera enciclopedica fondamentale sull'algoritmica, in particolare, e sull'intera informatica, in generale.

F. Luccio (1982), *La Struttura degli Algoritmi*, Boringhieri, Torino. Breve libro introduttivo in italiano, dedicato specialmente alla tecnica "divide et impera" ed alla valutazione della complessità di algoritmi ricorsivi.

R. Sedgewick (1988), *Algorithms*, Addison-Wesley, Reading, MA (seconda edizione). Ottimo testo introduttivo contenente programmi Pascal per i principali problemi computazionali. Ne esistono anche altre due edizioni successive con programmi in C e C++.

J. Welsh e J. Elder (1987), *Introduzione al Pascal*, ESA, Roma (seconda edizione). Buon manuale in italiano per apprendere rapidamente il linguaggio Pascal.

1 Tipi di dato e strutture di dati

E. Lodi e G. Pacini (1990), *Introduzione alle Strutture di Dati*, Boringhieri, Torino. Buona trattazione in italiano su specifica e algebra di strutture di dati.

D. Stubbs e N. Webre (1987), *Data Structures with Abstract Data Types and Modula-2*, Brooks/Cole, Monterey, CA. Introduce le strutture di dati specificandole come tipi di dato e utilizzando il linguaggio Modula-2, che è particolarmente adatto allo scopo.

2 Liste, 3 Pile e code, 4 Alberi

D. Knuth (1968), *The Art of Computer Programming; Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA. Contiene numerosissimi riferimenti sulle origini delle strutture elementari di dati, come vettori, liste, pile, code e alberi.

5 Insiemi

J. van Leeuwen e R.E. Tarjan (1984), "Worst-case Analysis of Set Union Algorithms", *Journal of the ACM* 7, pp. 245-281. Analizza svariate tecniche per realizzare un MFSET.

6 Dizionari

D. Knuth (1973), *The Art of Computer Programming; Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA. Testo di riferimento per l'analisi delle tabelle hash.

J. Ullman (1982), *Principles of Database Systems*, Computer Science Press, Rockville, MD. Testo rivolto al progetto e alla realizzazione di basi di dati e di dizionari su memoria secondaria.

7 Code con priorità

M.R. Brown (1978), "Implementation and Analysis of Binomial Queue Algorithms", *SIAM Journal on Computing* 7, pp. 298-319. Studia in dettaglio le proprietà degli "heap binomiali" introdotti da Vuillemin.

M.L. Fredman e R.E. Tarjan (1987), "Fibonacci Heaps and Their Use in Improved Network Optimization Problems", *Journal of the ACM* 34, pp. 596-615. Propone gli "heap di Fibonacci" per realizzare code con priorità. È ammessa anche l'operazione UNIONE, che richiede O(1) tempo ammortizzato.

J. Vuillemin (1978), "A Data Structure for Manipulating Priority Queues", *Communications of the ACM* 21, pp. 309-315. Introduce gli "heap binomiali" per realizzare code con priorità. È ammessa anche l'operazione UNIONE, che richiede $O(\log n)$ tempo nel caso pessimo.

J.W.J. Williams (1964), "Algorithm 232 (Heapsort)", *Communications of the ACM* 7, pp. 347-348. Propone per la prima volta sia la realizzazione di una coda con priorità con uno "heap" che l'algoritmo "heapsort".

8 Alberi bilanciati di ricerca

A. Aho, J. Hopcroft e J. Ullman (1983), *Data Structures and Algorithms*, Addison-Wesley, Reading, MA. Tratta anche gli alberi 2-3 e 2-3-4. Gli alberi 2-3, precursori dei B-alberi, furono ideati da Hopcroft nel 1970, ma non pubblicati.

A. Andersson (1996), "Balanced Search Trees Made Simple", *Lecture Notes in Computer Science*, pp. 60-71. Presenta la realizzazione dei B-alberi binari con le operazioni INCLINA e FRAZIONA.

R. Bayer (1972), "Symmetric Binary B-Trees: Data Structures and Maintainance Algorithms", *Acta Informatica* 1, pp. 290-306. Propone i B-alberi binari.

R. Bayer e E. McCreight (1972), "Organization and Maintainance of Large Ordered Indices", *Acta Informatica* 1, pp. 173-189. Introduce per la prima volta i B-alberi.

9 Grafi

C. Berge (1973), *Graphs and Hypergraphs*, North-Holland, Amsterdam. Testo classico su grafi e ipergrafi, trattati da un punto di vista squisitamente matematico.

M.C. Golumbic (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York. Testo introduttivo su algoritmi per grafi "perfetti", che includono i grafi di intervalli come caso particolare.

T. Nishizeki e N. Chiba (1988), *Planar Graph: Theory and Algorithms*, North-Holland, Amsterdam. Testo su teoria ed algoritmi per grafi "planari".

F. Roberts (1978), *Graph Theory and its Applications to Problems of Society*, SIAM, Philadelphia, PA. Buon testo introduttivo sulle innumerevoli applicazioni dei grafi nei problemi della vita quotidiana.

10 Strutture di dati e progetto di algoritmi

R.E. Bellman (1958), "On a Routing Problem", *Quarterly Applied Mathematics* 16, pp. 87-90. Introduce le condizioni di ottimalità per i CAMMINI MINIMI e fornisce un algoritmo generale che è stato scoperto anche da Ford e da Moore.

A.A. Bertossi e A. Gori (1988), "Total Domination and Irreduncance in Weighted Interval Graphs", *SIAM Journal on Discrete Mathematics* 1, pp. 317-327. Formula il problema dell'"insieme dominante" su un grafo di intervalli come problema di CAMMINI MINIMI e lo risolve usando una struttura di dati simile ad uno "heap".

E.W. Dijkstra (1959), "A Note on Two Problems in Connection with Graphs", *Numerische Mathematik* 1, pp. 269-271. Fornisce l'algoritmo per risolvere il problema dei CAMMINI MINIMI per grafi con lunghezze non negative sugli archi.

G. Gallo e S. Pallottino (1988), "Shortest Paths Algorithms", *Annals of Operations Research* 13, pp. 3-79. Buon articolo di rassegna su algoritmi e strutture di dati per il problema dei CAMMINI MINIMI.

J.E. Hopcroft e R.E. Tarjan (1974), "Efficient Planarity Testing", *Journal of the ACM* 21, pp. 549-568. Presenta un algoritmo di complessità ottima per verificare la planarità di un grafo.

D.B. Johnson (1977), "Efficient Algorithms for Shortest Paths in Sparse Networks", *Journal of the ACM* 24, pp. 1-13. Rende più efficiente l'algoritmo di Dijkstra per grafi sparsi.

U. Pape (1974), "Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem", *Mathematical Programming* 7, pp. 212-222. In pratica l'algoritmo di CAMMINI MINIMI più efficiente per grafi sparsi planari.

