Sommario

1 Automi: metodo e follia 1

1.1 Perché studiare la teoria degli automi . . . 2
1.1.1 Introduzione agli automi a stati finiti . . . 2
1.1.2 Rappresentazioni strutturali . . . 4
1.1.3 Automi e complessità . . . 5
1.2 Introduzione alle dimostrazioni formali . . . 6
1.2.1 Dimostrazioni deduttive . . . 7
1.2.2 Riduzione a definizioni . . . 10
1.2.3 Altre forme di teoremi . . . 11
1.2.4 Teoremi che non assomigliano a enunciati se-allora . . . 14
1.3 Altre forme di dimostrazione . . . 14
1.3.1 Dimostrazioni di equivalenze tra insiemi . . . 15
1.3.2 Il contronominale . . . 16
1.3.3 Dimostrazioni per assurdo . . . 18
1.3.4 Controesempi . . . 19
1.4 Dimostrazioni induttive . . . 20
1.4.1 Induzioni sugli interi . . . 21
1.4.2 Forme più generali di induzioni sugli interi . . . 24
1.4.3 Induzioni strutturali . . . 25
1.4.4 Induzione mutua . . . 28
1.5 I concetti centrali della teoria degli automi . . . 30
1.5.1 Alfabeti . . . 31
1.5.2 Stringhe . . . 31
1.5.3 Linguaggi . . . 33
1.5.4 Problemi . . . 34
1.6 Riepilogo . . . 36
1.7 Bibliografia . . . 38

2 Automi a stati finiti **39**
2.1 Una descrizione informale degli automi a stati finiti 40
2.1.1 Le regole fondamentali 40
2.1.2 Il protocollo 41
2.1.3 Automi che possono ignorare azioni 43
2.1.4 L'intero sistema come automa 44
2.1.5 Validazione del protocollo mediante l'automa prodotto 47
2.2 Automi a stati finiti deterministici 48
2.2.1 Definizione di automa a stati finiti deterministico 48
2.2.2 Elaborazione di stringhe in un DFA 49
2.2.3 Notazioni più semplici per i DFA 50
2.2.4 Estensione della funzione di transizione alle stringhe 52
2.2.5 Il linguaggio di un DFA 55
2.2.6 Esercizi 56
2.3 Automi a stati finiti non deterministici 58
2.3.1 Descrizione informale degli automi a stati finiti non deterministici 59
2.3.2 Definizione di automa a stati finiti non deterministico 60
2.3.3 La funzione di transizione estesa 62
2.3.4 Il linguaggio di un NFA 63
2.3.5 Equivalenza di automi a stati finiti deterministici e non deterministici 64
2.3.6 Un caso sfavorevole di costruzione per sottoinsiemi 69
2.3.7 Esercizi 70
2.4 Un'applicazione: ricerche testuali 73
2.4.1 Ricerca di stringhe in un testo 73
2.4.2 Automi a stati finiti non deterministici per ricerche testuali 74
2.4.3 Un DFA per riconoscere un insieme di parole chiave 75
2.4.4 Esercizi 77
2.5 Automi a stati finiti con epsilon-transizioni 77
2.5.1 Uso delle ϵ-transizioni 78
2.5.2 La notazione formale per gli ϵ-NFA 79
2.5.3 Epsilon-chiusure 80
2.5.4 Transizioni estese e linguaggi per gli ϵ-NFA 81
2.5.5 Eliminazione di ϵ-transizioni 83
2.5.6 Esercizi 86
2.6 Riepilogo 86
2.7 Bibliografia 87

3 Espressioni e linguaggi regolari **89**
3.1 Espressioni regolari 89
3.1.1 Gli operatori delle espressioni regolari 90
3.1.2 Costruzione di espressioni regolari 92
3.1.3 Precedenza degli operatori delle espressioni regolari 94
3.1.4 Esercizi 96
3.2 Automi a stati finiti ed espressioni regolari 97
3.2.1 Dai DFA alle espressioni regolari 97
3.2.2 Conversione di DFA in espressioni regolari per eliminazione di stati 103
3.2.3 Conversione di espressioni regolari in automi 109
3.2.4 Esercizi 114
3.3 Applicazioni delle espressioni regolari 116
3.3.1 Le espressioni regolari in UNIX 116
3.3.2 Analisi lessicale 118
3.3.3 Ricerca di pattern in un testo 120
3.3.4 Esercizi 122
3.4 Proprietà algebriche per le espressioni regolari 122
3.4.1 Associatività e commutatività 123
3.4.2 Identità e annichilatori 123
3.4.3 Proprietà distributive 124
3.4.4 La proprietà di idempotenza 125
3.4.5 Proprietà relative alla chiusura 125
3.4.6 Alla ricerca di proprietà per le espressioni regolari 126
3.4.7 Verifica di proprietà algebriche sulle espressioni regolari 128
3.4.8 Esercizi 129
3.5 Riepilogo 131
3.6 Bibliografia 131

4 Proprietà dei linguaggi regolari **133**
4.1 Dimostrare che un linguaggio non è regolare 133
4.1.1 Il pumping lemma per i linguaggi regolari 134
4.1.2 Applicazioni del pumping lemma 135
4.1.3 Esercizi 137
4.2 Proprietà di chiusura dei linguaggi regolari 139
4.2.1 Chiusura dei linguaggi regolari rispetto a operazioni booleane . . 139
4.2.2 Inversione 146
4.2.3 Omomorfismi 147
4.2.4 Omomorfismi inversi 149
4.2.5 Esercizi 155

4.3 Problemi di decisione per i linguaggi regolari 158
4.3.1 Conversioni 159
4.3.2 Verificare se un linguaggio regolare è vuoto 161
4.3.3 Appartenenza a un linguaggio regolare 162
4.3.4 Esercizi 163
4.4 Equivalenza e minimizzazione di automi 164
4.4.1 Verifica dell'equivalenza di stati 164
4.4.2 Equivalenza di linguaggi regolari 167
4.4.3 Minimizzazione di DFA 169
4.4.4 Perché il DFA minimo non può essere migliorato 172
4.4.5 Esercizi 175
4.5 Riepilogo 175
4.6 Bibliografia 176

5 Grammatiche e linguaggi liberi dal contesto 179
5.1 Grammatiche libere dal contesto 180
5.1.1 Un esempio informale 180
5.1.2 Definizione delle grammatiche libere dal contesto 181
5.1.3 Derivazioni per mezzo di una grammatica 183
5.1.4 Derivazioni a sinistra e a destra 186
5.1.5 Il linguaggio di una grammatica 188
5.1.6 Forme sentenziali 189
5.1.7 Esercizi 190
5.2 Alberi sintattici 192
5.2.1 Costruzione di alberi sintattici 193
5.2.2 Il prodotto di un albero sintattico 195
5.2.3 Inferenza, derivazioni e alberi sintattici 196
5.2.4 Dalle inferenze agli alberi 197
5.2.5 Dagli alberi alle derivazioni 198
5.2.6 Dalle derivazioni alle inferenze ricorsive 202
5.2.7 Esercizi 203
5.3 Applicazioni delle grammatiche libere dal contesto 204
5.3.1 Parser 204
5.3.2 YACC: un generatore di parser 206
5.3.3 Linguaggi di markup 208
5.3.4 XML e DTD 211
5.3.5 Esercizi 217
5.4 Ambiguità nelle grammatiche e nei linguaggi 218
5.4.1 Grammatiche ambigue 218
5.4.2 Eliminare le ambiguità da una grammatica 220

5.4.3 Derivazioni a sinistra come modo per esprimere l'ambiguità . . . 223
5.4.4 Ambiguità inerente . . . 224
5.4.5 Esercizi . . . 227
5.5 Riepilogo . . . 229
5.6 Bibliografia . . . 230

6 Automi a pila **233**
6.1 Definizione di automa a pila . . . 233
6.1.1 Introduzione informale . . . 234
6.1.2 Definizione formale di automa a pila . . . 236
6.1.3 Una notazione grafica per i PDA . . . 238
6.1.4 Descrizioni istantanee di un PDA . . . 239
6.1.5 Esercizi . . . 242
6.2 I linguaggi di un PDA . . . 243
6.2.1 Accettazione per stato finale . . . 244
6.2.2 Accettazione per stack vuoto . . . 245
6.2.3 Da stack vuoto a stato finale . . . 246
6.2.4 Da stato finale a stack vuoto . . . 249
6.2.5 Esercizi . . . 251
6.3 Equivalenza di PDA e CFG . . . 253
6.3.1 Dalle grammatiche agli automi a pila . . . 253
6.3.2 Dai PDA alle grammatiche . . . 257
6.3.3 Esercizi . . . 261
6.4 Automi a pila deterministici . . . 263
6.4.1 Definizione di PDA deterministico . . . 263
6.4.2 Linguaggi regolari e PDA deterministici . . . 264
6.4.3 DPDA e linguaggi liberi dal contesto . . . 265
6.4.4 DPDA e grammatiche ambigue . . . 266
6.4.5 Esercizi . . . 267
6.5 Riepilogo . . . 268
6.6 Bibliografia . . . 269

7 Proprietà dei linguaggi liberi dal contesto **271**
7.1 Forme normali per grammatiche libere dal contesto . . . 271
7.1.1 Eliminazione di simboli inutili . . . 272
7.1.2 Calcolo dei simboli generatori e raggiungibili . . . 274
7.1.3 Eliminazione di ϵ-produzioni . . . 275
7.1.4 Eliminazione delle produzioni unitarie . . . 279
7.1.5 Forma normale di Chomsky . . . 283
7.1.6 Esercizi . . . 286

7.2 Il pumping lemma per i linguaggi liberi dal contesto 291
7.2.1 Dimensione degli alberi sintattici 291
7.2.2 Enunciato del pumping lemma 292
7.2.3 Applicazioni del pumping lemma per i CFL 294
7.2.4 Esercizi 297
7.3 Proprietà di chiusura dei linguaggi liberi dal contesto 299
7.3.1 Sostituzioni 299
7.3.2 Applicazioni del teorema di sostituzione 301
7.3.3 Inversione 302
7.3.4 Intersezione con un linguaggio regolare 302
7.3.5 Omomorfismo inverso 307
7.3.6 Esercizi 309
7.4 Proprietà di decisione dei CFL 311
7.4.1 Complessità delle conversioni fra CFG e PDA 312
7.4.2 Tempo di esecuzione della conversione in forma normale di Chomsky 313
7.4.3 Verificare se un CFL è vuoto 314
7.4.4 Appartenenza a un CFL 316
7.4.5 Anteprima di problemi indecidibili per i CFL 320
7.4.6 Esercizi 321
7.5 Riepilogo 322
7.6 Bibliografia 323

8 Macchine di Turing: introduzione **325**
8.1 Problemi che i calcolatori non possono risolvere 325
8.1.1 Programmi che stampano "Ciao, mondo" 326
8.1.2 Un ipotetico verificatore di ciao-mondo 329
8.1.3 Ridurre un problema a un altro 332
8.1.4 Esercizi 335
8.2 La macchina di Turing 335
8.2.1 La ricerca della soluzione a tutte le domande matematiche 336
8.2.2 Notazione per la macchina di Turing 337
8.2.3 Descrizioni istantanee delle macchine di Turing 339
8.2.4 Diagrammi di transizione per le macchine di Turing 342
8.2.5 Il linguaggio di una macchina di Turing 346
8.2.6 Le macchine di Turing e l'arresto 346
8.2.7 Esercizi 347
8.3 Tecniche di programmazione per le macchine di Turing 349
8.3.1 Memoria nello stato 349
8.3.2 Tracce multiple 351

8.3.3 Subroutine 353
8.3.4 Esercizi 354
8.4 Estensioni alla macchina di Turing semplice 356
8.4.1 Macchine di Turing multinastro 356
8.4.2 Equivalenza di macchine di Turing mononastro e multinastro . . . 357
8.4.3 Tempo di esecuzione e costruzione da n a un nastro 359
8.4.4 Macchine di Turing non deterministiche 360
8.4.5 Esercizi 362
8.5 Macchine di Turing ridotte 365
8.5.1 Macchine di Turing con nastri semi-infiniti 365
8.5.2 Macchine multistack 368
8.5.3 Macchine a contatori 371
8.5.4 La potenza delle macchine a contatori 372
8.5.5 Esercizi 374
8.6 Le macchine di Turing e i computer 375
8.6.1 Simulazione di una macchina di Turing da parte di un computer . 375
8.6.2 Simulazione di un computer da parte di una macchina di Turing . 378
8.6.3 Confronto dei tempi di esecuzione dei computer e delle macchine di Turing 382
8.7 Riepilogo 385
8.8 Bibliografia 387

9 Indecidibilità **389**
9.1 Un linguaggio non ricorsivamente enumerabile 390
9.1.1 Enumerazione delle stringhe binarie 391
9.1.2 Codici per le macchine di Turing 391
9.1.3 Il linguaggio di diagonalizzazione 393
9.1.4 Dimostrazione che L_d non è ricorsivamente enumerabile 393
9.1.5 Esercizi 394
9.2 Un problema indecidibile ma ricorsivamente enumerabile 395
9.2.1 Linguaggi ricorsivi 396
9.2.2 Complementi di linguaggi ricorsivi e RE 396
9.2.3 Il linguaggio universale 400
9.2.4 Indecidibilità del linguaggio universale 402
9.2.5 Esercizi 403
9.3 Problemi indecidibili relativi alle macchine di Turing 405
9.3.1 Riduzioni 406
9.3.2 Macchine di Turing che accettano il linguaggio vuoto 407
9.3.3 Il teorema di Rice e le proprietà dei linguaggi RE 411
9.3.4 Problemi sulle specifiche di macchine di Turing 413

9.3.5 Esercizi 414
9.4 Il problema di corrispondenza di Post 415
9.4.1 Definizione del problema di corrispondenza di Post 416
9.4.2 Il PCP modificato 419
9.4.3 Dimostrazione di indecidibilità di PCP: finale 421
9.4.4 Esercizi 427
9.5 Altri problemi indecidibili 428
9.5.1 Problemi relativi a programmi 428
9.5.2 Indecidibilità dell'ambiguità delle CFG 428
9.5.3 Il complemento di un linguaggio associato a una lista 431
9.5.4 Esercizi 433
9.6 Riepilogo 434
9.7 Bibliografia 435