11 Divide et impera

J.W. Cooley e J.W. Tukey (1965), "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation* 19, pp. 297-301. Introduce l'algoritmo FFT.

C.A.R. Hoare (1962), "Quicksort", *Computer Journal* 5, pp. 10-15. Presenta per la prima volta il "quicksort", suggerendo anche tutte le principali varianti.

D. Knuth (1973), *The Art of Computer Programming; Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA. Un trattato enciclopedico sugli algoritmi di ordinamento.

R. Sedgewick (1978), *Quicksort*, Garland, New York. Libro interamente dedicato al "quicksort". Include l'analisi di complessità e gli effetti pratici di numerose varianti.

12 Backtrack

R.L. Graham (1972), "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set", *Information Processing Letters* 1, pp. 132-133. Presenta l'algoritmo di complessità ottima per l'INVILUPPO CONVESSO.

D.E. Knuth, J.H. Morris e V.R. Pratt (1977), "Fast Pattern Matching in Strings", *SIAM Journal on Computing* 6, pp. 323-350. Introduce l'algoritmo di complessità ottima per lo STRING MATCHING.

F. Preparata e M. Shamos (1985), *Computational Geometry: An Introduction*, Springer-Verlag, New York. Ottimo libro su algoritmi per problemi di geometria computazionale, incluso l'INVILUPPO CONVESSO.

13 Greedy

E.L. Lawler (1976), *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York. Include un'ampia trattazione dei matroidi.

J.B. Kruskal (1956), "On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem", *Proceedings of the AMS* 7, pp. 48-50. Introduce l'algoritmo per il MINIMO ALBERO DI COPERTURA che porta il suo nome.

J.M. Moore (1968), "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs", *Management Science* 15, pp. 102-109. Introduce l'algoritmo per il problema dei PROGRAMMI IN RITARDO.

R.C. Prim (1957), "Shortest Connection Networks and Some Generalizations", *Bell System Technical Journal* 36, pp. 1389-1401. Introduce un algoritmo per il MINIMO ALBERO DI COPERTURA molto simile all'algoritmo di Dijkstra.

H. Whitney (1935), "On the Abstract Properties of Linear Dependence", *American Journal of Mathematics* 57, pp. 509-533. Definisce la teoria dei matroidi.

14 Programmazione dinamica

R. Bellman (1957), *Dynamic Programming*, Princeton University Press. Testo scritto dall'inventore stesso della "programmazione dinamica".

A.A. Bertossi, E. Lodi, F. Luccio e L. Pagli (1990), "String Matching with Weighted Errors", *Theoretical Computer Science* 73, pp. 319-328. Estende l'algoritmo di Landau e Vishkin comprendendo errori di costo arbitrario anziché unitario.

R. Floyd (1962), "Algorithm 92 (Shortest Path)", *Communications of the ACM* 5, pp. 345. Propone l'algoritmo per i CAMMINI MINIMI tra tutte le coppie, basandosi su un teorema di Warshall pubblicato nello stesso anno.

Z. Galil e R. Giancarlo (1988), "Data Structures and Algorithms for Approximate String Matching", *Journal of Complexity* 4, pp. 33-72. Articolo di rassegna sullo STRING MATCHING APPROSSIMATO.

G.M. Landau e U. Vishkin (1988), "Fast String Matching with k Differences", *Journal of Computer and System Sciences* 37, pp. 63-78. Risolve molto efficientemente una variante dello STRING MATCHING APPROSSIMATO.

15 Ricerca locale

G.B. Dantzig (1963), *Linear Programming and Extensions*, Princeton University Press. Testo scritto da chi ha inventato nel 1947 l'algoritmo del "simplesso" per la PROGRAMMAZIONE LINEARE.

V.M. Malhotra, M.P. Kumar e S.N. Maheshwari (1978), "An $O(n^3)$ Algorithm for Finding Maximum Flows in Networks", *Information Processing Letters* 7, pp. 277-278. Introduce il cosiddetto algoritmo dei "3 indiani" (dalla nazionalità degli autori), basandosi su risultati precedenti di Dinic e Karzanov.

C. Papadimitriou e K. Steiglitz (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ. Ottimo testo su algoritmi di ottimizzazione, dedicato alla PROGRAMMAZIONE LINEARE e a problemi su grafi, come FLUSSO MASSIMO, CAMMINI MINIMI, e ABBINAMENTO.

D.L. Shell (1959), "A High-speed Sorting Procedure", *Communications of the ACM* 2, pp. 30-32. Inventa l'algoritmo "shellsort".

R.E. Tarjan (1983), *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA. Testo specialistico su strutture di dati e algoritmi per problemi di ottimizzazione su grafi, inclusi il FLUSSO MASSIMO e i CAMMINI MINIMI.

16 Analisi di algoritmi

J.L. Bentley, D. Haken e J.B. Saxe (1980), "A General Method for Solving Divide-and-Conquer Recurrences", SIGACT *News* 12, pp. 36-44. Propone un metodo generale per risolvere relazioni di ricorrenza.

A.A. Bertossi (1999), "Online operation", in: *Encyclopedia of Electrical and Electronics Engineering* (a cura di J.G. Webster), Wiley, New York, vol. 15, pp. 146-150. Articolo divulgativo sugli algoritmi "on line".

D. Bini, M. Capovani, G. Lotti e F. Romani (1981), *Complessità Numerica*, Boringhieri, Torino. Buon testo introduttivo in italiano su algoritmi e complessità per problemi "numerici" (come prodotto di matrici, interpolazione, integrazione, ecc.).

M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest e R.E. Tarjan (1973), "Time Bounds for Selection", *Journal of Computer and System Sciences* 7, pp. 448-461. È proposto un algoritmo di complessità lineare nel caso pessimo per la SELEZIONE.

C.A.R. Hoare (1961), "Algorithm 63 (Partition) and Algorithm 65 (Find)", *Communications of the ACM* 4, pp. 321-322. Propone l'algoritmo di complessità lineare nel caso medio per la SELEZIONE.

R. Sedgewick e P. Flajolet (1996), *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA. Testo sulle tecniche di analisi della complessità di algoritmi, comprese le cosiddette "funzioni generatrici".

D.D. Sleator e R.E. Tarjan (1985), "Amortized Efficiency of List Update and Paging Rules", *Communications of the ACM* 28, pp. 202-208. Descrive e analizza l'algoritmo "Move to Front".

V. Strassen (1969), "Gaussian Elimination is not Optimal", *Numerische Matematik* 14, pp. 354-356. Il primo algoritmo per la MOLTIPLICAZIONE DI MATRICI con complessità inferiore ad $O(n^3)$.

R.E. Tarjan (1985), "Amortized Computational Complexity", *SIAM Journal on Algebraic and Discrete Methods* 6, pp. 306-318. Inventa l'analisi ammortizzata.