10 Problemi intrattabili **437**
10.1 Le classi $\mathcal{P}$ ed $\mathcal{NP}$ 438
10.1.1 Problemi risolvibili in tempo polinomiale 438
10.1.2 Un esempio: l'algoritmo di Kruskal 439
10.1.3 Tempo polinomiale non deterministico 443
10.1.4 Un esemplare di $\mathcal{NP}$: il problema del commesso viaggiatore . . . 444
10.1.5 Riduzioni polinomiali 445
10.1.6 Problemi NP-completi 446
10.1.7 Esercizi 448
10.2 Un problema NP-completo 451
10.2.1 Il problema della soddisfacibilità 451
10.2.2 Rappresentazione di istanze di SAT 452
10.2.3 NP-completezza del problema SAT 453
10.2.4 Esercizi 460
10.3 Un problema di soddisfacibilità vincolato 461
10.3.1 Forme normali di espressioni booleane 461
10.3.2 Conversione in CNF di espressioni booleane 462
10.3.3 NP-completezza di CSAT 465
10.3.4 NP-completezza di 3SAT 470
10.3.5 Esercizi 472
10.4 Altri problemi NP-completi 472
10.4.1 Descrivere problemi NP-completi 473
10.4.2 Il problema degli insiemi indipendenti 474
10.4.3 Il problema della copertura per nodi 477
10.4.4 Il problema del circuito hamiltoniano orientato 479
10.4.5 Circuiti hamiltoniani non orientati e TSP 484

10.4.6 Riepilogo dei problemi NP-completi 487
10.4.7 Esercizi 488
10.5 Riepilogo 492
10.6 Bibliografia 493

11 Altre classi di problemi **495**
11.1 Complementi dei linguaggi in $\mathcal{NP}$ 496
11.1.1 La classe di linguaggi co-$\mathcal{NP}$ 497
11.1.2 Problemi NP-completi e co-$\mathcal{NP}$ 498
11.1.3 Esercizi 499
11.2 Problemi risolvibili in spazio polinomiale 499
11.2.1 Macchine di Turing a spazio polinomiale 500
11.2.2 Le relazioni di $\mathcal{PS}$ ed $\mathcal{NPS}$ con altre classi 501
11.2.3 Spazio polinomiale deterministico e non deterministico 502
11.3 Un problema completo per $\mathcal{PS}$ 505
11.3.1 PS-completezza 505
11.3.2 Formule booleane con quantificatori 506
11.3.3 Valutazione di formule booleane con quantificatori 507
11.3.4 PS-completezza del problema QBF 509
11.3.5 Esercizi 514
11.4 Classi di linguaggi basate sulla randomizzazione 515
11.4.1 Quicksort: un esempio di algoritmo randomizzato 515
11.4.2 Un modello di macchina di Turing con randomizzazione 516
11.4.3 Il linguaggio di una macchina di Turing randomizzata 518
11.4.4 La classe $\mathcal{RP}$ 520
11.4.5 Riconoscimento di linguaggi in $\mathcal{RP}$ 522
11.4.6 La classe $\mathcal{ZPP}$ 524
11.4.7 Relazioni tra $\mathcal{RP}$ e $\mathcal{ZPP}$ 524
11.4.8 Relazioni con le classi $\mathcal{P}$ ed $\mathcal{NP}$ 525
11.5 Complessità e numeri primi 527
11.5.1 L'importanza della verifica di primalità 527
11.5.2 Introduzione all'aritmetica modulare 529
11.5.3 La complessità del calcolo in aritmetica modulare 531
11.5.4 Verifica di primalità in tempo polinomiale randomizzato 532
11.5.5 Verifiche di primalità non deterministiche 533
11.5.6 Esercizi 536
11.6 Riepilogo 537
11.7 Bibliografia 538

Indice analitico **541**

Prefazione

Nella prefazione all'edizione del 1979 di questo libro, Hopcroft e Ullman si mostravano sorpresi di fronte al fiorire degli studi sugli automi rispetto alla situazione del 1969, anno in cui veniva pubblicato il loro primo volume. In effetti il libro del 1979 trattava numerosi argomenti nuovi ed era lungo circa il doppio. Paragonata a quella del 1979, si scopre che la presente edizione, come le automobili degli anni '70, è "più grande fuori e più piccola dentro". Può sembrare un'involuzione, ma questa nuova veste ci soddisfa per diverse ragioni.

Innanzitutto nel 1979 la teoria degli automi e dei linguaggi era ancora un'area vitale di ricerca, e quel libro aveva anche lo scopo di incoraggiare gli studenti tagliati per la matematica a dare un contributo attivo. Rispetto alle applicazioni, la ricerca pura nella teoria degli automi è attualmente limitata; non c'è più motivo, quindi, di conservare lo stile conciso e più "matematico" del libro del 1979.

In secondo luogo il ruolo della teoria degli automi e dei linguaggi negli ultimi vent'anni è cambiato. Nel 1979 lo studio degli automi era riservato perlopiù a studenti degli ultimi anni di un corso di laurea che avevano intrapreso un indirizzo specifico; essi formavano perciò il pubblico di riferimento, soprattutto per gli ultimi capitoli. Oggi questo tipo di studi è parte integrante del curriculum di base di un corso di laurea. Di conseguenza la scelta dei contenuti deve presupporre che lo studente abbia un bagaglio di conoscenze inferiore, e fornire più nozioni fondamentali e maggiori dettagli nelle dimostrazioni.

Un terzo cambiamento riguarda il contesto: negli ultimi vent'anni l'informatica ha raggiunto un livello quasi inimmaginabile. Nel 1979 era spesso difficile trovare argomenti in grado di superare la successiva ondata di progresso tecnologico e sufficienti a riempire un piano di studi; oggi lo spazio limitato del piano di studi di un corso di laurea è conteso da un numero elevato di discipline.

Un quarto motivo è che la scelta di studiare informatica appare sempre più legata a obiettivi concreti e fra gli studenti si è diffuso un rigido pragmatismo. Siamo ancora convinti che certi aspetti della teoria degli automi siano strumenti essenziali per diverse nuove discipline e che gli esercizi di natura teorica e creativa che fanno parte di un tipico corso sugli automi restino utili, per quanto gli studenti preferiscano apprendere solo le tecni-

che immediatamente monetizzabili. D'altra parte, se vogliamo che la materia conservi il suo ruolo tra le discipline proposte agli studenti di informatica, riteniamo indispensabile sottolinearne le applicazioni, accanto agli aspetti matematici. Abbiamo perciò sostituito alcuni fra gli argomenti più avanzati dell'edizione precedente con esempi di come i concetti possono essere applicati nella pratica. Le applicazioni della teoria degli automi e dei linguaggi formali ai compilatori sono ormai talmente consolidate da essere di norma trattate nei corsi sui compilatori, ma ci sono impieghi più recenti, tra cui gli algoritmi di *model-checking* per verificare protocolli, e i linguaggi di descrizione di documenti, strutturati in modo simile alle grammatiche libere dal contesto.

Un'ultima spiegazione dell'ampliamento e al contempo riduzione del libro risiede nell'impiego di due sistemi di composizione tipografica, TEX e LATEX, sviluppati da Don Knuth e Les Lamport. Uno stile "aperto" di composizione, che produce libri di mole maggiore, ma di più agevole lettura, è favorito in particolare da LATEX. Gli sforzi dei loro ideatori meritano il nostro apprezzamento.

Prerequisiti

Per sfruttare al meglio il libro lo studente dovrebbe aver seguito un corso di matematica discreta e aver acquisito nozioni sui grafi, gli alberi, la logica formale e le tecniche di dimostrazione. Assumiamo inoltre che abbia seguito un corso di programmazione e che conosca le strutture dati più comuni, la ricorsione, e il ruolo dei componenti più importanti di un sistema di elaborazione, tra cui i compilatori. Di solito queste nozioni di base si acquisiscono nei primi due anni di un corso universitario di informatica.

Esercizi

Il libro contiene numerosi esercizi, distribuiti in quasi tutti i paragrafi. Gli esercizi, o parti di esercizi, più difficili sono indicati da un punto esclamativo o, per i più ardui, da un doppio punto esclamativo.

Altri esercizi sono segnalati da un asterisco: le loro soluzioni sono disponibili nella pagina Web del libro e possono servire per valutare il proprio livello di preparazione. In alcuni casi un esercizio B chiede di modificare o adattare la soluzione di un altro esercizio A. Se alcune parti di A ammettono soluzione, ci si può aspettare che lo stesso valga anche per le corrispondenti parti di B.

Supporto in rete

La *home page* del libro è

`http://www-db.stanford.edu/~ullman/ialc.html`

Vi si trovano le soluzioni degli esercizi segnalati da un asterisco, gli *errata corrige* e altro materiale utile. A mano a mano che le lezioni procedono intendiamo pubblicare le dispense del nostro corso, inclusi esercizi e testi d'esame.

Ringraziamenti

La stesura di parte del primo capitolo è stata influenzata da una dispensa di Craig Silverstein su come scrivere le dimostrazioni. Commenti e segnalazioni di errori sulle bozze del libro sono giunti da Zoe Abrams, George Candea, Haowen Chen, Byong-Gun Chun, Jeffrey Shallit, Bret Taylor, Jason Townsend ed Erik Uzureau. Riconosciamo volentieri il loro contributo. Gli errori che restano sono, ovviamente, da imputare a noi.

J. E. H.
R. M.
J. D. U.
Ithaca NY e Stanford CA
Settembre 2000

Capitolo 1

Automi: metodo e follia

La teoria degli automi è lo studio di dispositivi astratti di calcolo, o "macchine". Negli anni '30, prima dell'avvento dei computer, A. Turing studiò una macchina astratta che aveva tutte le capacità degli elaboratori odierni, almeno per quanto riguarda ciò che possono calcolare. Il fine di Turing era descrivere precisamente il confine tra quello che un dispositivo di calcolo può fare e quello che non può fare; le sue conclusioni non riguardano solo le sue *macchine di Turing* astratte, ma anche le macchine reali di adesso.

Negli anni '40 e '50 diversi ricercatori studiarono alcuni tipi più semplici di macchine, che oggi sono dette *automi a stati finiti*. Questi automi, pensati originariamente per fornire un modello del funzionamento cerebrale, risultarono utili per molti altri scopi, che saranno introdotti nel Paragrafo 1.1. Nello stesso periodo, nei tardi anni '50, il linguista N. Chomsky iniziò a studiare le grammatiche formali. Per quanto non siano delle macchine in senso proprio, queste grammatiche sono strettamente collegate agli automi astratti e oggi stanno alla base di alcuni importanti componenti software, tra cui parti dei compilatori.

Nel 1969 S. Cook approfondì gli studi di Turing su ciò che si può calcolare. Cook riuscì a distinguere i problemi risolvibili in modo efficiente da un elaboratore da quelli che possono essere risolti in linea di principio, ma che di fatto richiedono così tanto tempo da rendere inutilizzabile un computer se non per istanze del problema di dimensione limitata. Questa seconda classe di problemi viene definita *intrattabile*, o *NP-hard*. È molto probabile che nemmeno il progresso esponenziale nella velocità di calcolo dell'hardware (legge di Moore) avrà un effetto determinante sulle nostre capacità di risolvere casi significativi di problemi intrattabili.

Tutti questi sviluppi teorici hanno un rapporto diretto con quanto gli informatici fanno attualmente. Alcuni concetti, come gli automi a stati finiti e certi tipi di grammatiche formali, vengono usati nella progettazione e realizzazione di importanti tipi di software. Altri concetti, come la macchina di Turing, aiutano a comprendere che cosa ci si può aspetta-

re dal software. In particolare la teoria dei problemi intrattabili permette di capire se è probabile che si riesca ad affrontare un problema direttamente e a scrivere un programma per risolverlo (in quanto non incluso nella classe intrattabile), oppure se sia necessario escogitare un modo per aggirarlo: trovare un'approssimazione, usare un metodo euristico o di qualche altra natura per limitare la quantità di tempo che il programma impiega per risolvere il problema.

In questo capitolo introduttivo si comincia da una panoramica ad alto livello della teoria degli automi e delle sue applicazioni. Gran parte del capitolo è dedicata all'indagine delle tecniche di dimostrazione e ai modi per ideare dimostrazioni. Ci occupiamo di dimostrazioni deduttive, di riformulazioni di enunciati, di dimostrazioni per assurdo e per induzione, e di altri importanti concetti. L'ultimo paragrafo introduce i concetti che permeano la teoria degli automi: alfabeti, stringhe e linguaggi.

1.1 Perché studiare la teoria degli automi

Ci sono svariate ragioni per cui lo studio degli automi e della complessità è parte essenziale dell'informatica. Questo paragrafo presenta al lettore la motivazione principale e delinea gli argomenti più rilevanti trattati in questo libro.

1.1.1 Introduzione agli automi a stati finiti

Gli automi a stati finiti sono un utile modello di molte categorie importanti di hardware e software. A partire dal Capitolo 2 esemplificheremo l'impiego dei concetti. Per ora ci limitiamo a elencare alcuni dei casi più significativi.

1. Software per progettare circuiti digitali e verificarne il comportamento.

2. L'analizzatore lessicale di un compilatore, ossia il componente che scompone l'input (i dati in ingresso) in unità logiche, come gli identificatori, le parole chiave e la punteggiatura.

3. Software per esaminare vaste collezioni di testi, ad esempio pagine Web, per trovare occorrenze di parole o di frasi.

4. Software per verificare sistemi di qualsiasi tipo, che abbiano un numero finito di stati discreti, come i protocolli di comunicazione oppure i protocolli per lo scambio sicuro di informazioni.

Forniremo una definizione precisa di automi di vario tipo tra breve. Intanto cominciamo questa introduzione informale descrivendo che cos'è e che cosa fa un automa a stati finiti. Ci sono molti sistemi o componenti, come quelli elencati sopra, di cui si può dire che in

ogni istante si trovano in uno "stato" preso da un insieme finito. Uno stato ha lo scopo di ricordare una parte pertinente della storia del sistema. Dal momento che c'è solo un numero finito di stati generalmente non si può ricordare l'intera storia, perciò il sistema dev'essere progettato attentamente affinché ricordi ciò che è importante e dimentichi ciò che non lo è. Il vantaggio di avere solo un numero finito di stati è che il sistema può essere implementato con un insieme fissato di risorse. Per esempio è possibile implementarlo come un circuito, oppure come un semplice programma che può prendere decisioni esaminando solo un numero limitato di dati o usando la posizione nel codice stesso per prendere la decisione.

Esempio 1.1 Un interruttore è forse il più semplice automa a stati finiti non banale. Un dispositivo di questo tipo ricorda se è nello stato *on* (acceso) oppure nello stato *off* (spento) e permette all'utente di premere (*push*) un pulsante il cui effetto è diverso a seconda del suo stato: se l'interruttore è nello stato "spento", allora premere il pulsante lo fa passare nello stato "acceso"; viceversa, se l'interruttore è nello stato "acceso", allora premere il medesimo pulsante lo porta nello stato "spento".