17 Non determinismo ed enumerazione

J. Cohen (1979), "Non-Deterministic Algorithms", *ACM Computing Surveys* 11, pp. 79-94. Articolo divulgativo sugli algoritmi non deterministici.

18 Problemi NP-completi

A. Cobham (1964), "The Intrinsic Computational Difficulty of Functions", *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pp. 24-30. Introduce per la prima volta la classe $\mathbb{P}$.

S. Cook (1971), "The Complexity of Theorem Proving Procedures", *Proceedings of the 27th Annual Symposium on Foundation of Computer Science*, pp. 151-158. Introduce per la prima volta, indipendentemente da Levin, la nozione di NP-completezza e dimostra che la SODDISFATTIBILITÀ è NP-completa.

J. Edmonds (1965), "Paths, Trees, and Flowers", *Canadian Journal of Mathematics* 17, pp. 449-467. Introduce le classi $\mathbb{P}$ ed $\mathbb{NP}$, congetturando $\mathbb{P} \neq \mathbb{NP}$.

M.R. Garey e D.S. Johnson (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA. Il primo e miglior testo enciclopedico sull'NP-completezza, contenente un'appendice con centinaia di problemi NP-completi.

R.M. Karp (1972), "Reducibility among Combinatorial Problems", in: *Complexity of Computer Computations* (a cura di R.E. Miller e J.W. Thatcher), Plenum Press, New York, pp. 85-103. Prosegue il lavoro di Cook, mostrando l'NP-completezza di svariati problemi con riduzioni dalla SODDISFATTIBILITÀ.

L. Levin (1972), "Universal Sorting Problems", *Problemy Peredachi Informatsii* 9, pp. 265-266 (in russo). Definisce, indipendentemente da Cook, la nozione di NP-completezza e dimostra che il DOMINO LIMITATO è NP-completo.

19 Gerarchia di complessità

G. Ausiello (1975), *Complessità di Calcolo delle Funzioni*, Boringhieri, Torino. Testo specialistico in italiano sulla complessità astratta.

D.P. Bovet e P. Crescenzi (1991), *Teoria della Complessità Computazionale*, Angeli, Milano. Buon testo introduttivo in italiano sulle classi di complessità.

R.E. Ladner (1975), "On the Structure of Polynomial-Time Reducibility", *Journal of the ACM* 22, pp. 155-171. Prova l'esistenza di problemi NP-intermedi, qualora $\mathbb{P} \neq \mathbb{NP}$.

G.L. Miller (1976), "Riemann's Hypothesis and Tests for Primality", *Journal of Computer and System Sciences* 13, pp. 300-317. Fornisce un algoritmo polinomiale deterministico per la PRIMALITÀ nel caso sia vera l'"ipotesi estesa di Riemann".

C.H. Papadimitriou (1995), *Computational Complexity*, Addison-Wesley, Reading, MA. Ottimo testo specializzato sulla complessità computazionale.

L. Stockmeyer (1976), "The Polynomial-Time Hierarchy", *Theoretical Computer Science* 3, pp. 1-22. Definisce la gerarchia polinomiale.

L. Stockmeyer e A.K. Chandra (1978), "Provably Difficult Combinatorial Games", Technical Report No. RC-6957, IBM T.J. Watson Research Center, Yorktown Heights, NY. Tratta la complessità di svariati giochi di natura combinatoria.

L. Stockmeyer e A.R. Meyer (1973), "Word Problems Requiring Exponential Time", *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pp. 1-19. Dimostra che QBF è PSPAZIO-completa.

C. Wrathall (1976), "Complete Sets and Polynomial-Time Hierarchy", *Theoretical Computer Science* 3, pp. 23-33. Caratterizza la gerarchia polinomiale così come è stata presentata in questo testo.

20 Algoritmi probabilistici

L. Babai (1979), "Monte-Carlo Algorithms in Graph Isomorphism Testing", Technical Report No. DMS-79-10, Département de Mathématique et Statistique, Université de Montreal. Distingue per la prima volta tra Monte Carlo e Las Vegas.

J.T. Gill (1977), "Computational Complexity of Probabilistic Turing Machines", *SIAM Journal on Computing* 6, pp. 675-695. Introduce le classi probabilistiche di complessità.

M.O. Rabin (1976), "Probabilistic Algorithms", in: *Algorithms and Complexity* (a cura di J. Traub), Academic Press, New York, pp. 21-39. Inventa gli algoritmi probabilistici.

M.O. Rabin (1980), "Probabilistic Algorithm fot Testing Primality", *Journal of Number Theory* 12, pp. 128-138. Presenta l'algoritmo probabilistico per la PRIMALITÀ.

R. Seidel e C.R. Aragon (1996), "Randomized Search Trees", *Algorithmica*, pp. 464-497. Presenta gli alberi randomizzati di ricerca.

21 Algoritmi pseudo-polinomiali

A.A. Bertossi, L.V. Mancini e F. Rossini (1999), "Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems", *IEEE Transactions on Parallel and Distributed Systems* 10, pp 934-945. Generalizza "Rate-Monotonic" e CTT per tollerare processori guasti.

M.R. Garey e D.S. Johnson (1978), "Strong NP-Completeness Results: Motivations, Examples, and Implications", *Journal of the ACM* 25, pp. 499-508. Definisce la nozione di NP-completezza "forte".

M. Joseph e P. Pandya (1986), "Finding Response Times in a Real-Time System", *Computer Journal* 29, pp. 390-395. Presenta l'algoritmo CTT.

M.H. Klein, J.P. Lehoczky e R. Rajkumar (1994), "Rate-Monotonic Analysis for Real-Time Industrial Computing", *IEEE Computer*, pp. 24-33. Articolo di rassegna sulla tecnica "Rate-Monotonic" e sue varianti.

C.L. Liu e J.W. Layland (1973), "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM* 20, pp. 46-61. Inventa l'algoritmo "Rate-Monotonic".

22 Algoritmi di approssimazione

N. Christofides (1976), "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem", Technical Report GSIA, Carnagie Mellon University, Pittsburgh, PA. Propone l'algoritmo con scarto 50% dal minimo per il ΔTSP.

M.R. Garey e D.S. Johnson (1978), "Strong NP-Completeness Results: Motivations, Examples, and Implications", *Journal of the ACM* 25, pp. 499-508. Dimostra tra l'altro il Teorema 22.2.

R.L. Graham (1969), "Bounds on Multiprocessor Timing Anomalies", *SIAM Journal on Applied Mathematics* 17, pp. 416-429. Dimostra l'approssimazione per l'algoritmo LPT.

O.H. Ibarra e C.E. Kim (1975), "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems", *Journal of the ACM* 22, pp. 463-468. Propone algoritmi di approssimazione per la SOMMA DI SOTTOINSIEME e lo ZAINO.