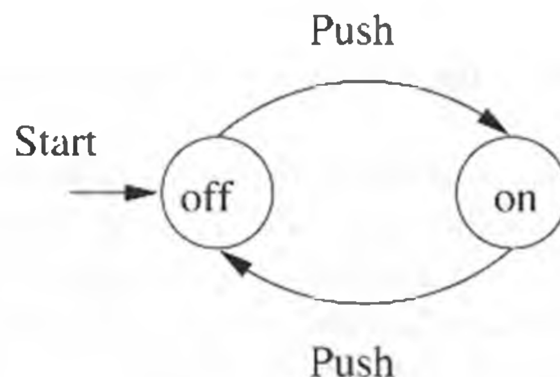


Figura 1.1 Un automa a stati finiti che rappresenta un interruttore.

Il modello di automa a stati finiti per l'interruttore è rappresentato nella Figura 1.1. Come per tutti gli automi a stati finiti, gli stati vengono indicati per mezzo di cerchi; nell'esempio considerato gli stati sono stati chiamati *off* e *on*. Gli archi tra gli stati vengono etichettati dagli input, che rappresentano le influenze esterne sul sistema. Qui ambedue gli archi sono etichettati dall'input *Push*, che rappresenta la pressione del pulsante. Il significato dei due archi è che, indipendentemente dallo stato presente, a fronte di un input *Push* il sistema passa nell'altro stato.

Uno degli stati è detto "stato iniziale", cioè lo stato in cui il sistema si trova inizialmente. Nell'esempio lo stato iniziale è *off*, e per convenzione lo denoteremo con la parola *Start* (avvio) e una freccia che punta a quello stato. Spesso è necessario indicare uno o più stati come gli stati "finali" o "accettanti". Se uno di questi stati viene raggiunto dopo una sequenza di input, tale sequenza di input si considera valida. Per esempio potremmo

considerare lo stato *on*, nella Figura 1.1, come lo stato accettante, in quanto il dispositivo controllato dall'interruttore in quello stato è attivo. Per convenzione gli stati accettanti vengono rappresentati da un doppio cerchio, sebbene nella Figura 1.1 questa convenzione non sia stata rispettata. □

Esempio 1.2 Talvolta ciò che uno stato ricorda è più complesso di una semplice scelta acceso/spento. La Figura 1.2 mostra un altro automa a stati finiti che potrebbe far parte di un analizzatore lessicale. Il compito di questo automa è riconoscere la parola-chiave `then`. L'automa ha quindi bisogno di cinque stati, ognuno dei quali rappresenta una diversa posizione raggiunta nella parola `then`. Le posizioni corrispondono ai prefissi della parola, a partire dalla stringa vuota (vale a dire: finora nessuna parte della parola è stata vista) sino alla parola completa.

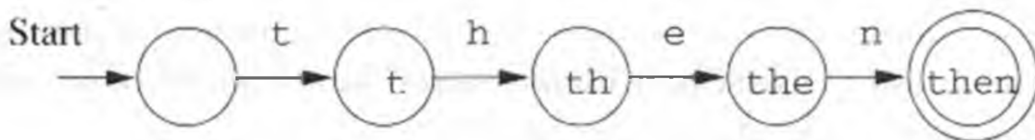


Figura 1.2 Un automa a stati finiti che modella il riconoscimento di `then`.

Nella Figura 1.2 i cinque stati prendono il nome del prefisso di `then` visto fino a quel punto. Gli input corrispondono alle lettere. Possiamo immaginare che l'analizzatore lessicale esamini un carattere per volta tra quelli del programma che sta compilando; ogni nuovo carattere della sequenza da esaminare è l'input dell'automa. Lo stato iniziale corrisponde alla stringa vuota e ogni stato ha una transizione attraverso la successiva lettera di `then` nello stato che corrisponde al prefisso più ampio che viene immediatamente dopo. Si passa allo stato chiamato `then` quando l'input ha formato la parola `then`. Poiché il compito di questo automa è riconoscere quando `then` è stato visto, possiamo considerare quello come l'unico stato accettante. □

1.1.2 Rappresentazioni strutturali

Ci sono due notazioni importanti, diverse dagli automi, ma che hanno un ruolo di primo piano nello studio di questi e delle loro applicazioni.

1. Le *grammatiche* sono modelli utili quando si progetta software che elabora dati con una struttura ricorsiva. L'esempio più noto è il *parser*, un componente del compilatore che tratta gli elementi ricorsivi di un tipico linguaggio di programmazione, come le espressioni aritmetiche, condizionali, ecc. Per esempio una regola grammaticale come $E \Rightarrow E + E$ dichiara che un'espressione può essere formata prendendo due espressioni qualunque e connettendole con un segno più; questa

regola mostra come si formano di solito le espressioni dei linguaggi di programmazione. Le grammatiche libere dal contesto (*context-free grammars*), come vengono comunemente chiamate, verranno introdotte nel Capitolo 5.

2. Anche le *espressioni regolari* denotano la struttura di un dato, specialmente di stringhe testuali. Come si vedrà nel Capitolo 3, le configurazioni di stringhe che vengono descritte dalle espressioni regolari sono esattamente le stesse che possono essere descritte da automi a stati finiti. Lo stile di queste espressioni si differenzia significativamente da quello delle grammatiche. Limitiamoci a un semplice esempio. L'espressione regolare in stile UNIX `'[A-Z][a-z]*[ ][A-Z][A-Z]'` rappresenta parole con lettere iniziali maiuscole seguite da uno spazio e due lettere maiuscole. Questa espressione rappresenta configurazioni testuali che potrebbero indicare una città seguita dall'indicazione dello stato, come `Ithaca NY`. Non riesce a rappresentare nomi di città formati da più parole, come `Palo Alto CA`, che potrebbe essere reso dall'espressione più complessa

   ```
   '[A-Z][a-z]*([ ][A-Z][a-z]*)*[ ][A-Z][A-Z]'
   ```

   Quando interpretiamo questa espressione, l'unica cosa che dobbiamo sapere è che `[A-Z]` rappresenta una gamma di caratteri dalla "A" maiuscola alla "Z" maiuscola (ossia qualunque lettera maiuscola) e `[ ]` indica il singolo carattere di spaziatura. Inoltre `*` rappresenta "qualunque numero di" rispetto all'espressione precedente. Le parentesi si usano per raggruppare componenti dell'espressione: non rappresentano caratteri del testo descritto.

1.1.3 Automi e complessità

Gli automi sono essenziali per lo studio dei limiti della computazione. Come accennato nell'introduzione al capitolo, ci sono due punti importanti.

1. Che cosa può fare un computer in assoluto? Tale studio è detto "decidibilità", e i problemi risolvibili da un computer sono detti "decidibili". Il tema è affrontato nel Capitolo 9.

2. Che cosa può fare un computer in maniera efficiente? Tale studio è detto "intrattabilità", e i problemi risolvibili da un computer in un tempo limitato da una funzione lentamente crescente della dimensione dell'input sono detti "trattabili". Spesso si considerano le funzioni polinomiali come "lentamente crescenti", mentre si ritiene che le funzioni che crescono più rapidamente delle polinomiali crescano troppo velocemente. L'argomento viene affrontato nel Capitolo 10.

1.2 Introduzione alle dimostrazioni formali

Se avete studiato geometria piana alle superiori prima degli anni '90, molto probabilmente avete dovuto fare accurate dimostrazioni deduttive, in cui si dimostra la verità di un enunciato per mezzo di una dettagliata sequenza di passi e di ragionamenti. Mentre la geometria ha un suo lato pratico (per esempio, se si deve comprare la giusta quantità di moquette per una stanza, bisogna conoscere la formula per calcolare l'area del rettangolo), lo studio dei metodi di dimostrazione formale era un motivo almeno altrettanto importante per trattare questo ramo della matematica nelle scuole superiori.

Negli anni '90 è diventato prassi negli USA insegnare la dimostrazione come se si trattasse di una questione di sensazioni personali circa un enunciato. Sebbene sia giusto percepire la verità di un enunciato da usare, nelle scuole superiori non si padroneggiano più importanti tecniche di dimostrazione. Eppure la dimostrazione è un elemento che ogni informatico deve comprendere. Alcuni studiosi di infomatica arrivano a dire che una dimostrazione formale della correttezza di un programma e la scrittura del programma stesso dovrebbero andare di pari passo. Dubitiamo che tale procedimento sia produttivo. D'altro canto c'è chi non lascia alcuno spazio alle dimostrazioni nella disciplina della programmazione, affidandosi spesso allo slogan "se non sei sicuro che il tuo programma sia corretto, eseguilo e vedi".

Noi ci collochiamo tra i due estremi. Sicuramente provare un programma è essenziale. Tuttavia il controllo è limitato, in quanto non è possibile provare un programma su ogni input. Inoltre, e questo è ancora più importante, quando il programma è complesso, come nel caso di una ricorsione o di un'iterazione complicata, se non si capisce che cosa succede in prossimità di un ciclo o di una chiamata ricorsiva, difficilmente si potrà scrivere codice corretto. Quando una prova rivela che il programma non è corretto, sarà comunque necessario sistemarlo.

Per scrivere un'iterazione o una ricorsione corretta bisogna fondarsi su un'ipotesi induttiva. È utile valutare, formalmente o informalmente, che l'ipotesi sia coerente con l'iterazione o la ricorsione. Questo processo di comprensione dei meccanismi di un programma corretto coincide in sostanza con il processo di dimostrazione di teoremi per induzione. Perciò, oltre a fornire modelli utili per determinati tipi di software, in un corso di teoria degli automi si è affermato l'uso di trattare metodi di dimostrazione formale. Forse più di altri temi centrali dell'informatica la teoria degli automi si presta a dimostrazioni naturali e interessanti, sia di tipo *deduttivo* (una sequenza di passi giustificati) sia di tipo *induttivo* (dimostrazioni ricorsive di un enunciato parametrizzato, che usano l'enunciato stesso con valori del parametro "più piccoli").

1.2.1 Dimostrazioni deduttive

Come accennato sopra, una dimostrazione deduttiva consiste in una sequenza di enunciati la cui verità conduce da un enunciato iniziale, l'*ipotesi* o *enunciato dato*, a un enunciato *conclusivo*. Ogni passo della dimostrazione deve seguire, in virtù di qualche principio logico accettato, dai fatti dati, oppure da uno dei precedenti enunciati nella dimostrazione deduttiva, oppure da una combinazione di questi.

Le ipotesi possono essere vere o false, di solito a seconda dei valori dei loro parametri. Spesso le ipotesi consistono di svariati enunciati indipendenti connessi da un AND logico. In questi casi ogni enunciato viene considerato come un'ipotesi.

Il teorema che viene dimostrato quando si va da un'ipotesi H a una conclusione C è l'enunciato "se H allora C". Diciamo che C è *dedotto* da H. Un teorema esemplificativo della forma "se H allora C" illustrerà questi punti.

Teorema 1.3 Se $x \geq 4$, allora $2^x \geq x^2$. □

Non è difficile convincersi informalmente che il Teorema 1.3 è vero, sebbene una dimostrazione formale richieda l'induzione (si veda più avanti l'Esempio 1.17). In primo luogo si noti che l'ipotesi H è "$x \geq 4$". Quest'ipotesi ha un parametro, x, e dunque non è né vera né falsa. La sua verità dipende piuttosto dal valore del parametro x: per esempio H è vera per $x = 6$ e falsa per $x = 2$.

Analogamente la conclusione C è "$2^x \geq x^2$". Anche quest'enunciato usa il parametro x ed è vero per certi valori di x e non per altri. Per esempio C è falsa per $x = 3$, dato che $2^3 = 8$, che non è tanto grande quanto $3^2 = 9$. D'altra parte C è vera per $x = 4$, in quanto $2^4 = 4^2 = 16$. Per $x = 5$ l'enunciato è altrettanto vero, poiché $2^5 = 32$ è maggiore o uguale a $5^2 = 25$.

È probabilmente palese l'argomentazione intuitiva per cui la conclusione $2^x \geq x^2$ sarà vera ogni volta che $x \geq 4$. Abbiamo già visto che è vero per $x = 4$. Quando x diventa maggiore di 4, il membro sinistro 2^x raddoppia ogni volta che x cresce di 1. Tuttavia il membro destro, x^2, cresce secondo il rapporto $\left(\frac{x+1}{x}\right)^2$. Se $x \geq 4$, allora $(x + 1)/x$ non può essere maggiore di 1,25, e perciò $\left(\frac{x+1}{x}\right)^2$ non può essere maggiore di 1,5625. Dato che $1,5625 < 2$, quando x è maggiore di 4 il membro sinistro 2^x cresce più del membro destro x^2. Dunque, purché si parta da un valore come $x = 4$, per cui la disuguaglianza $2^x \geq x^2$ è già soddisfatta, possiamo incrementare x quanto vogliamo e la disuguaglianza resterà soddisfatta.

Con ciò abbiamo completato una dimostrazione informale ma rigorosa del Teorema 1.3. Ci ritorneremo precisando la dimostrazione nell'Esempio 1.17, dopo aver introdotto le dimostrazioni induttive.

Il Teorema 1.3, come tutti i teoremi interessanti, implica un numero infinito di fatti correlati, in questo caso l'enunciato "se $x \geq 4$ allora $2^x \geq x^2$" per tutti gli interi x. Di fatto non abbiamo bisogno di assumere che x sia un intero, ma nella dimostrazione

si è parlato ripetutamente di incrementare x di 1 partendo da $x = 4$. Ci siamo dunque effettivamente occupati solo della situazione in cui x è un intero.

Enunciati con quantificatori

In molti teoremi compaiono enunciati che usano i *quantificatori* "per ogni" ed "esiste", o varianti simili, come "per qualunque" invece di "per ogni". L'ordine in cui compaiono influisce sul significato dell'enunciato. Spesso è utile vedere gli enunciati con più di un quantificatore come una "partita" tra due giocatori, "per-ogni" ed "esiste", che a turno specificano i valori dei parametri menzionati nel teorema. Poiché "per-ogni" deve considerare tutte le scelte possibili, generalmente le sue scelte sono lasciate come variabili; "esiste", invece, deve selezionare un valore, che può dipendere dai valori scelti in precedenza. L'ordine dei quantificatori nell'enunciato determina chi parte per primo. Se l'ultimo giocatore che compie una scelta può trovare sempre un valore accettabile, l'enunciato è vero.

Consideriamo per esempio una definizione alternativa di insieme infinito: l'insieme S è *infinito* se e solo se, per ogni intero n, esiste almeno un sottoinsieme T di S con n elementi. Qui "per-ogni" precede "esiste", dunque dobbiamo considerare un intero arbitrario n. Ora "esiste" seleziona un sottoinsieme T servendosi della sua conoscenza di n. Per esempio, se S fosse l'insieme degli interi, "esiste" potrebbe scegliere il sottoinsieme $T = \{1, 2, \ldots, n\}$ e così farcela quale che sia n. Questa è una dimostrazione che l'insieme degli interi è infinito.

L'enunciato seguente sembra simile alla definizione di infinito, ma è *scorretto* perché rovescia l'ordine dei quantificatori: "esiste un sottoinsieme T dell'insieme S tale che, per qualunque n, l'insieme T ha esattamente n elementi". Ora, dato un insieme S come gli interi, il giocatore "esiste" può selezionare qualsiasi insieme T. Poniamo che venga scelto $\{1, 2, 5\}$. Per questa scelta il giocatore "per-ogni" deve mostrare che T ha n elementi per *ogni* possibile n. Tuttavia "per-ogni" non può farlo. Per esempio l'affermazione è falsa per $n = 4$ e, di fatto, per ogni $n \neq 3$.

Il Teorema 1.3 può essere utilizzato come ausilio per dedurre altri teoremi. Nel prossimo esempio si passa a considerare una dimostrazione deduttiva completa di un semplice teorema che impiega il Teorema 1.3.

Teorema 1.4 Se x è la somma dei quadrati di quattro interi positivi, allora $2^x \geq x^2$.

DIMOSTRAZIONE L'idea intuitiva della dimostrazione è che se l'ipotesi è vera per x, cioè x è la somma dei quadrati di quattro numeri interi positivi, allora x dev'essere almeno

uguale a 4. Perciò l'ipotesi del Teorema 1.3 è valida e, poiché siamo certi della validità di tale teorema, possiamo affermare che la sua conclusione è vera anche per x. Si può esprimere il ragionamento come una sequenza di passi. Ogni passo è l'ipotesi del teorema da dimostrare, una parte di quell'ipotesi, oppure un enunciato che consegue da uno o più enunciati precedenti.

Con "consegue" si intende che, se l'ipotesi di un determinato teorema è un enunciato precedente, la conclusione di quel teorema è vera e può essere scritta come un enunciato della nuova dimostrazione. Questa regola logica è detta spesso *modus ponens*. Vale a dire, se sappiamo che H è vera e sappiamo che "se H allora C" è vera, possiamo concludere che C è vera. Inoltre ammettiamo che altri passi logici vengano usati nel formare un enunciato che segue da uno o più enunciati precedenti. Per esempio se A e B sono due enunciati precedenti, allora possiamo dedurre e scrivere l'enunciato "A e B."

La Figura 1.3 mostra la sequenza degli enunciati di cui abbiamo bisogno per dimostrare il Teorema 1.4. Sebbene generalmente non dimostreremo teoremi in forma così schematizzata, è utile pensare le dimostrazioni come esplicite liste di enunciati, ciascuno con una precisa giustificazione. Nel passo (1) abbiamo ripetuto uno degli enunciati dati del teorema: che x è la somma dei quadrati di quattro interi. Nelle dimostrazioni è spesso comodo dare un nome a quantità che compaiono senza nome nell'ipotesi. In questo caso facciamo così, e diamo ai quattro numeri interi i nomi a, b, c e d.

	Enunciato	Giustificazione
1.	$x = a^2 + b^2 + c^2 + d^2$	ipotesi
2.	$a \geq 1; b \geq 1; c \geq 1; d \geq 1$	ipotesi
3.	$a^2 \geq 1; b^2 \geq 1; c^2 \geq 1; d^2 \geq 1$	(2) e proprietà dell'aritmetica
4.	$x \geq 4$	(1), (3), e proprietà dell'aritmetica
5.	$2^x \geq x^2$	(4) e Teorema 1.3

Figura 1.3 Una dimostrazione formale del Teorema 1.4.

Nel passo (2) stabiliamo l'altra parte dell'ipotesi del teorema: che i valori da elevare al quadrato siano uguali almeno a 1. Tecnicamente questo enunciato rappresenta quattro enunciati distinti, uno per ognuno dei quattro interi in questione. Poi nel passo (3) osserviamo che se un numero è almeno 1, anche il suo quadrato è almeno 1. Adduciamo come giustificazione il fatto che l'enunciato (2) è valido e facciamo riferimento a "proprietà dell'aritmetica". Cioè presumiamo che il lettore conosca, o possa dimostrare, enunciati semplici sulle diseguaglianze, come l'enunciato "se $y \geq 1$, allora $y^2 \geq 1$".

Il passo (4) impiega gli enunciati (1) e (3). Il primo enunciato dice che x è la somma dei quattro quadrati in questione e l'enunciato (3) dice che ognuno dei quadrati è almeno 1. Ricorrendo di nuovo a ben noti principi dell'aritmetica, concludiamo che x è almeno $1 + 1 + 1 + 1$, cioè 4.

Al passo finale (5) usiamo l'enunciato (4), che è l'ipotesi del Teorema 1.3. Il teorema stesso è la giustificazione per poter scrivere la sua conclusione, dato che la sua ipotesi è un enunciato precedente. Poiché l'enunciato (5), che è la conclusione del Teorema 1.3, è anche la conclusione del Teorema 1.4, quest'ultimo risulta dimostrato. Ossia siamo partiti dall'ipotesi di un teorema e siamo riusciti a dedurne la conclusione. □

1.2.2 Riduzione a definizioni

Nei due teoremi precedenti le ipotesi usano termini che supponiamo familiari, per esempio interi, addizione e moltiplicazione. In altri teoremi, compresi molti che sono tratti dalla teoria degli automi, i termini usati nell'enunciato possono avere implicazioni meno evidenti. Un modo di procedere utile in molte dimostrazioni si può esprimere come segue.

- Se non siete sicuri di come iniziare una dimostrazione, convertite tutti i termini dell'ipotesi nelle rispettive definizioni.

Vediamo un esempio di teorema semplice da dimostrare, una volta che i suoi enunciati siano stati espressi in termini elementari. Esso fa uso delle due definizioni seguenti.

1. Un insieme S è *finito* se esiste un intero n tale che S abbia esattamente n elementi. Scriviamo $\|S\| = n$, dove $\|S\|$ denota il numero di elementi in un insieme S. Se l'insieme S non è finito, diciamo che è *infinito*. Intuitivamente un insieme infinito è un insieme il cui numero di elementi è maggiore di qualsiasi numero intero.

2. Se S e T sono ambedue sottoinsiemi di un determinato insieme U, allora T è il *complemento* di S (rispetto a U) se $S \cup T = U$ e $S \cap T = \emptyset$. Cioè ogni elemento di U è esattamente in uno dei due insiemi S e T. In altre parole T consta esattamente degli elementi di U che non sono in S.

Teorema 1.5 Sia S un sottoinsieme finito di un insieme infinito U. Sia T il complemento di S in U. Allora T è infinito.

DIMOSTRAZIONE Intuitivamente questo teorema dice che se si ha una quantità infinita di qualcosa (U), e se ne porta via una quantità finita (S), allora resta comunque una quantità infinita. Cominciamo riformulando i fatti del teorema, come si vede nella Figura 1.4.

Siamo ancora bloccati, quindi dobbiamo usare una tecnica di dimostrazione comune, detta "dimostrazione per assurdo". In questo metodo di dimostrazione, che verrà discusso ulteriormente nel Paragrafo 1.3.3, supponiamo che la conclusione sia falsa. Ricorriamo a questa supposizione, insieme a parti dell'ipotesi, per provare il contrario di uno degli enunciati dati dell'ipotesi. Abbiamo così mostrato che è impossibile che tutte le parti dell'ipotesi siano vere e che la conclusione sia falsa allo stesso tempo. L'unica possibilità che rimane è che la conclusione sia vera quando l'ipotesi è vera. In altre parole il teorema è vero.

Enunciato originale	Nuovo enunciato
S è finito	Esiste un intero n tale che $\|S\| = n$
U è infinito	Per nessun intero p vale $\|U\| = p$
T è il complemento di S	$S \cup T = U$ e $S \cap T = \emptyset$

Figura 1.4 Riformulazione dei dati del Teorema 1.5.

Nel caso del Teorema 1.5, la negazione della conclusione è "T è finito". Supponiamo che T sia finito, insieme all'enunciato dell'ipotesi che pone S come finito; cioè $\|S\| = n$ per un intero n. Analogamente possiamo riformulare l'assunto che T è finito come $\|T\| = m$ per un intero m.

Uno degli enunciati dati ci dice ora che $S \cup T = U$ e $S \cap T = \emptyset$, cioè che gli elementi di U sono esattamente gli elementi di S e T. Dunque ci devono essere $n + m$ elementi in U. Dato che $n + m$ è un intero e abbiamo dimostrato che $\|U\| = n + m$, ne consegue che U è finito. Più precisamente abbiamo mostrato che il numero di elementi in U è un intero, che è la definizione di "finito". Ma l'enunciato che U è finito contraddice l'ipotesi che U sia infinito. Abbiamo dunque usato la negazione della nostra conclusione per dimostrare la negazione di uno degli enunciati dati dell'ipotesi, e in virtù del principio della dimostrazione per assurdo possiamo concludere che il teorema è vero. $\square$

Non è necessario che le dimostrazioni siano così prolisse. Dopo aver esaminato le idee su cui poggia la dimostrazione, dimostriamo un'altra volta il Teorema 1.5 limitandoci a poche righe.

DIMOSTRAZIONE Sappiamo che $S \cup T = U$ e che S e T sono disgiunti, per cui $\|S\| + \|T\| = \|U\|$. Poiché S è finito, abbiamo che $\|S\| = n$ per un intero n, e poiché U è infinito, non esiste nessun intero p tale che $\|U\| = p$. Supponiamo che T sia finito, cioè $\|T\| = m$ per un intero m. Allora $\|U\| = \|S\| + \|T\| = n + m$, il che contraddice l'ipotesi che non esiste alcun intero p uguale a $\|U\|$. $\square$

1.2.3 Altre forme di teoremi

Il teorema nella forma "se-allora" è il più comune in molte aree della matematica. Tuttavia ci sono anche altri tipi di enunciati dimostrati come teoremi. In questo paragrafo esamineremo le più comuni forme di enunciato e che cosa bisogna fare di solito per dimostrarle.

Modi di dire se-allora

In primo luogo ci sono svariati tipi di enunciati di teoremi che sembrano diversi da una forma semplice come "se H allora C", ma che effettivamente dicono la stessa cosa: se l'ipotesi H è vera per un valore dato del parametro (o dei parametri), allora la conclusione C è vera per lo stesso valore. Ecco altri modi in cui può presentarsi "se H allora C".

1. H implica C.
2. H solo se C.
3. C se H.
4. Quando H, segue C.

Inoltre ci sono molte varianti della forma (4), come "se vale H, allora ne consegue C", oppure "ogni volta che vale H, vale anche C".

Esempio 1.6 L'enunciato del Teorema 1.3 si presenta così in queste quattro forme:

1. $x \geq 4$ implica $2^x \geq x^2$
2. $x \geq 4$ solo se $2^x \geq x^2$
3. $2^x \geq x^2$ se $x \geq 4$
4. quando $x \geq 4$, segue $2^x \geq x^2$.

□

Nella logica formale si trova spesso l'operatore $\rightarrow$ anziché se-allora. In alcuni testi di matematica l'enunciato "se H allora C" compare quindi come $H \rightarrow C$. Qui non se ne farà uso.

Enunciati nella forma se-e-solo-se

A volte si trovano enunciati nella forma "A se e solo se B". Altre forme di quest'enunciato sono "A sse (iff) B"[1], "A è equivalente a B", oppure "A esattamente quando B". Quest'enunciato consta effettivamente di due enunciati se-allora: "se A allora B" e "se B allora A". Si dimostra "A se e solo se B" dimostrando questi due enunciati.

1. La *parte se*: "se B allora A".

[1] sse, abbreviazione di "se e solo se", è una sigla che viene usata in alcuni testi di matematica per brevità (in inglese *iff*, abbreviazione di "if and only if").

Quanto formali devono essere le dimostrazioni

Risolvere la questione non è facile. Il fattore essenziale in materia di dimostrazioni è che il loro scopo è quello di convincere qualcuno, un docente che valuta il compito o l'autore stesso, della correttezza di una strategia che si sta usando nel codice. Se è convincente non occorre altro; se invece non riesce a convincere il destinatario della dimostrazione, è segno che si sono tralasciati troppi elementi.

Parte dell'incertezza riguardo alle dimostrazioni è dovuta al diverso grado di conoscenza del destinatario. Perciò nel Teorema 1.4 abbiamo supposto che il lettore avesse una conoscenza completa dell'aritmetica e che avrebbe creduto a un enunciato come "se $y \geq 1$ allora $y^2 \geq 1$". In caso contrario si sarebbe dovuto procedere con qualche passo nella dimostrazione deduttiva.

Tuttavia determinati elementi sono obbligatori nelle dimostrazioni: la loro omissione rende inadeguata una dimostrazione. Per esempio qualunque dimostrazione deduttiva che impieghi enunciati non giustificati dall'ipotesi o da enunciati precedenti non può essere adeguata. Quando svolgiamo la dimostrazione di un enunciato "se e solo se", di certo dobbiamo avere una dimostrazione per la parte "se" e un'altra per la parte "solo-se". A titolo di esempio aggiuntivo, le dimostrazioni induttive, discusse nel Paragrafo 1.4, richiedono dimostrazioni per la base e per l'induzione.

2. La *parte solo-se*: "se A allora B", spesso formulata nella forma equivalente "A solo se B".

Le dimostrazioni possono essere presentate in qualsiasi ordine. In molti teoremi una parte è decisamente più facile dell'altra, ed è comune, per togliersi il pensiero, presentare prima la direzione più semplice.

Per denotare un enunciato se-e-solo-se, nella logica formale si può trovare l'operatore $\leftrightarrow$ oppure $\equiv$. Cioè $A \equiv B$ e $A \leftrightarrow B$ hanno lo stesso significato di "A se e solo se B".

Quando si dimostra un enunciato se-e-solo-se, è importante ricordare che si deve dimostrare sia la parte "se" sia la parte "solo-se". A volte sarà utile scomporre un se-e-solo-se in una successione di equivalenze. Vale a dire, per dimostrare "A se e solo se B" si può in primo luogo dimostrare "A se e solo se C", e poi dimostrare "C se e solo se B". Il metodo funziona, purché si ricordi che ogni passo se-e-solo-se deve essere dimostrato in entrambe le direzioni. Dimostrare un qualunque passo in una sola direzione inficia l'intera dimostrazione.

Vediamo un esempio di semplice dimostrazione se-e-solo-se in cui si fa uso delle notazioni seguenti.

1. $\lfloor x \rfloor$, la *parte intera* (*floor*) di un numero reale x, è il più grande intero uguale o minore di x.

2. $\lceil x \rceil$, il *tetto* (*ceiling*) di un numero reale x, è il più piccolo intero uguale o maggiore di x.

Teorema 1.7 Sia x un numero reale. Allora $\lfloor x \rfloor = \lceil x \rceil$ se e solo se x è un intero.