D.J. Rosenkrantz, R.E. Stearns e P.M. Lewis (1977), "An Analysis of Several Heuristics for the Travelling Salesman Problem", *SIAM Journal on Computing* 6, pp. 563-581. Tratta anche l'algoritmo con scarto 100% dal minimo per il ΔTSP.

S. Sahni e T. Gonzalez (1976), "P-complete Approximation Problems", *Journal of the ACM* 23, pp. 555-565. Dimostra che il TSP non è approssimabile (Teorema 22.1).

23 Algoritmi branch-&-bound

A.A. Bertossi, G. Bongiovanni e M.A. Bonuccelli (1987), "Time Slot Assignment in SS/TDMA Systems with Intersatellite Links", *IEEE Transactions on Communications* 35, pp. 602-608. Propone l'algoritmo branch-&-bound per lo SCHEDULING ISL.

M. Held e R.M. Karp (1971), "The Travelling Salesman Problem and Minimum Spanning Trees: Part II", *Mathematical Programming* 1, pp. 6-25. Articolo sull'utilizzo di 1-alberi modificati in algoritmi branch-&-bound per il TSP (è il seguito della "Parte I" pubblicata l'anno precedente su *Operations Research* 18, pp. 1138-1162).

E.L. Lawler e D.E. Wood (1966), "Branch-and-Bound Methods: A Survey", *Operations Research* 14, pp. 699-719. Articolo di rassegna sul branch-&-bound con numerosi riferimenti "storici".

C. Papadimitriou e K. Steiglitz (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ. Tratta anche il branch-&-bound, sia per la PROGRAMMAZIONE LINEARE INTERA che per problemi su grafi.

24 Algoritmi euristici

R. Battiti e A.A. Bertossi (1999), "Greedy, Prohibition, and Reactive Heuristics for Graph Partitioning", *IEEE Transactions on Computers* 48, pp. 361-385. Propone l'euristica di ricerca tabù reattiva e randomizzata per la BISEZIONE.

R. Battiti e G. Tecchiolli (1994), "The Reactive Tabu Search", *ORSA Journal on Computing* 6, pp. 126-140. Inventa la ricerca tabù reattiva.

F. Glover (1989), "Tabu Search: Part I", *ORSA Journal on Computing* 1, pp. 190-260. Inventa la ricerca tabù.

25 Indecidibilità ed universalità, 26 Macchine di Turing

E. Börger (1988), *Computabilità, Complessità, Logica*, Boringhieri, Torino. Opera specialistica in italiano su modelli di calcolo, computabilità e complessità astratta.

H. Lewis e C. Papadimitriou (1981), *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ. Ottimo testo su modelli di calcolo, intrattabilità e indecidibilità.

H. Rogers (1967), *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York. Un testo classico su funzioni calcolabili e indecidibilità, inclusa la gerarchia di Kleene. La trattazione è specialistica e piuttosto formale.

27 Algoritmi PRAM

S. Fortune e J.C. Wyllie (1978), "Parallelism in Random Access Machines", *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pp. 114-118. Introduce il modello PRAM.

L. Goldschlager, R. Shaw e J. Staples (1982), "The Maximum Flow Problem is Log-Space-Complete for $\mathbb{P}$", *Theoretical Computer Science* 21, pp. 105-111. Dimostra che il FLUSSO MASSIMO è LOG-SPAZIO-completo.

J. Ja'Ja' (1992), *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA. Testo dedicato agli algoritmi paralleli per modelli PRAM.

R.M. Karp e V. Ramachandran (1990), "Parallel Algorithms for Shared-Memory Machines", in: *Handbook of Computer Science*, vol. A: *Algorithms and Complexity* (a cura di J. van Leeuwen), Elsevier, Amsterdam. Articolo di rassegna su algoritmi PRAM.

R.E. Tarjan e U. Vishkin (1985), "An Efficient Parallel Biconnectivity Algorithm", *SIAM Journal on Computing* 14, pp. 862-874. Introduce la tecnica del "ciclo Euleriano".

J.C. Wyllie (1979), "The Complexity of Parallel Computations", PhD Thesis, Department of Computer Science, Cornell University. Introduce la tecnica del "salto dei puntatori".

28 Algoritmi per reti a grado limitato

K.E. Batcher (1968), "Sorting Networks and Their Applications", *Proceedings of the Spring AFIPS Joint Computer Conference*, pp. 307-314. Propone il "mergesort bitonico".

E. Dekel, D. Nassimi, e S. Sahni (1981), "Parallel Matrix and Graph Algorithms", *SIAM Journal on Computing* 10, pp. 657-675. Illustra l'algoritmo di MOLTIPLICAZIONE DI MATRICI su un ipercubo.

Z. Galil e W.J. Paul (1983), "An Efficient General Purpose Parallel Computer", *Journal of the ACM* 30, pp. 360-387. Descrive un calcolatore parallelo "universale".

F.T. Leighton (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Francisco, CA. Testo rivolto al progetto e all'analisi di algoritmi paralleli per reti a grado limitato.

F. Preparata e J. Vuillemin (1981), "The Cube-Connected-Cycles: A Versatile Network for Parallel Computation", *Communications of the ACM* 24, pp. 300-309. Definisce il "CCC".

H.S. Stone (1971), "Parallel Processing with the Perfect Shuffle", *IEEE Transactions on Computers* 28, pp. 907-917. Definisce la "shuffle".

C.D. Thompson e H.T. Kung (1977), "Sorting on a Mesh-Connected Parallel Computer", *Communications of the ACM* 20, pp. 263-271. Contiene l'adattamento del "bitonic-mergesort" ad una "mesh".

C.D. Thompson e C.G. Plaxton (1998), "Hypercubic Sorting Networks", *SIAM Journal on Computing* 27, pp. 1-47. Articolo recente sull'ORDINAMENTO con un ipercubo, con riferimenti bibliografici aggiornati, p.e., ad un algoritmo ottimo di complessità $O(\log n \log\log n)$.

29 Algoritmi VLSI

M. Ajtai, J. Komlós e E. Szemerédi (1983), "An $O(n \log n)$ Sorting Network", *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pp. 1-9. Mostra l'esistenza di una rete, detta "AKS", per ordinare in tempo $O(\log n)$. Le enormi costanti coinvolte nell'ordine di grandezza non la rendono però utilizzabile in pratica.

A.A. Bertossi e M.A. Bonuccelli (1989), "A gracefully degradable VLSI system for linear programming", *IEEE Transactions on Computers* 38, pp. 853-861. Propone varianti del layout ad "H" per tollerare guasti.

G. Bilardi (1984), "The Area-Time Complexity of Sorting", PhD Thesis, University of Illinois at Urbana Champaign. Studia la complessità dell'ORDINAMENTO nel modello VLSI.