DIMOSTRAZIONE (Parte solo-se) In questa parte supponiamo che $\lfloor x \rfloor = \lceil x \rceil$ e proviamo a dimostrare che x è un intero. Usando le definizioni di parte intera e di tetto notiamo che $\lfloor x \rfloor \leq x$ e $\lceil x \rceil \geq x$. Tuttavia si è detto per ipotesi che $\lfloor x \rfloor = \lceil x \rceil$. Perciò possiamo sostituire la parte intera con il tetto nella prima disuguaglianza per concludere che $\lceil x \rceil \leq x$. Poiché sia $\lceil x \rceil \leq x$ sia $\lceil x \rceil \geq x$ sono valide, possiamo concludere, per le proprietà delle disuguaglianze aritmetiche, che $\lceil x \rceil = x$. Dato che $\lceil x \rceil$ è sempre un intero, in questo caso anche x dev'essere un intero.

(Parte se) Supponiamo ora che x sia un intero e cerchiamo di dimostrare $\lfloor x \rfloor = \lceil x \rceil$. Questa parte è facile. Per definizione, quando x è un intero, sia $\lfloor x \rfloor$ sia $\lceil x \rceil$ sono uguali a x, e dunque uguali fra loro. □

1.2.4 Teoremi che non assomigliano a enunciati se-allora

Talvolta si incontra un teorema che sembra non avere un'ipotesi. Un esempio è la ben nota formula trigonometrica:

Teorema 1.8 $\sin^2 \theta + \cos^2 \theta = 1$. □

In realtà quest'enunciato *ha* un'ipotesi, che consiste di tutti gli enunciati di cui si ha bisogno per interpretarlo. In particolare l'ipotesi nascosta è che θ è un angolo, e dunque le funzioni seno e coseno hanno il loro significato geometrico consueto. Partendo dalla definizione di questi termini e dal teorema di Pitagora (in un triangolo rettangolo il quadrato dell'ipotenusa è uguale alla somma dei quadrati degli altri due lati) possiamo dimostrare il teorema. Essenzialmente la forma se-allora del teorema è: "se θ è un angolo, allora $\sin^2 \theta + \cos^2 \theta = 1$".

1.3 Altre forme di dimostrazione

In questo paragrafo trattiamo alcuni argomenti supplementari riguardanti la stesura di dimostrazioni:

1. dimostrazioni sugli insiemi

2. dimostrazioni per assurdo

3. dimostrazioni per controesempio.

1.3.1 Dimostrazioni di equivalenze tra insiemi

Nella teoria degli automi si deve spesso dimostrare un teorema per il quale due insiemi costruiti in modo diverso sono uguali. Frequentemente si tratta di insiemi di stringhe di caratteri, detti "linguaggi", ma in questo paragrafo la loro natura non è importante. Se E ed F sono due espressioni che rappresentano insiemi, l'enunciato $E = F$ significa che i due insiemi rappresentati sono uguali. Più precisamente, ogni elemento dell'insieme rappresentato da E si trova nell'insieme rappresentato da F e ogni elemento dell'insieme rappresentato da F si trova nell'insieme rappresentato da E.

Esempio 1.9 La *proprietà commutativa dell'unione* afferma che possiamo formare l'unione di due insiemi R ed S in entrambi gli ordini; in altre parole $R \cup S = S \cup R$. In questo caso E è l'espressione $R \cup S$ ed F è l'espressione $S \cup R$. La proprietà commutativa dell'unione dice che $E = F$. □

Possiamo scrivere l'uguaglianza di insiemi $E = F$ come un enunciato della forma se-e-solo-se: un elemento x è in E se e solo se x è in F. Da questo possiamo ricavare lo schema di una dimostrazione di qualunque enunciato che asserisca l'uguaglianza dei due insiemi $E = F$ secondo la forma di una dimostrazione se-e-solo-se:

1. dimostrare che se x è in E, allora x è in F
2. dimostrare che se x è in F, allora x è in E.

Per esemplificare questo processo dimostrativo proviamo la *proprietà distributiva dell'unione rispetto all'intersezione.*

Teorema 1.10 $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

DIMOSTRAZIONE Le due espressioni insiemistiche sono $E = R \cup (S \cap T)$ e

$$F = (R \cup S) \cap (R \cup T)$$

Dimostreremo a turno le due parti del teorema. Nella parte "se" poniamo che l'elemento x sia in E e mostriamo che è in F. Questa parte, riassunta nella Figura 1.5, usa le definizioni di unione e intersezione, che si presumono note.

Dobbiamo poi dimostrare la parte solo-se del teorema. Qui poniamo che x sia in F e mostriamo che è in E. Le fasi sono riassunte nella Figura 1.6. Dato che abbiamo dimostrato entrambe le parti dell'enunciato se-e-solo-se, abbiamo anche provato la proprietà distributiva dell'unione rispetto all'intersezione. □

	Enunciato	Giustificazione
1.	x è in $R \cup (S \cap T)$	ipotesi
2.	x è in R o x è in $S \cap T$	(1) e definizione di unione
3.	x è in R o x è sia in S sia in T	(2) e definizione di intersezione
4.	x è in $R \cup S$	(3) e definizione di unione
5.	x è in $R \cup T$	(3) e definizione di unione
6.	x è in $(R \cup S) \cap (R \cup T)$	(4), (5), e definizione di intersezione

Figura 1.5 Passi della parte "se" del Teorema 1.10.

	Enunciato	Giustificazione
1.	x è in $(R \cup S) \cap (R \cup T)$	ipotesi
2.	x è in $R \cup S$	(1) e definizione di intersezione
3.	x è in $R \cup T$	(1) e definizione di intersezione
4.	x è in R o x è sia in S sia in T	(2), (3), e ragionamento sulle unioni
5.	x è in R o x è in $S \cap T$	(4) e definizione di intersezione
6.	x è in $R \cup (S \cap T)$	(5) e definizione di unione

Figura 1.6 Passi della parte "solo-se" del Teorema 1.10.

1.3.2 Il contronominale

Ogni enunciato se-allora ha una forma equivalente che in alcune circostanze è più semplice da dimostrare. Il *contronominale* dell'enunciato "se H allora C" è "se non C allora non H". Un enunciato e il suo contronominale sono entrambi veri oppure entrambi falsi. Dunque possiamo dimostrarne uno per dimostrare anche l'altro.

Per vedere perché "se H allora C" e "se non C allora non H" sono logicamente equivalenti, osserviamo in primo luogo che ci sono quattro casi da considerare:

1. H e C entrambe vere

2. H vera e C falsa

3. C vera e H falsa

4. H e C entrambe false.

Se-e-solo-se per insiemi

Come menzionato, i teoremi che affermano equivalenze fra espressioni sugli insiemi sono enunciati se-e-solo-se. Dunque il Teorema 1.10 si può esprimere così: un elemento x è in $R \cup (S \cap T)$ se e solo se x è in

$$(R \cup S) \cap (R \cup T)$$

Un'altra tipica espressione di equivalenza fra insiemi si serve della locuzione "tutti-e-soli". Per esempio il Teorema 1.10 si può esprimere così: "gli elementi di $R \cup (S \cap T)$ sono tutti e soli gli elementi di $(R \cup S) \cap (R \cup T)$".

C'è solo un modo per rendere falso un enunciato se-allora: l'ipotesi dev'essere vera e la conclusione falsa, come nel caso (2). Per gli altri tre casi, incluso il caso (4) in cui la conclusione è falsa, l'enunciato se-allora in sé è vero.

Consideriamo ora per quali casi il contronominale "se non C allora non H" è falso. Affinché quest'enunciato sia falso, la sua ipotesi (che è "non C") dev'essere vera e la sua conclusione (che è "non H") deve essere falsa. Ma "non C" è vera esattamente quando C è falsa e "non H" è falsa esattamente quando H è vera. Queste due condizioni sono nuovamente il caso (2), il che mostra che in ognuno dei quattro casi l'enunciato originale e il suo contronominale sono ambedue veri oppure ambedue falsi. In altre parole sono logicamente equivalenti.

Esempio 1.11 Torniamo al Teorema 1.3, il cui enunciato è: "se $x \geq 4$, allora $2^x \geq x^2$". Il contronominale di quest'enunciato è: "se non $2^x \geq x^2$, allora non $x \geq 4$". In termini più colloquiali, tenendo conto che "non $a \geq b$" è lo stesso di $a < b$, il contronominale è "se $2^x < x^2$ allora $x < 4$". □

Quando si deve dimostrare un teorema se-e-solo-se, l'uso del contronominale in una delle parti permette numerose opzioni. Supponiamo per esempio di dover dimostrare l'equivalenza di insiemi $E = F$. Invece di dimostrare "se x è in E allora x è in F e se x è in F allora x è in E", possiamo esprimere una direzione in forma contronominale. Una forma di dimostrazione equivalente è:

- se x è in E allora x è in F e se x non è in E allora x non è in F.

Nell'enunciato sopra si possono anche scambiare E ed F.

L'inverso

Non si confondano i termini "contronominale" e "inverso". L'*inverso* di un enunciato se-allora è "l'altra direzione"; ossia l'inverso di "se H allora C" è "se C allora H". A differenza del contronominale, che è logicamente equivalente all'originale, l'inverso *non* è equivalente all'enunciato originale. In effetti le due parti della dimostrazione se-e-solo-se sono sempre un enunciato e il suo inverso.

1.3.3 Dimostrazioni per assurdo

Un altro modo di dimostrare un enunciato nella forma "se H allora C" è dimostrare l'enunciato

- "H e non C implica il falso".

In altre parole si assumono inizialmente sia l'ipotesi H sia la negazione della conclusione C e si completa la dimostrazione provando che qualcosa di notoriamente falso segue logicamente da H e non C. Questa forma è detta *dimostrazione per assurdo*.

Esempio 1.12 Consideriamo il Teorema 1.5 in cui è stato dimostrato un enunciato se-allora con l'ipotesi H = "U è un insieme infinito, S è un sottoinsieme finito di U e T è il complemento di S rispetto a U". La conclusione C era "T è infinito". Abbiamo dimostrato questo teorema per assurdo, assumendo "non C"; vale a dire, abbiamo assunto che T sia finito.

La dimostrazione mirava a derivare il falso da H e non C. In primo luogo abbiamo mostrato, a partire dall'assunto che S e T sono entrambi finiti, che anche U dev'essere finito. Ma poiché si è detto nell'ipotesi H che U è infinito, e un insieme non può essere contemporaneamente finito e infinito, è stato dimostrato l'enunciato logico "falso". In termini logici abbiamo sia una proposizione p (U è finito) sia la sua negazione non p (U è infinito). Ci serviamo dunque del fatto che "p e non p" equivale logicamente a "falso". □

Per capire come mai le dimostrazioni per assurdo sono logicamente corrette si ricordi il Paragrafo 1.3.2, in cui ci sono quattro combinazioni di valori di verità per H e C. Solo il secondo caso, H vera e C falsa, rende falso l'enunciato "se H allora C". Mostrando che H e non C porta al falso stiamo dimostrando che il caso 2 non può aver luogo. Perciò le sole combinazioni possibili di valori di verità per H e C sono le tre combinazioni che rendono vera "se H allora C".

1.3.4 Controesempi

Nella vita quotidiana non si devono dimostrare teoremi. Piuttosto ci si trova di fronte a cose che sembrano vere – una strategia per implementare un programma, per esempio – e si deve decidere se il "teorema" è vero o no. Per risolvere la questione possiamo cercare di dimostrare il teorema o, se non ci riusciamo, cercare di dimostrare che il suo enunciato è falso.

Generalmente i teoremi sono enunciati per un numero infinito di casi, per esempio tutti i valori dei rispettivi parametri. In realtà una rigida convenzione matematica nobilita un enunciato del titolo di teorema solo se ha un numero infinito di casi. Gli enunciati che non hanno parametri, o che si applicano solo a un numero finito di valori dei parametri, sono detti *osservazioni*. Basta dimostrare che un teorema presunto è falso in uno qualunque dei casi per dimostrare che non si tratta di un teorema. La situazione è analoga ai programmi, poiché generalmente si considera che un programma contiene un errore se non riesce a operare correttamente anche per un solo input su cui dovrebbe funzionare.

Spesso è più facile dimostrare che un enunciato non è un teorema anziché dimostrare che lo *è*. Come detto sopra, se S è un qualunque enunciato, allora "S non è un teorema" è a sua volta un enunciato senza parametri, e perciò può essere considerato un'osservazione invece di un teorema. Esponiamo due esempi, il primo riguardante un evidente non-teorema, il secondo un enunciato che solo per poco non è un teorema e che richiede qualche ragionamento per stabilire se lo sia o no.

Teorema presunto 1.13 Tutti i numeri primi sono dispari (in termini più formali si potrebbe dire: se l'intero x è un numero primo, allora x è dispari).

CONFUTAZIONE L'intero 2 è un numero primo, ma 2 è pari. □

Discutiamo ora un "teorema" relativo all'aritmetica modulare. Si deve porre in primo luogo una definizione essenziale: se a e b sono interi positivi, allora a mod b è il resto della divisione di a per b, vale a dire l'unico intero r tra 0 e $b-1$ tale che $a = qb + r$ per un intero q. Per esempio 8 mod 3 = 2 e 9 mod 3 = 0. Il primo teorema candidato, che si vedrà falso, è il seguente.

Teorema presunto 1.14 Non esiste una coppia di numeri interi a e b tali che

$$a \bmod b = b \bmod a$$

□

Quando ci si trova a operare con coppie di oggetti come a e b, spesso è possibile sfruttare qualche simmetria per semplificare le loro relazioni reciproche. Qui ci si può concentrare sul caso in cui $a < b$, dato che se $b < a$ si possono scambiare a e b, e ottenere la stessa equazione del presunto Teorema 1.14. Tuttavia bisogna prestare attenzione a non

dimenticare il terzo caso: $a = b$. Questo caso risulterà decisivo per i nostri tentativi di dimostrazione.

Supponiamo che $a < b$. Allora $a \bmod b = a$, poiché nella definizione di $a \bmod b$ abbiamo $q = 0$ e $r = a$. In altre parole, quando $a < b$ abbiamo $a = 0 \times b + a$. Ma $b \bmod a < a$, poiché qualsiasi numero mod a sta tra 0 e $a - 1$. Dunque, quando $a < b$, $b \bmod a < a \bmod b$, e $a \bmod b = b \bmod a$ è impossibile. Ricorrendo all'argomento della simmetria di cui sopra, sappiamo anche che $a \bmod b \neq b \bmod a$ quando $b < a$.

Consideriamo tuttavia il terzo caso: $a = b$. Poiché $x \bmod x = 0$ per qualunque intero x, *abbiamo* effettivamente $a \bmod b = b \bmod a$ se $a = b$. Perciò abbiamo confutato il presunto Teorema 1.14.

CONFUTAZIONE Sia $a = b = 2$. Allora

$$a \bmod b = b \bmod a = 0$$

□

Nel corso della ricerca del controesempio abbiamo in effetti scoperto le condizioni esatte sotto cui il teorema presunto è valido. Ecco la versione corretta del teorema e la sua dimostrazione.