C. Mead e L. Conway (1980), *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA. Testo su sistemi VLSI, più orientato agli aspetti elettronici e meno a quelli algoritmici rispetto al testo di Ullman.

F. Preparata e J. Vuillemin (1980), "Area-Time Optimal VLSI Networks for Multiplying Matrices", *Information Processing Letters* 11, pp. 77-80. Propone l'algoritmo per moltiplicare matrici in tempo $O(\log n)$ e AT^2 ottima.

C.D. Thompson (1980), "A Complexity Theory for VLSI", PhD Thesis, Carnagie-Mellon University, Pittsburgh. Introduce il modello a "griglia".

J.D. Ullman (1983), *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD. Testo specialistico su algoritmi e complessità nel modello VLSI.

30 Algoritmi per mesh riconfigurabili

Y. Ben-Asher, D. Peleg, R. Ramaswani e A. Shuster (1992), "The Power of Reconfiguration", *Journal of Parallel and Distributed Computing* 13, pp. 139-153. Propone e analizza il modello con mesh riconfigurabili.

A.A. Bertossi e A. Mei (1998), "A Residue Number System on Reconfigurable Mesh with Applications to Prefix Sums and Approximate String Matching", *IEEE Transactions on Parallel and Distributed Systems* (in stampa). Propone algoritmi molto efficienti per le SOMME PREFISSE.

A.A. Bertossi e A. Mei (1999), "Constant Time Dynamic Programming on Reconfigurable Networks", *IEEE Transactions on Parallel and Distributed Systems* (in stampa). Risolve in tempo $O(1)$ alcuni schemi di programmazione dinamica.

J. Jang, H. Park e V.K. Prasanna (1997), "An Optimal Multiplication Algorithm on Reconfigurable Mesh", *IEEE Transactions on Parallel and Distributed Systems* 8, pp. 521-532. Propone un algoritmo ottimo per la MOLTIPLICAZIONE.

T. Leighton (1985), "Tight Bounds on the Complexity of Parallel Sorting", *IEEE Transactions on Computers* 34, pp. 344-354. Propone l'algoritmo "column-sort".

R. Miller, V.K. Prasanna, D.I. Reisis e Q.F. Stout (1993), "Parallel Computations on Reconfigurable Meshes", *IEEE Transactions on Computers* 42, pp. 678-692. Propone e analizza il modello con mesh riconfigurabili.

M. Nigam and S. Sahni (1994), "Sorting n numbers on $n \times n$ reconfigurable meshes with buses", *Journal of Parallel and Distributed Computing* 23, pp.37-48. Adatta l'algoritmo "column-sort" alle mesh riconfigurabili.

31 Algoritmi concorrenti

G.M. Baudet (1978), "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", PhD Thesis, Carnagie-Mellon University, Pittsburgh. Tratta il progetto e l'analisi di algoritmi concorrenti.

H.T. Kung (1976), "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", in: *Algorithms and Complexity* (a cura di J. Traub), Academic Press, New York, pp. 153-200. Articolo di rassegna su algoritmi paralleli che tratta anche di algoritmi concorrenti.

P.N. Oleinik (1982), *Parallel Algorithms on a Multiprocessor*, UMI Research Press, Ann Arbor. Testo introduttivo su algoritmi concorrenti.

M. Quinn (1987), *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York. Buon testo introduttivo su algoritmi paralleli, che tratta anche di algoritmi concorrenti.

32 Algoritmi distribuiti

A.A. Bertossi e M.A. Bonuccelli (1995), "Code assignment for hidden terminal interference avoidance in multihop packet radio networks", *IEEE/ACM Transactions on Networking* 3, pp. 441-449. Dà una euristica greedy distribuita per la COLORAZIONE.

E.G. Chang e R. Roberts (1979), "An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processors", *Communications of the ACM* 22, pp. 281-283. Propone l'algoritmo per l'elezione del leader in un anello unidirezionale con complessità $O(n^2)$ nel caso pessimo.

D. Dolev, M. Klawe e M. Rodeh (1982), "An $O(n \log n)$ Distributed Algorithm for Extrema-Finding in a Circle", *Journal of Algorithms* 3, pp. 245-260. Propone l'algoritmo per l'elezione del leader in un anello unidirezionale con complessità $O(n \log n)$ nel caso pessimo.

R.G. Gallager, P.A. Humblet e P.M. Spira (1983), "A Distributed Algorithm for Minimum Spanning Tree", *ACM Transactions on Programming Languages and Systems* 3, pp. 66-77. Dà l'algoritmo per il MINIMO ALBERO DI COPERTURA.

D.S. Hirschberg e J.B. Sinclair (1980), "Decentralized Extrema-Finding in Circular Configurations of Processors", *Communications of the ACM* 23, pp. 627-628. Propone l'algoritmo per l'elezione del leader in un anello bidirezionale.

N. Lynch (1996), *Distributed Algorithms*, Morgan Kaufmann, San Francisco, CA. Testo enciclopedico su progetto e analisi di algoritmi distribuiti.

M. Raynal (1988), *Distributed Algorithms and Protocols*, Wiley, New York. Buon testo introduttivo su progetto di algoritmi distribuiti.

G. Ricart e A.K. Agrawala (1981), "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM* 24, pp. 9-17. Propone l'algoritmo per la mutua esclusione in un grafo completo. Alcune imprecisioni sono corrette nel 1983 su *Communications of the ACM* 26, pp. 147-148.

Conclusione

R.S. Boyer e J.S. Moore (1977), "A Fast String-Matching Algorithm", *Communications of the ACM* 20, pp. 762-772. In pratica il miglior algoritmo per lo STRING-MATCHING, insieme a quello di Karp-Rabin.

J.W. Cooley e J.W. Tukey (1965), "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation* 19, pp. 297-301. Algoritmo FFT ottimo per la TRASFORMATA DISCRETA DI FOURIER.

D. Coppersmith e S. Winograd (1987), "Matrix Multiplication Via Arithmetic Progression", *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pp. 1-6. In teoria il miglior algoritmo per la MOLTIPLICAZIONE DI MATRICI, ma non utilizzabile in pratica a causa delle enormi costanti coinvolte nell'ordine di grandezza.

M.L. Fredman e R.E. Tarjan (1987), "Fibonacci Heaps and Their Use in Improved Network Optimization Problems", *Journal of the ACM* 34, pp. 596-615. I migliori algoritmi per i CAMMINI MINIMI e il MINIMO ALBERO DI COPERTURA.

A.V. Goldberg e R.E. Tarjan (1986), "A New Approach to the Maximum Flow Problem", *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pp. 136-146. Miglior algoritmo per il FLUSSO MASSIMO.