Teorema 1.15 $a \bmod b = b \bmod a$ se e solo se $a = b$.

DIMOSTRAZIONE (Parte se) Sia $a = b$. Allora, come osservato precedentemente, $x \bmod x = 0$ per qualunque intero x. Perciò $a \bmod b = b \bmod a = 0$ se $a = b$.

(Parte solo-se) Supponiamo ora che $a \bmod b = b \bmod a$. Il miglior modo di procedere è una dimostrazione per assurdo; assumiamo dunque anche la negazione della conclusione, ossia sia $a \neq b$. Allora, poiché $a = b$ è eliminato, restano da considerare solo i casi $a < b$ e $b < a$.

È già stato osservato che quando $a < b$ abbiamo $a \bmod b = a$ e $b \bmod a < a$. Perciò da questi enunciati e dall'ipotesi $a \bmod b = b \bmod a$ si ricava una contraddizione. Per simmetria, se $b < a$ allora $b \bmod a = b$ e $a \bmod b < b$. Si può nuovamente dedurre una contraddizione dell'ipotesi e concludere che anche la parte solo-se è vera. Con ciò sono state dimostrate entrambe le direzioni e si può affermare che il teorema è vero. □

1.4 Dimostrazioni induttive

Esiste una forma speciale di dimostrazione, detta *induttiva*, che è essenziale quando si ha a che fare con oggetti definiti ricorsivamente. Molte delle dimostrazioni induttive più familiari si occupano di interi, ma nella teoria degli automi c'è bisogno anche di dimostrazioni induttive su concetti definiti ricorsivamente come alberi ed espressioni di

varia natura, per esempio le espressioni regolari cui si è accennato nel Paragrafo 1.1.2. In questo paragrafo si introdurrà il concetto di dimostrazione induttiva a partire da semplici induzioni su interi. In seguito si mostrerà come operare induzioni strutturali su qualunque concetto definito ricorsivamente.

1.4.1 Induzioni sugli interi

Supponiamo di dover dimostrare un enunciato $S(n)$ su un intero n. Una soluzione comune consiste nel dimostrare due cose.

1. La *base*, in cui si dimostra $S(i)$ per un particolare intero i. Di solito $i = 0$ oppure $i = 1$, ma ci sono esempi in cui si comincia da un i più alto, magari perché l'enunciato S è falso per alcuni interi piccoli.

2. Il *passo induttivo*, in cui supponiamo che $n \geq i$, dove i è l'intero di base, e si dimostra che "se $S(n)$ allora $S(n+1)$".

Intuitivamente queste due parti dovrebbero convincerci che $S(n)$ è vero per ogni intero n che sia uguale o maggiore dell'intero di base i. Il ragionamento è il seguente. Supponiamo che $S(n)$ sia falso per uno o più di quei numeri interi. Allora dovrebbe esistere un minimo valore di n, poniamo j, per il quale $S(j)$ è falso e tuttavia $j \geq i$. Ma j non può essere i perché abbiamo dimostrato (base) che $S(i)$ è vero; perciò j dev'essere maggiore di i. Ora sappiamo che $j - 1 \geq i$ e $S(j - 1)$ è vero.

Proseguendo, nel passo induttivo abbiamo dimostrato che se $n \geq i$ allora $S(n)$ implica $S(n+1)$. Supponiamo che sia $n = j - 1$. Allora sappiamo dal passo induttivo che $S(j-1)$ implica $S(j)$. Poiché sappiamo che $S(j - 1)$ è vero, possiamo concludere che lo è anche $S(j)$.

Abbiamo ipotizzato la negazione di quanto volevamo dimostrare; abbiamo cioè supposto che $S(j)$ fosse falso per un qualche $j \geq i$. In ognuno dei casi abbiamo ricavato una contraddizione, così abbiamo ottenuto una dimostrazione per assurdo che $S(n)$ è vero per ogni $n \geq i$.

Purtroppo c'è un sottile difetto logico in questo ragionamento. Il nostro assunto, che si possa scegliere il minimo $j \geq i$ per il quale $S(j)$ è falso, già dipende dalla nostra fiducia nel principio di induzione. In altre parole l'unico modo di dimostrare che si può trovare questo j è dimostrarlo per mezzo di un metodo che è essenzialmente una dimostrazione induttiva. Tuttavia la "dimostrazione" discussa sopra è intuitivamente ragionevole e si accorda con la comune comprensione del mondo reale. Generalmente si prende dunque come parte integrante del nostro modo di ragionare il seguente principio.

- *Principio di induzione*: se dimostriamo $S(i)$ e dimostriamo che, per ogni $n \geq i$, $S(n)$ implica $S(n+1)$, allora possiamo concludere $S(n)$ per ogni $n \geq i$.

I due esempi seguenti illustrano l'uso del principio di induzione per dimostrare teoremi sugli interi.

Teorema 1.16 Per ogni $n \geq 0$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

DIMOSTRAZIONE Questa dimostrazione consta di due parti che saranno dimostrate a turno: la base e il passo induttivo.

BASE Per la base scegliamo $n = 0$. Può sembrare sorprendente che il teorema sia valido anche per $n = 0$, dato che il membro sinistro dell'Equazione (1.1) è $\sum_{i=1}^{0} i^2$ quando $n = 0$. Tuttavia c'è un principio generale per cui, se il limite superiore di una somma (in questo caso 0) è minore del limite inferiore (qui 1), la somma non ha termini e quindi è 0. In altre parole $\sum_{i=1}^{0} i^2 = 0$.

Anche il membro destro dell'Equazione (1.1) è 0, dato che $0 \times (0+1) \times (2 \times 0+1)/6 = 0$. Perciò l'Equazione (1.1) è vera quando $n = 0$.

INDUZIONE Supponiamo ora che $n \geq 0$. Dobbiamo dimostrare il passo induttivo, cioè che l'Equazione (1.1) implica la stessa formula con $n + 1$ al posto di n. La seconda formula è

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

Possiamo semplificare le equazioni (1.1) e (1.2) espandendo le somme e i prodotti a destra:

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.4}$$

Dobbiamo dimostrare (1.4) facendo uso di (1.3) perché nel principio di induzione questi sono rispettivamente gli enunciati $S(n + 1)$ e $S(n)$. Il trucco è di scomporre la somma fino a $n + 1$ a sinistra di (1.4) in una somma fino a n più il termine $(n + 1)$-esimo. In questo modo possiamo rimpiazzare la somma fino a n con il membro destro di (1.3) e mostrare che (1.4) è vera. I passi si configurano così:

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n)/6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6)/6 \quad (1.6)$$

La verifica finale, che (1.6) è vera, richiede solo semplici nozioni di algebra dei polinomi applicate al membro sinistro per dimostrare che è identico alla parte destra. □

Esempio 1.17 Nel prossimo esempio dimostriamo il Teorema 1.3 del Paragrafo 1.2.1. Ricordiamo che questo teorema dice che se $x \geq 4$ allora $2^x \geq x^2$. Ne abbiamo dato una dimostrazione informale, basata sull'idea che il rapporto $x^2/2^x$ si riduce quando x cresce oltre 4. Si può precisare l'idea dimostrando l'enunciato $2^x \geq x^2$ per induzione su x, partendo da una base di $x = 4$. Si noti che l'enunciato è effettivamente falso per $x < 4$.

BASE Se $x = 4$, allora 2^x e x^2 sono entrambi 16. Perciò $2^4 \geq 4^2$ è valido.

INDUZIONE Supponiamo che $2^x \geq x^2$ per $x \geq 4$. Con questa ipotesi dobbiamo dimostrare lo stesso enunciato con $x + 1$ al posto di x, vale a dire $2^{[x+1]} \geq [x + 1]^2$. Questi sono gli enunciati $S(x)$ e $S(x + 1)$ nel principio di induzione. Il fatto che si stia usando x invece di n come parametro non è rilevante; x ed n sono solo variabili locali.

Come nel Teorema 1.16, dobbiamo riscrivere $S(x + 1)$ in modo che contenga $S(x)$. In questo caso possiamo scrivere $2^{[x+1]}$ come 2×2^x. Dato che $S(x)$ ci dice che $2^x \geq x^2$, possiamo concludere che $2^{x+1} = 2 \times 2^x \geq 2x^2$.

Abbiamo però bisogno di qualcosa di diverso, in quanto dobbiamo dimostrare che $2^{x+1} \geq (x + 1)^2$. A tale scopo possiamo dimostrare che $2x^2 \geq (x + 1)^2$ e poi ricorrere alla transitività di $\geq$ per mostrare che $2^{x+1} \geq 2x^2 \geq (x + 1)^2$. Nella dimostrazione che

$$2x^2 \geq (x + 1)^2 \quad (1.7)$$

possiamo far uso dell'assunto che $x \geq 4$. Cominciamo semplificando la (1.7):

$$x^2 \geq 2x + 1 \quad (1.8)$$

Dividiamo la (1.8) per x e otteniamo:

$$x \geq 2 + \frac{1}{x} \quad (1.9)$$

Dato che $x \geq 4$, sappiamo che $1/x \leq 1/4$. Perciò il membro sinistro di (1.9) è almeno 4 e il membro destro è al massimo 2.25. In questo modo abbiamo dimostrato che (1.9) è vera. Perciò le Equazioni (1.8) e (1.7) sono altrettanto vere. L'Equazione (1.7) dà a sua volta il risultato $2x^2 \geq (x + 1)^2$ per $x \geq 4$ e dimostra l'enunciato $S(x + 1)$ che, come ricordiamo, era $2^{x+1} \geq (x + 1)^2$. □

I numeri interi come concetti definiti ricorsivamente

Si è detto che le dimostrazioni induttive sono utili quando l'oggetto su cui si ragiona è definito ricorsivamente. Tuttavia i primi esempi trattati sono induzioni su interi, che normalmente non si considerano definiti ricorsivamente. Ciononostante, esiste una definizione naturale e ricorsiva di numero intero non negativo, e questa definizione si accorda con la maniera in cui si svolgono le induzioni sugli interi: dagli oggetti definiti prima a quelli definiti in seguito.

BASE 0 è un intero.

INDUZIONE Se n è un intero, lo è anche $n + 1$.

1.4.2 Forme più generali di induzioni sugli interi

A volte è possibile fare una dimostrazione induttiva solo ricorrendo a uno schema più generale rispetto a quello proposto nel Paragrafo 1.4.1, in cui abbiamo dimostrato l'enunciato S per un valore di base e poi abbiamo dimostrato che "se $S(n)$ allora $S(n + 1)$". Esistono due importanti generalizzazioni di questo schema.

1. Possiamo usare diversi casi di base. In altre parole dimostriamo, per un $j > i$, gli enunciati $S(i), S(i + 1), \ldots, S(j)$.

2. Nella dimostrazione di $S(n + 1)$ possiamo usare la validità di tutti gli enunciati

$$S(i), S(i + 1), \ldots, S(n)$$

piuttosto che usare semplicemente $S(n)$. Inoltre, se abbiamo dimostrato i casi di base fino a $S(j)$, possiamo supporre che $n \geq j$, anziché solo $n \geq i$.

La conclusione che si può ricavare da questa base e dal passo induttivo è che $S(n)$ è vero per tutti i valori $n \geq i$.

Esempio 1.18 Il seguente esempio illustrerà il potenziale di entrambi i principi. L'enunciato $S(n)$ che vogliamo dimostrare è che se $n \geq 8$, allora n può essere scritto come la somma di multipli di 3 e di 5. Si noti, per inciso, che 7 non può essere scritto così.

BASE I casi di base sono $S(8)$, $S(9)$ e $S(10)$. Le dimostrazioni sono rispettivamente $8 = 3 + 5$, $9 = 3 + 3 + 3$ e $10 = 5 + 5$.

INDUZIONE Supponiamo che $n \geq 10$ e che $S(8), S(9), \ldots, S(n)$ siano veri. Da questi dati dobbiamo dimostrare $S(n+1)$. La strategia consiste nel sottrarre 3 da $n+1$, osservare

che il risultato si può scrivere come somma di multipli di 3 e di 5, e aggiungere un ulteriore 3 alla somma per poter scrivere $n + 1$.

In termini più formali si osserva che $n - 2 \geq 8$, così possiamo supporre $S(n - 2)$. In altre parole $n - 2 = 3a + 5b$ per due interi a e b. Allora $n + 1 = 3 + 3a + 5b$, così $n + 1$ può essere scritto come la somma di 3, moltiplicato per $a + 1$, e di 5, moltiplicato per b. Ciò dimostra $S(n + 1)$ e conclude il passo induttivo. □

1.4.3 Induzioni strutturali

Nella teoria degli automi esistono diverse strutture definite ricorsivamente su cui si devono dimostrare enunciati. Importanti esempi sono le nozioni familiari di alberi e di espressioni. Similmente alle induzioni, tutte le definizioni ricorsive hanno un caso di base, in cui si definiscono una o più strutture elementari, e un passo induttivo, in cui si definiscono strutture più complesse nei termini di strutture definite in precedenza.

Esempio 1.19 Una definizione ricorsiva di albero.

BASE Un singolo nodo è un albero, e tale nodo è la *radice* dell'albero.

INDUZIONE Se $T_1, T_2, \ldots, T_k$ sono alberi, allora si può formare un nuovo albero in questo modo:

1. si comincia con un nuovo nodo N, che è la radice dell'albero
2. si aggiunge una copia degli alberi $T_1, T_2, \ldots, T_k$
3. si aggiungono lati dal nodo N alle radici di ogni albero $T_1, T_2, \ldots, T_k$.

La Figura 1.7 mostra la costruzione induttiva di un albero con radice N a partire da k alberi più piccoli. □

Esempio 1.20 Consideriamo un'altra definizione ricorsiva. Questa volta definiamo *espressioni* con gli operatori matematici $+$ e $*$, aventi come operandi sia numeri sia variabili.

BASE Qualunque numero o lettera (ossia una variabile) è un'espressione.

INDUZIONE Se E ed F sono espressioni, allora lo sono anche $E + F$, $E * F$ e (E).

Per esempio sia 2 sia x sono espressioni (caso di base). Il passo induttivo dice che $x + 2$, $(x + 2)$ e $2 * (x + 2)$ sono espressioni. Si noti come ognuna di queste espressioni dipende dal fatto che le precedenti sono a loro volta espressioni. □

Data una definizione ricorsiva, si possono dimostrare teoremi su di essa con la seguente forma di dimostrazione, detta *induzione strutturale*. Sia $S(X)$ un enunciato sulle strutture X definite in modo ricorsivo.