R.L. Graham (1972), "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set", *Information Processing Letters* 1, pp. 132-133. Algoritmo di complessità ottima per l'INVILUPPO CONVESSO.

C.A.R. Hoare (1962), "Quicksort", *Computer Journal* 5, pp. 10-15. In pratica il miglior algoritmo di ordinamento per sequenze "lunghe".

N. Karmarkar (1984), "A New Polynomial-Time Algorithm for Linear Programming", *Combinatorica* 4, pp. 373-395. In teoria il miglior algoritmo per la PROGRAMMAZIONE LINEARE, ma l'algoritmo del "simplesso" resta competitivo in pratica.

R.M. Karp e M.O. Rabin (1981), "Efficient Randomized Pattern-Matching Algorithms", Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University. In pratica il miglior algoritmo per lo STRING-MATCHING, insieme a quello di Boyer-Moore.

D.E. Knuth, J.H. Morris, e V.R. Pratt (1977), "Fast Pattern Matching in Strings", *SIAM Journal on Computing* 6, pp. 323-350. Algoritmo di complessità ottima per lo STRING MATCHING, anche se in pratica è battuto da quelli di Boyer-Moore e di Karp-Rabin.

G.M. Landau e U. Vishkin (1988), "Fast String Matching with k Differences", *Journal of Computer and System Sciences* 37, pp. 63-78. Il miglior algoritmo per una importante variante dello STRING MATCHING APPROSSIMATO.

S. Micali e V.V. Vazirani (1980), "An $O(m\sqrt{n})$ Algorithm for Finding Maximum Matching in General Graphs", *Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science*, pp. 17-27. Miglior algoritmo per l'ABBINAMENTO in un grafo qualsiasi (non necessariamente bipartito).

D.L. Shell (1959), "A High-speed Sorting Procedure", *Communications of the ACM* 2, pp. 30-32. In pratica il miglior algoritmo di ordinamento per sequenze "corte".

J.W.J. Williams (1964), "Algorithm 232 (Heapsort)", *Communications of the ACM* 7, pp. 347-348. Algoritmo ottimo in ordine di grandezza, anche se in pratica è battuto da "quicksort" e "shellsort".

A Numeri casuali

S.K. Park e K.W. Miller (1988), "Random Number Generators: Good Ones Are Hard to Find", *Communications of the ACM* 31, pp. 1192-1201. Consigli utili per progettare un generatore di numeri pseudo-casuali.

B Il linguaggio C

B.W. Kernighan e D.M. Ritchie (1978), *Linguaggio C*, Jackson, Milano. Buon manuale in italiano per apprendere rapidamente il linguaggio C.

Indice analitico

ABBINAMENTO, 235
  3-DIMENSIONALE, 276
  BIPARTITO, 236
Addizionatore, 408
Adiacenza, 139
AGGIORNAMENTO, 249
Agglomerato, 97
Albero
  2-3, 121
  bilanciato di ricerca, 118
  binario, 71
  binario di ricerca, 119
  delle scelte, 261
  di cammini, 157
  di copertura DFS, 146
  di decisione, 72
  di derivazione, 62
  genealogico, 60
  libero, 136
  ordinato, 60
  radicato, 62
  randomizzato di ricerca, 293
  realizzato con liste dei figli, 68
  realizzato con puntatori, 69
  realizzato con vettore dei padri, 67
  sintattico, 60
Algoritmo, XI
  branch-&-bound, 319
  concorrente, 421
  dei 3 indiani, 230
  deterministico, 261
  di approssimazione, 308
  di Batcher, 382
  di Bellman-Ford-Moore, 162
  di Chang-Roberts, 435
  di Dijkstra, 159
  di Floyd, 215
  di Gallager-Humblet-Spira, 440
  di Graham, 189
  di Hirshberg-Sinclair, 447
  di Johnson, 159
  di Knuth-Morris-Pratt, 194
  di Kruskal, 201
  di McNaughton, 307
  di Moore, 204
  di Pape-D'Esopo, 165
  di Preparata-Vuillemin, 406
  di Prim, 208
  di Rabin, 292
  di Ricart-Agrawala, 433
  di Strassen, 243
  distribuito, 430
  efficiente, 15
  enumerativo, 261
  euristico, 329
  inefficiente, 15
  LPT, 311
  non deterministico, 258
  off line, 249
  on line, 249
  ottimo, 18
  parallelo, 363
  polilogaritmico, 370
  polinomiale, 14
  probabilistico, 291
  pseudo-polinomiale, 300
  randomizzato, *vedi* probabilistico
  superpolinomiale, 14
Analisi di algoritmi, 238
  per tentativi, 244
Anello
  bidirezionale, 432
  unidirezionale, 432
Antenato, 62
Antitrasformata di Fourier, 181
Appartenenza, 80
APPARTIENE, 68, 81, 118
Approssimazione assoluta, 308
Arco
  all'indietro, 146
  di attraversamento, 146
  in avanti, 146
  non orientato, 136
  orientato, 135

Area, 395
array, 8
Assegnamento, 9
AT^2, 395
Automa a stati finiti, 355
Azione elementare, 4

B-albero, 129
  binario, 122
Backtrack, 186
begin, 5
BIN-PACKING, 318
BISEZIONE, 330, 334
Bit, 4
BLOCCO STRADALE, 287
$\mathbb{BPP}$, 297
Branch-&-bound, 319
BUBBLESORT, 234
Bucket, *vedi* Cestino
Butterfly, 377

C, 461
CAMMINI MINIMI, 156, 214, 426
CAMMINO
  MINIMO, 156
Cammino, 135
  aumentante, 229
CANCELLA, 68, 81, 118, 294
CANCELLAMIN, 110, 118, 294
Caratterizzazione della soluzione, 157
Cardinalità, 80
Catena, 136
CCC, 378
Cella, 29
Certificato polinomiale, 257
Cestino, 331
Chiave, 93
choice, 259
Ciclo (di un grafo), 135
  Euleriano, 234, 368
Ciclo infinito (di un algoritmo), 339
Circuito, 136
CIRCUITO HAMILTONIANO, 277
Classi di complessità, 345
  probabilistiche, 297
Classi di complessità., 289
Classificazione di un problema, 155
co-$\mathbb{NP}$, 282
co-$\mathbb{RP}$, 297
Coda, 144
  con priorità, 110
    realizzata con heap, 111
    realizzata con uno heap, 159
    realizzata con vettore circolare, 51
coda, 48
Codifica concisa, 267
Coefficiente binomiale, 210
COLLATZ, 339
Collisione, 97
COLORAZIONE, 256
COLUMN-SORT, 416
COMMESSO VIAGGIATORE, 256, 321
  con disuguaglianze triangolari, 309
COMPLEMENTA, 387
Complessità, 18, *vedi* Tempo di calcolo
COMPONENTI CONESSE, 145
COMPOSTEZZA, 284
Compressione dei percorsi, 89
Condizionale, 6
Condizioni di Bellman, 157
Confrontatore a k bit, 405
CONGIUNGI, 296
Connessione, 136
const, 9
Controllo guidato da eventi, 431
Coordinatore, *vedi* Leader
COPERTURA ESATTA DI INSIEMI, 276
CORRETTEZZA, 340
Costo delle operazioni, 10
COUNTING-SORT, 182
CRICCA, 256
CTT, 306
Cursore, 42, 69