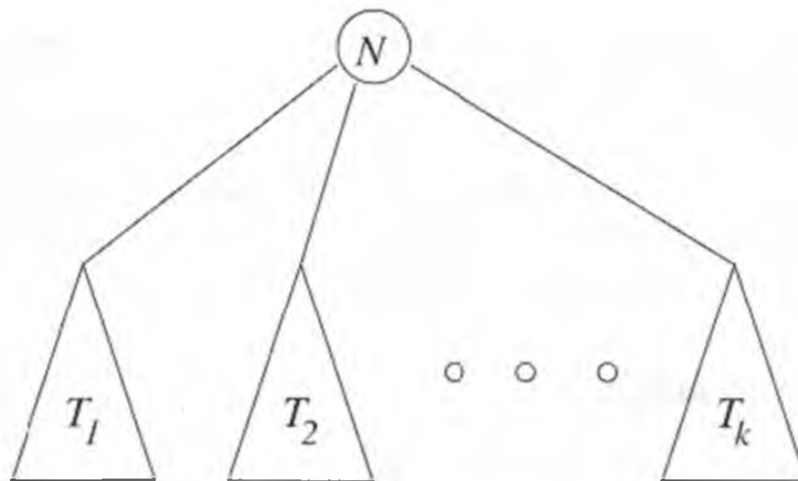


Figura 1.7 Costruzione induttiva di un albero.

1. Come base si dimostra $S(X)$ per la struttura (o le strutture) di base di X.

2. Per il passo induttivo si prende una struttura X che secondo la definizione ricorsiva è formata da $Y_1, Y_2, \ldots, Y_k$. Si assumono veri gli enunciati $S(Y_1), S(Y_2), \ldots, S(Y_k)$ e da essi si dimostra $S(X)$.

Si conclude che $S(X)$ è vera per ogni X. I prossimi due teoremi sono esempi di enunciati dimostrabili relativi ad alberi ed espressioni.

Teorema 1.21 Ogni albero ha un nodo in più rispetto ai suoi lati.

DIMOSTRAZIONE L'enunciato formale $S(T)$ che dobbiamo dimostrare per induzione strutturale è: "se T è un albero e T ha n nodi ed e lati, allora $n = e + 1$".

BASE Il caso di base si ha quando T è un nodo singolo. Allora $n = 1$ e $e = 0$, dunque la relazione $n = e + 1$ è valida.

INDUZIONE Sia T un albero costituito, secondo il passo induttivo della definizione, dal nodo radice N e da k alberi più piccoli $T_1, T_2, \ldots, T_k$. Possiamo supporre che gli enunciati $S(T_i)$ siano validi per $i = 1, 2, \ldots, k$. In altre parole, supponendo che T_i abbia n_i nodi ed e_i lati, si ha $n_i = e_i + 1$.

I nodi di T sono il nodo N e tutti i nodi degli alberi T_i. Esistono dunque $1 + n_1 + n_2 + \cdots + n_k$ nodi in T. I lati di T sono i k lati che sono stati aggiunti esplicitamente nel passo di induzione, più i lati di tutti i T_i. Perciò T ha

$$k + e_1 + e_2 + \cdots + e_k \tag{1.10}$$

lati. Se si sostituisce $e_i + 1$ a n_i nel conteggio del numero di nodi di T, si trova che T ha

$$1 + [e_1 + 1] + [e_2 + 1] + \cdots + [e_k + 1] \tag{1.11}$$

Fondamenti intuitivi dell'induzione strutturale

Possiamo suggerire in termini informali perché l'induzione strutturale è un metodo dimostrativo valido. Consideriamo una definizione ricorsiva in cui si asserisce che le strutture $X_1, X_2, \ldots$, prese una per volta, soddisfano la definizione. Gli elementi di base vengono per primi, e il fatto che X_i appartenga a quell'insieme di strutture deve dipendere solo dall'insieme delle strutture che precedono X_i nell'elenco. Da questo punto di vista un'induzione strutturale non è altro che un'induzione sull'intero n dell'enunciato $S(X_n)$. Quest'induzione può avere la forma generalizzata discussa nel Paragrafo 1.4.2, con casi di base multipli e un passo induttivo che usa tutti i casi precedenti dell'enunciato. Tuttavia si deve tenere presente, come spiegato nel Paragrafo 1.4.1, che questa intuizione non è una dimostrazione formale. Infatti dobbiamo supporre la validità di questo principio induttivo come è stato fatto con la validità del principio induttivo originale nel detto paragrafo.

nodi. Poiché ci sono k termini "+1" in (1.11), possiamo riscriverla:

$$k + 1 + e_1 + e_2 + \cdots + e_k \tag{1.12}$$

Quest'espressione vale esattamente 1 più dell'espressione (1.10) che era stata data per il numero di lati di T. Perciò T ha un nodo in più del numero dei suoi lati. □

Teorema 1.22 Ogni espressione ha un numero uguale di parentesi aperte e chiuse.

DIMOSTRAZIONE In termini formali dimostriamo l'enunciato $S(G)$ su qualunque espressione G che sia definita dalla ricorsione dell'esempio 1.20: il numero delle parentesi aperte e chiuse in G è lo stesso.

BASE Se G è definita dalla base, allora G è un numero o una variabile. Queste espressioni hanno zero parentesi aperte e zero parentesi chiuse, dunque il numero è uguale.

INDUZIONE Esistono tre regole per costruire G secondo il passo induttivo nella definizione.

1. $G = E + F$.

2. $G = E * F$.

3. $G = (E)$.

Possiamo supporre che $S(E)$ e $S(F)$ siano vere; in altre parole E ha lo stesso numero di parentesi aperte e chiuse, diciamo n, e similmente F ha lo stesso numero di parentesi aperte e chiuse, diciamo m. Allora possiamo calcolare il numero di parentesi aperte e chiuse in G per ognuno dei tre casi come segue.

1. Se $G = E + F$, allora G ha $n + m$ parentesi aperte e $n + m$ parentesi chiuse; n provenienti da E ed m da F.

2. Se $G = E * F$, il conteggio delle parentesi di G dà nuovamente $n+m$ per ciascuna parentesi, per la stessa ragione del caso (1).

3. Se $G = (E)$, allora ci sono $n+1$ parentesi aperte in G, una mostrata esplicitamente ed n in E. Analogamente ci sono $n + 1$ parentesi chiuse in G; una è esplicita, le altre sono in E.

In ognuno dei tre casi si vede che il numero delle parentesi aperte e chiuse in G è lo stesso. Questa osservazione completa il passo induttivo e termina la dimostrazione. □

1.4.4 Induzione mutua

Talvolta non è possibile dimostrare un singolo enunciato per induzione, ma si deve dimostrare congiuntamente un gruppo di enunciati $S_1(n), S_2(n), \ldots, S_k(n)$ per induzione su n. La teoria degli automi fornisce molte situazioni di questo genere. Nell'Esempio 1.23 illustriamo un tipico caso in cui si deve spiegare che cosa fa un automa dimostrando un gruppo di enunciati, uno per ogni stato. Tali enunciati indicano per quali sequenze di input l'automa passa in ognuno degli stati.

A rigor di termini, dimostrare un gruppo di enunciati non differisce dal dimostrare la *congiunzione* (AND logico) di tutti gli enunciati. Per esempio il gruppo di enunciati $S_1(n), S_2(n), \ldots, S_k(n)$ può essere sostituito dal singolo enunciato

$$S_1(n) \text{ AND } S_2(n) \text{ AND } \cdots \text{ AND } S_k(n)$$

Tuttavia, quando bisogna dimostrare molti enunciati effettivamente indipendenti, di solito conviene tenere separati gli enunciati e dimostrare per ciascuno la rispettiva base e il passo induttivo. Questo tipo di dimostrazione è detto *induzione mutua*. Un esempio illustrerà i passi necessari.

Esempio 1.23 Consideriamo l'interruttore *on/off*, presentato come automa nell'Esempio 1.1. L'automa stesso è riprodotto nella Figura 1.8. Dato che la pressione del pulsante cambia lo stato tra *on* e *off*, e l'interruttore parte dallo stato *off*, ci si aspetta che i seguenti enunciati descrivano insieme il funzionamento dell'interruttore.

$S_1(n)$: l'automa si trova nello stato *off* dopo n pressioni se e solo se n è pari.

$S_2(n)$: l'automa si trova nello stato *on* dopo n pressioni se e solo se n è dispari.

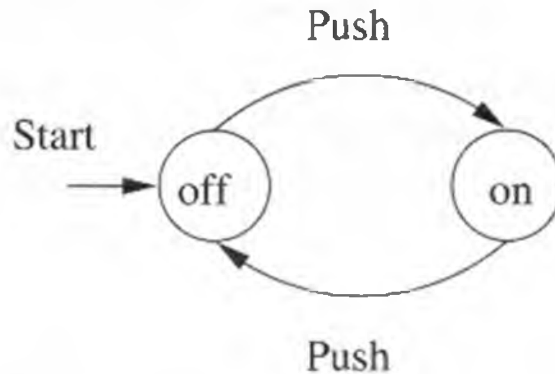


Figura 1.8 L'automa della Figura 1.1.

Sapendo che un numero n non può essere allo stesso tempo pari e dispari, si potrebbe supporre che S_1 implichi S_2, e viceversa. Tuttavia ciò che non è sempre vero di un automa è che si trovi in un solo stato. Di fatto l'automa della Figura 1.8 è sempre esattamente in un solo stato, ma questo deve essere dimostrato come parte dell'induzione mutua.

Diamo la base e l'induzione delle dimostrazioni degli enunciati $S_1(n)$ e $S_2(n)$. Le dimostrazioni dipendono da diverse proprietà degli interi pari e dispari: se si aggiunge o toglie 1 da un intero pari si ottiene un intero dispari, e se si aggiunge o toglie 1 da un intero dispari si ottiene un intero pari.

BASE Per la base scegliamo $n = 0$. Dato che ci sono due enunciati, ognuno dei quali deve essere dimostrato in entrambe le direzioni (in quanto S_1 ed S_2 sono ciascuno enunciati "se-e-solo-se"), in effetti ci sono quattro casi per la base e altrettanti per l'induzione.

1. [S_1; se] Dato che 0 è pari, dobbiamo dimostrare che dopo 0 pressioni l'automa della Figura 1.8 si trova nello stato *off*. Poiché questo è lo stato iniziale, l'automa si trova effettivamente nello stato *off* dopo 0 pressioni.

2. [S_1; solo-se] L'automa si trova nello stato *off* dopo 0 pressioni, perciò dobbiamo mostrare che 0 è pari. Ma 0 è pari per la definizione di "pari", dunque non resta altro da dimostrare.

3. [S_2; se] L'ipotesi della parte "se" di S_2 è che 0 sia dispari. Essendo quest'ipotesi H falsa, qualsiasi enunciato della forma "se H allora C" è vero, come è stato discusso nel Paragrafo 1.3.2. Di conseguenza è valida anche questa parte della base.

4. [S_2; solo-se] Anche l'ipotesi che l'automa sia nello stato *on* dopo 0 pressioni è falsa, in quanto l'unico modo di pervenire allo stato *on* è seguire un arco etichettato

Push, il che richiede almeno una pressione sul pulsante. Poiché l'ipotesi è falsa, possiamo concludere nuovamente che l'enunciato se-allora è vero.

INDUZIONE Ora supponiamo che $S_1(n)$ e $S_2(n)$ siano vere, e proviamo a dimostrare $S_1(n+1)$ e $S_2(n-1)$. Anche questa dimostrazione si suddivide in quattro parti.

1. [$S_1(n+1)$; se] L'ipotesi per questa parte è che $n+1$ sia pari. Di conseguenza n è dispari. La parte "se" dell'enunciato $S_2(n)$ dice che dopo n pressioni l'automa si trova nello stato *on*. L'arco da *on* a *off* recante l'etichetta *Push* dice che la $(n+1)$-esima pressione farà passare l'automa nello stato *off*. Ciò completa la dimostrazione della parte "se" di $S_1(n+1)$.

2. [$S_1(n+1)$; solo-se] L'ipotesi è che l'automa si trovi nello stato *off* dopo $n+1$ pressioni. L'esame dell'automa indica che l'unico modo di pervenire allo stato *off* è di trovarsi nello stato *on* e di ricevere un input *Push*. Perciò, se ci si trova nello stato *off* dopo $n+1$ pressioni, l'automa deve essersi trovato nello stato *on* dopo n pressioni. Allora possiamo ricorrere alla parte "solo-se" dell'enunciato $S_2(n)$ per concludere che n è dispari. Dunque $n+1$ è pari, come si voleva dimostrare per la parte solo-se di $S_1(n+1)$.

3. [$S_2(n+1)$; se] Questa parte è essenzialmente uguale alla parte (1), con i ruoli di S_1 ed S_2 e con i ruoli di "dispari" e "pari" scambiati. Il lettore dovrebbe essere in grado di costruire agevolmente questa parte della dimostrazione.

4. [$S_2(n+1)$; solo-se] Questa parte è essenzialmente uguale alla parte (2), con i ruoli di S_1 ed S_2 e con i ruoli di "dispari" e "pari" scambiati.

□

Dall'esempio 1.23 si può ricavare il modello di tutte le induzioni mutue.

- Ogni enunciato dev'essere dimostrato separatamente nella base e nel passo induttivo.

- Se si tratta di enunciati se-e-solo-se, allora entrambe le direzioni di ogni enunciato devono essere dimostrate, sia nella base sia nel passo induttivo.

1.5 I concetti centrali della teoria degli automi

In questo paragrafo verranno introdotte le definizioni dei termini più importanti che permeano la teoria degli automi. Questi concetti sono "alfabeto" (un insieme di simboli), "stringa" (una lista di simboli di un alfabeto) e "linguaggio" (un insieme di stringhe dallo stesso alfabeto).

1.5.1 Alfabeti

Un *alfabeto* è un insieme finito e non vuoto di simboli. Per indicare un alfabeto si usa convenzionalmente il simbolo Σ. Fra gli alfabeti più comuni citiamo:

1. $\Sigma = \{0, 1\}$, l'alfabeto *binario*
2. $\Sigma = \{a, b, \ldots, z\}$, l'insieme di tutte le lettere minuscole
3. l'insieme di tutti i caratteri ASCII o l'insieme di tutti i caratteri ASCII stampabili.

1.5.2 Stringhe

Una *stringa* (o *parola*) è una sequenza finita di simboli scelti da un alfabeto. Per esempio 01101 è una stringa sull'alfabeto binario $\Sigma = \{0, 1\}$. La stringa 111 è un'altra stringa sullo stesso alfabeto.

La stringa vuota

La *stringa vuota* è la stringa composta da zero simboli. Questa stringa, indicata con ϵ, è una stringa che può essere scelta da un qualunque alfabeto.