Dato, 25
ΔTSP, 309
Dequeue, 426
Diagonalizzazione, 347
Diagramma di transizione, 352
Diametro, 375
Differenza, 80
Diffusione, 431
Dimensione, 10
DIMENSIONE CRICCA MASSIMA, 284
Discendente, 62
dispose, 32
Dispositivo di lettura/scrittura, 350
Distanza, 157
  di Hamming, 332
Diversificazione, 332
Divide et impera, 167
Dizionario, 93
  realizzato con tabella hash, 96
  realizzato con vettore ordinato, 93
DOMINO
  ILLIMITATO, 345, 359
    RICORRENTE, 345
  LIMITATO, 255, 276
  LINEARE, 270

$\mathbb{E}$, 289
ϵ-approssimazione, 315
Elemento, 80
ELEZIONE DEL LEADER, 435

else, 6
end, 5
Enumerazione, 261
EQUAZIONE DIOFANTINA, 451
Esplorazione, *vedi* Visita
Espressione Polinomiale
  ESPRESSIONE POLINOMIALE
    NULLA, 292
ETERO, 342
Etichetta, 55
Euristica, 329
Eventi contabili, 18
$\mathbb{EXP}$, 287
$\mathbb{EXP}$-$\mathbb{SPAZIO}$, 289

failure, 259
FALSITÀ, 283
Fattore di carico, 104
FATTORIALE, 339
FFT, 177
Figlio, 62
  destro, 71
  sinistro, 71
Finger search, 296
FLUSSO MASSIMO, 228
Flusso saturante, 230
Foglia, 62
FONDI, 88
for, 9
for all, 364
for each, 143
FORMULA ARITMETICA
  DI PRESBURGER, 289
FORMULA BOOLEANA
  k-QUANTIFICATA, 285
  QUANTIFICATA, 286
FORMULA WS1S, 289
Frammenti, 438
Fratello, 62
FRAZIONA, 123
function, 7
Funzione, 7
  hash, 98
  μ, *vedi* MU
  universale, *vedi* INTERPRETA

Generatore di numeri casuali, 455
Gerarchia
  aritmetica (di Kleene), 345
  di complessità, 281
  polinomiale, 285
goto, 55
Grado, 375
Grafo
  aciclico, 148
  bipartito, 148
  completo, 309
  connesso, 136
  della computazione, 396
  denso, 162
  di Batcher, 404
  di intervalli, 138
  fortemente connesso, 136
  non orientato, 136
  orientato, 135
  pesato, 140
  planare, 165
  realizzato con matrice, 140
  realizzato con vettori, 142
  residuo, 229
  sparso, 141, 162
Greedy, 200

Hash, 96
Heap, 111
Heapsort, 114

if, 6
Incidenza, 136
INCLINA, 123
Inclusione, 80
Indecidibilità, 339
Indirizzo, 29
INSERISCI, 81, 112, 118, 158, 294, 337
INSERTIONSORT, 225
Insieme, 80
  di adiacenza, 139
  dominante, 166
  indipendente, 166
  realizzato con liste non ordinate, 83
  realizzato con liste ordinate, 84
  realizzato con vettore booleano, 82
  vuoto, 80
Integrazione di circuiti, *vedi* VLSI
Interpolazione
  di un polinomio, 181
  per ricerca in un vettore, 95
INTERPRETA, 341
Intersezione, 80
Intervallo (tipo di dato), 8
Intorno, 221
Inversione, 183, 224
INVILUPPO CONVESSO, 186
Invisita, 66
Ipercubo, 377
ISOMORFISMO TRA GRAFI, 282
Iterazione
  illimitata, 5
  limitata, 9

label, 55
Las Vegas, 450
Lavoro, 365
Layout, 396

Leader, 431
Limitazione
inferiore, 14
superiore, 13
Linguaggio, 281
C, 461
Pascal, 8
LIST UPDATE, *vedi* AGGIORNAMENTO
Lista, 35
bidirezionale, 39
circolare, 39
con cursori, 42
con puntatori, 38
con sentinella, 39
di adiacenza, 142
di trabocco, 101
libera, 42
monodirezionale, 39
Livello
di un nodo in un albero, 62, 137
Livello di dettaglio, 4
lock, 422
LOG-SPAZIO-completezza, 370
Lunghezza
di un cammino, 135
di una catena, 136

Macchina di Turing, 351
generalizzata, 354
universale, 352
MASSIMA
SOTTOSEQUENZA COMUNE, 220
SOTTOSTRINGA COMUNE, 218
MASSIMA SOTTOSEQUENZA COMUNE, 417
MATCHING, *vedi* ABBINAMENTO
Matrice, 28
ISL di traffico, 324
di adiacenza nodi-nodi, 141
di incidenza nodi-archi, 140
di switch, 325
Matroide, 206
maxint, 29
MEDIANO, 245
Memoria secondaria, 106, 130, 170
Memorizzazione
di un record, 31
di un vettore, 29
di una matrice, 30
MERGE, 169
Merge-Find Set (Mfset), 85, 202
realizzato con foresta, 86
MERGESORT
bitonico, 376
MERGESORT, 168
polifase, 170
MESCOLA, 381
Mesh, 376
riconfigurabile, 411
Messaggio, 430
Metodo di scansione, *vedi* Scansione
MIN, 85, 110, 121, 294
MIN-MAX-GREEDY, 330
MINIMO ALBERO DI COPERTURA, 201
MINIMO ALBERO DI COPERTURA, 221
MINIMO ALBERO DI COPERTURA, 438
Modello a griglia, 396
Moltiplicatore VLSI
ricorsivo, 406
sistolico, 406
Moltiplicazione
di matrici, 215, 243, 385, 406
di polinomi, 180
Monte Carlo, 450
MU, 343
Mutua esclusione, 421, 432

Nastro, 350
semi-infinito, 355
$\mathbb{NC}$, 370
new, 32
$\mathbb{NEXP}$, 287
nil, 32
Nodo, 60, 135
isolato, 139
Non determinismo, 255
$\mathbb{NP}$, 268
NP-completezza, 272
forte, 301
$\mathbb{NP}$-$\mathbb{SPAZIO}$, 286
Numerabilità, 347
Numero
casuale, 455
complesso, 177
di Fibonacci, 218
pseudo-casuale, 291

Operazioni primitive, 7
Oracolo, 18
Ordinamento, 15, 72, 114, 168, 171, 189, 224, 225, 234, 268, 366, 372, 382, 399, 404, 416, 424
Ordine
di grandezza, 13
di visita di un albero, 66
topologico, 150
Ottimo
globale, 221
locale, 221
Overflow, 100

$\mathbb{P}$, 268
$\mathbb{P}$-$\mathbb{SPAZIO}$, 286
P-SPAZIO-completo, 286

Padre, 62
Pagina, 106, 130
Paradosso di Russel, 340
Parallelismo
asincrono, 421, 431
sincrono, 363, 375, 397, 412
Parametro
attuale, 5
formale, 5
Parola (di memoria), 29
Parola chiave, 9
PARTIZIONE, 276
Partizione di Eulero, 234
Pascal, 8
Passaggio di un parametro
per valore, 9
per variabile, 9
PERNO, 171
Pila, 48
e ricorsione, 53
realizzata con vettore, 50
Pipeline, 405, 409
Pointer jumping, *vedi* vedi Salto dei puntatori
Poligono convesso, 186
Polinomio
rappresentato con coefficienti, 180
rappresentato con punti, 180
Porta, 412
Portata, 232
Posizione, 35
Postcondizione, 27
Postvisita, 66
Potenziale, 238
$\mathbb{PP}$, 297
PRAM, 363
Precondizione, 27
Predicato, 283
Previsita, 66
PRIMALITÀ, 284, 289
Primitiva di sincronizzazione, 422
Probabilismo, 297
Problema
co-NP-completo, 283
decisionale, 155
dell'arresto (o terminazione), 340
di ottimizzazione, 155
di ricerca, 155
fortemente NP-completo, 301
indecidibile, 339
LOG-SPAZIO-completo, 370
NP-completo, 20, 272
NP-intermedio, 282
semidecidibile, 344
Procedura, 4
procedure, 5
Processo, 421
Processore, 363
Prodotto di una sequenza di matrici, 218
Profondità, 75
Progetto di algoritmi, 155
Programmazione dinamica, 210
PROGRAMMAZIONE LINEARE, 140, 371
INTERA (0/1), 256
PROGRAMMI IN RITARDO, 204
Proibizione fissa, 333
Proibizione randomizzata, 334
Proibizione reattiva, 335
Prova sperimentale (di algoritmo), 459
Puntatore, 31
Punto di ritorno, 57

Quantificatore
esistenziale, 283
universale, 283
QUICKSORT, 171
Quintupla, 351

$\mathbb{R}$, 297
Radice
complessa dell'unità, 177
di un albero, 60
RANDOM, 291, 457
Rappresentazione
binaria di alberi ordinati, 74
di una funzione con un array, 341
in memoria, 29
Rate-Monotonic, 304
REAL-TIME SCHEDULING, 304
Realizzazione, 27
Record, 28
concatenati, 33
di attivazione, 57
record, 28
Regola di Horner, 100
Relazioni di ricorrenza, 12
con partizione bilanciata, 241
lineari a coefficienti costanti, 239
Rete
a grado limitato, 375
di calcolatori, 430
di interconnessione, 376
Ricerca
binaria, 93, 119
con insuccesso, 104
con successo, 104
locale, 221
per interpolazione, 95
tabù, 331
reattiva, 335
Riconoscimento (di linguaggio), 281
Ricorsione, 6, 53
Riducibilità polinomiale, 269
Riduzione con macchine di Turing, 356

Riporto, 414
Ristrutturazione, 106, 247
Rotazione, 123, 294
$\mathbb{RP}$, 297

Salto (istruzione di), 55
Salto dei puntatori, 368
Satelliti geostazionari intercomunicanti, 324
SCAMBIA, 381
Scansione (di una tabella hash), 101
Scheduling, 151, 204, 234, 276, 303, 307, 325
SCHEDULING ISL, 325
SCHEDULING MULTIPROCESSORE, 276
SCHEDULING MULTIPROCESSORE, 307
SCHEDULING MULTIPROCESSORE, 311
Segnale, 395, 411
SELEZIONE, 245
Semidecidibilità, 344
Sentinella, 39, 125, 196
Separa, 294
Sequenza
    bitonica, 382
    unimodale, 382
set, 81
Sezione critica, 421
SHELLSORT, 224
Shuffle, 376
SODDISFATTIBILITÀ, 256
Solido spazio/tempo, 397
Soluzione
    ammissibile, 155
    ottima, 155
SOMMA DI SOTTOINSIEME, 276, 300, 316
SOMMATORIA, 363, 379
SOMME PREFISSE, 364
SORTING, *vedi* ORDINAMENTO
Sottobus, 411
Sottostruttura ottima, 201
Specifica
    astratta, 26
    semantica, 26
    sintattica, 26
Speedup, 423
Stallo, 422
Stella, 432
String Matching
    APPROSSIMATO, 212
STRING MATCHING, 193
Struttura di dati, 26
success, 259
Switch, 325

Tabella
    dei simboli, 96
    hash, 96
TAPPABUCO, 328
Tecniche
    di progetto, 155
Tempo di calcolo, *vedi anche* Complessità
    ammortizzato, 13, 90, 107
    nel caso medio, 10, 104
    nel caso pessimo, 10
Teorema
    del flusso massimo / taglio minimo, 229
    di Bellman, 157
    di Cantor, 348
    di Cook-Levin, 271, 356
    di Eulero, 234
    di Fermat, 289
TERMINA, 342
TERMINAZIONE, 340
Tesi
    della Computazione Parallela, 370
    di Church-Turing, 354
Tesi della Computazione Sequenziale, 354
Testimone di compostezza, 292
then, 6
Timestamp, 430
Tipo di dato, 6
    astratto, 25
    primitivo, 6, 25
Token, 431
Topologia di una rete di calcolatori, 432
Torneo
    TORNEO, 366
TORRI DI HANOI, 53
Trasformata di Fourier, 176, 390
    inversa, 181
3-PARTIZIONE, 302
Treap, 294
Triangolo di Tartaglia, 211
TROVA, 86
TSP, *vedi* COMMESSO VIAGGIATORE
type, 8

Unione, 80
Unione di insiemi disgiunti, 85
Unità di elaborazione, 395
Universalità, 339
unlock, 422

Valutazione di un polinomio, 100, 177
var, 8
Variabile, 7
Vettore, 26
    dei padri, 67
    di adiacenza, 142
Visita
    BFS, 143
    DFS, 143
    anticipata, 66
    differita, 66

    per livelli, 143
    simmetrica, 66
VLSI, 395

when, 431
while, 5

ZAINO
    INTERO (0-1), 208, 265
    REALE, 207
$\mathbb{ZPP}$, 298