Lunghezza di una stringa

Spesso è utile classificare le stringhe a seconda della loro *lunghezza*, vale a dire il numero di posizioni per i simboli della stringa. Per esempio 01101 ha lunghezza 5. Comunemente si dice che la lunghezza di una stringa è "il numero di simboli" nella stringa stessa; quest'enunciato è un'espressione accettabile colloquialmente, ma formalmente non corretta. Infatti ci sono solo due simboli, 0 e 1, nella stringa 01101, ma ci sono 5 *posizioni*, e la lunghezza della stringa è 5. Tuttavia ci si può aspettare di vedere usato di solito "il numero di simboli" quando invece s'intende il "numero di posizioni".

La notazione standard per la lunghezza di una stringa w è $|w|$. Per esempio $|011| = 3$ e $|\epsilon| = 0$.

Potenze di un alfabeto

Se Σ è un alfabeto possiamo esprimere l'insieme di tutte le stringhe di una certa lunghezza su tale alfabeto usando una notazione esponenziale. Definiamo Σ^k come l'insieme di stringhe di lunghezza k, con simboli tratti da Σ.

Esempio 1.24 Si noti che $\Sigma^0 = \{\epsilon\}$ per qualsiasi alfabeto Σ. In altre parole ϵ è la sola stringa di lunghezza 0.

Se $\Sigma = \{0, 1\}$, allora $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$,

> **Convenzioni tipografiche per simboli e stringhe**
>
> Useremo comunemente lettere minuscole prese dall'inizio dell'alfabeto (oppure cifre) per denotare i simboli, e lettere minuscole dalla fine dell'alfabeto, di solito w, x, y e z, per denotare le stringhe. Si consiglia di abituarsi a tale convenzione perché facilita il riconoscimento del tipo di oggetto in discussione.

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

e così via. Si osservi che si crea una certa confusione tra Σ e Σ^1. Il primo è un alfabeto: i suoi membri, 0 e 1, sono simboli. Il secondo è un insieme di stringhe; i suoi membri sono le stringhe 0 e 1, ognuna delle quali è di lunghezza 1. Qui non cercheremo di usare due notazioni distinte per i due insiemi, ma ci affideremo piuttosto al contesto per chiarire se $\{0, 1\}$ o insiemi simili sono alfabeti o insiemi di stringhe. □

L'insieme di tutte le stringhe su un alfabeto Σ viene indicato convenzionalmente con Σ^*. Per esempio $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Formulato altrimenti

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Talvolta si desidera escludere la stringa vuota da un insieme di stringhe. L'insieme di stringhe non vuote sull'alfabeto Σ è indicato con Σ^+. Di conseguenza valgono le due equivalenze seguenti.

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$
- $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Concatenazione di stringhe

Siano x e y stringhe. Allora xy denota la *concatenazione* di x e y, vale a dire la stringa formata facendo una copia di x e facendola seguire da una copia di y. Più precisamente, se x è la stringa composta da i simboli $x = a_1a_2 \cdots a_i$ e y è la stringa composta da j simboli $y = b_1b_2 \cdots b_j$, allora xy è la stringa di lunghezza $i + j$: $xy = a_1a_2 \cdots a_ib_1b_2 \cdots b_j$.

Esempio 1.25 Poniamo $x = 01101$ e $y = 110$. Allora $xy = 01101110$ e $yx = 11001101$. Per qualunque stringa w sono valide le equazioni $\epsilon w = w\epsilon = w$. In altre parole ϵ è l'*identità per la concatenazione*, dato che, concatenata con una qualunque stringa, dà come risultato la stringa stessa (nello stesso modo in cui 0, l'identità per l'addizione, può essere sommato a qualunque numero x e dà come risultato x). □

1.5.3 Linguaggi

Un insieme di stringhe scelte da Σ^*, dove Σ è un particolare alfabeto, si dice un *linguaggio*. Se Σ è un alfabeto e $L \subseteq \Sigma^*$, allora L è un *linguaggio su* Σ. Si noti che un linguaggio su Σ non deve necessariamente includere stringhe con tutti i simboli di Σ; perciò, una volta che abbiamo stabilito che L è un linguaggio su Σ, sappiamo anche che è un linguaggio su qualunque alfabeto includa Σ.

La scelta del termine "linguaggio" può sembrare strana; d'altra parte i linguaggi comuni possono essere visti come insiemi di stringhe. Un esempio è l'inglese, in cui la raccolta delle parole accettabili della lingua è un insieme di stringhe sull'alfabeto che consiste di tutte le lettere. Un altro esempio è il C, o qualunque altro linguaggio di programmazione, in cui i programmi accettabili sono un sottoinsieme di tutte le possibili stringhe che si possono formare con l'alfabeto del linguaggio. Quest'alfabeto è un sottoinsieme dei caratteri ASCII. L'alfabeto può variare leggermente tra diversi linguaggi di programmazione, ma generalmente include le lettere maiuscole e minuscole, le cifre, la punteggiatura e i simboli matematici.

Nello studio degli automi ci si imbatte in molti altri linguaggi. Alcuni sono esempi astratti, come quelli che elenchiamo.

1. Il linguaggio di tutte le stringhe che consistono di n 0 seguiti da n 1, per $n \geq 0$: $\{\epsilon, 01, 0011, 000111, \ldots\}$.

2. L'insieme delle stringhe con un uguale numero di 0 e di 1:

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

3. L'insieme dei numeri binari il cui valore è un numero primo:

$$\{10, 11, 101, 111, 1011, \ldots\}$$

4. Σ^* è un linguaggio per qualunque alfabeto Σ.

5. $\emptyset$, il linguaggio vuoto, è un linguaggio rispetto a qualunque alfabeto.

6. Anche $\{\epsilon\}$, il linguaggio che consta della sola stringa vuota, è un linguaggio rispetto a ogni alfabeto. Si noti che $\emptyset \neq \{\epsilon\}$; il primo non ha stringhe mentre il secondo ne ha una.

L'unica restrizione di rilievo sui linguaggi è che tutti gli alfabeti sono finiti. Di conseguenza i linguaggi, sebbene possano avere un numero infinito di stringhe, devono consistere di stringhe tratte da un determinato alfabeto finito.

I formatori di insiemi come un modo di definire linguaggi

Spesso si descrive un linguaggio usando un *formatore di insiemi*:

$$\{w \mid \text{enunciato su } w\}$$

Quest'espressione va letta come "l'insieme delle parole w tali che vale l'enunciato su w a destra della barra verticale". Alcuni esempi:

1. $\{w \mid w$ consiste di un numero uguale di 0 e di 1$\}$
2. $\{w \mid w$ è un intero binario che è primo$\}$
3. $\{w \mid w$ è un programma C sintatticamente corretto$\}$.

Si suole anche sostituire w con un'espressione con parametri e descrivere le stringhe del linguaggio enunciando le condizioni sui parametri. Ecco alcuni esempi, il primo con un parametro n, il secondo con i parametri i e j.

1. $\{0^n1^n \mid n \geq 1\}$. Leggi "l'insieme di 0 elevato alla n, 1 alla n tale che n è maggiore o uguale a 1"; questo linguaggio è formato dalle stringhe $\{01, 0011, 000111, \ldots\}$. Si noti che, come con gli alfabeti, è possibile elevare un singolo simbolo alla potenza n per rappresentare n copie di quel simbolo.
2. $\{0^i1^j \mid 0 \leq i \leq j\}$. Questo linguaggio consiste di stringhe con un certo numero di 0 (eventualmente nessuno) seguiti da almeno altrettanti 1.

1.5.4 Problemi

Nella teoria degli automi un *problema* è la questione se una data stringa sia o no membro di un particolare linguaggio. Come si vedrà, tutto ciò che definiamo correntemente un "problema" può essere espresso in termini di appartenenza a un linguaggio. In termini più precisi, se Σ è un alfabeto e L è un linguaggio su Σ, allora il problema L è:

- data una stringa w in Σ^*, decidere se w appartiene a L.

Esempio 1.26 Il problema di verificare se un numero è primo si può esprimere con il linguaggio L_p, che consiste di tutte le stringhe binarie il cui valore come numero binario è un numero primo. In altre parole, data una stringa di 0 e di 1, diremo "sì" se la stringa

Linguaggio o problema

Linguaggi e problemi sono in realtà la stessa cosa. Scegliere l'uno o l'altro termine dipende dal punto di vista. Se ci si occupa di stringhe prese come tali, per esempio quelle dell'insieme $\{0^n1^n \mid n \geq 1\}$, allora si è portati a pensare all'insieme di stringhe come a un linguaggio. Negli ultimi capitoli del libro tenderemo ad assegnare una "semantica" alle stringhe, per esempio pensandole come codifiche di grafi, espressioni logiche o persino numeri interi. Nei casi in cui ci occupiamo di quanto viene rappresentato dalla stringa anziché della stringa stessa, tenderemo a considerare un insieme di stringhe come un problema.

è la rappresentazione binaria di un numero primo e "no" in caso contrario. Per alcune stringhe questa decisione è facile. Per esempio 0011101 non può essere la rappresentazione di un primo, per la semplice ragione che ogni intero, con l'eccezione di 0, ha una rappresentazione binaria che comincia con 1. D'altra parte è meno palese se la stringa 11101 appartenga a L_p. Dunque qualunque soluzione a questo problema dovrà avvalersi di risorse computazionali di qualche tipo: tempo e spazio, per esempio. □

Un aspetto potenzialmente insoddisfacente della nostra definizione di "problema" è che comunemente non si pensa a un problema in termini di decisione (p è o non è vero?), bensì come se si trattasse di una richiesta di computare o trasformare un certo input (trovare il miglior modo di svolgere un dato compito). Per esempio il compito di un *parser* in un compilatore C può essere visto come un problema nel nostro senso formale, in cui si dà una stringa ASCII e si chiede di decidere se la stringa sia o no un elemento di L_c, l'insieme dei programmi C validi. Tuttavia un *parser* non si limita a decidere: produce infatti un albero sintattico, voci in una tabella di simboli ed eventualmente altro. Non solo: il compilatore nel suo insieme risolve il problema di trasformare un programma C in codice oggetto per una certa macchina, ben più di una semplice risposta "sì" o "no" alla domanda sulla validità di un programma.

Nonostante ciò, la definizione di problema come linguaggio ha resistito nel tempo come il modo più opportuno di trattare le questioni cruciali della teoria della complessità. In questa teoria l'interesse è volto a scoprire limiti inferiori della complessità di determinati problemi. Di particolare importanza sono le tecniche per dimostrare che certi problemi non possono essere risolti in tempo meno che esponenziale nella dimensione del loro input. Risulta che le versioni "sì-no" oppure "basate su linguaggi" di problemi noti sono in questo senso altrettanto difficili delle versioni "risolvi".

In altre parole, se possiamo dimostrare che è difficile decidere se una data stringa appartiene al linguaggio L_X delle stringhe valide in un linguaggio di programmazione X,

allora è evidente che non sarà più facile tradurre in codice oggetto i programmi in linguaggio X: se fosse facile generare il codice, allora si potrebbe eseguire il traduttore e concludere che l'input è un membro valido di L_X nel momento in cui il traduttore riuscisse a produrre il codice oggetto. Dato che il passo finale nel determinare se il codice oggetto è stato prodotto non può essere difficile, si può usare l'algoritmo rapido di generazione del codice oggetto per decidere efficientemente dell'appartenenza a L_X. Così si giunge a contraddire l'assunto che provare l'appartenenza a L_X è difficile. Abbiamo una dimostrazione per assurdo dell'enunciato "se provare l'appartenenza a L_X è difficile, allora compilare programmi nel linguaggio di programmazione X è difficile".

Questa tecnica, cioè mostrare quanto sia complesso un problema usando un suo algoritmo, che si presume efficiente, per risolvere efficientemente un altro problema di cui si conosce già la difficoltà, è detta "riduzione" del secondo problema al primo. Si tratta di uno strumento essenziale nello studio della complessità dei problemi e il suo utilizzo è molto agevolato dalla nozione che i problemi sono questioni di appartenenza a un linguaggio anziché questioni di tipo più generale.

1.6 Riepilogo

- *Automi a stati finiti*: gli automi a stati finiti sono composti di stati e transizioni tra gli stati come reazioni agli input. Sono utili per la costruzione di diversi tipi di software, tra cui il componente per l'analisi lessicale di un compilatore e sistemi per la verifica della correttezza di circuiti o protocolli.

- *Espressioni regolari*: si tratta di una notazione strutturale per la descrizione di pattern che possono essere rappresentati da automi a stati finiti. Vengono usati in molti tipi comuni di software, tra cui strumenti per la ricerca di pattern in testi o in nomi di file.

- *Grammatiche libere dal contesto*: si tratta di un'importante notazione per descrivere la struttura di linguaggi di programmazione e insiemi di stringhe correlati; sono usate per costruire il *parser* di un compilatore.

- *Macchine di Turing*: sono automi che forniscono un modello della capacità dei computer reali. Tali macchine permettono di studiare la decidibilità, cioè che cosa un computer può fare e che cosa non può fare. Grazie alle macchine di Turing è inoltre possibile distinguere tra problemi trattabili, ossia quelli che possono essere risolti in tempo polinomiale, e problemi intrattabili, quelli che invece non possono essere risolti in tempo polinomiale.

- ✦ *Dimostrazioni deduttive*: questo metodo dimostrativo di base procede elencando enunciati che sono dati come veri oppure seguono logicamente da uno degli enunciati precedenti.

- ✦ *Dimostrazioni di enunciati se-allora*: molti teoremi si presentano nella forma "se (qualcosa) allora (qualcos'altro)". L'enunciato o gli enunciati che seguono "se" sono l'ipotesi, quelli che seguono "allora" sono la conclusione. Le dimostrazioni deduttive di enunciati se-allora cominciano dall'ipotesi e continuano con enunciati che seguono logicamente dall'ipotesi e dagli enunciati precedenti, finché si dimostra la conclusione come uno degli enunciati.

- ✦ *Dimostrazioni di enunciati se-e-solo-se*: altri teoremi hanno la forma "(qualcosa) se e solo se (qualcos'altro)". Si dimostrano provando gli enunciati se-allora in entrambe le direzioni. Un analogo tipo di teorema asserisce l'uguaglianza di due insiemi descritti in modi diversi; la dimostrazione si ottiene provando che ognuno dei due insiemi è contenuto nell'altro.

- ✦ *Dimostrazioni per contronominale*: talvolta è più facile dimostrare un enunciato nella forma "se H allora C" dimostrando l'enunciato equivalente: "se non C allora non H". Il secondo è detto il contronominale del primo.

- ✦ *Dimostrazioni per assurdo*: in altri casi conviene dimostrare l'enunciato "se H allora C" dimostrando "se H e non C allora (qualcosa che si sa essere falso)". Una dimostrazione di questo tipo è detta "per assurdo".

- ✦ *Controesempi*: talora si deve provare che un certo enunciato non è vero. Se l'enunciato ha uno o più parametri, allora si può mostrare che è falso in generale fornendo solo un controesempio, vale a dire un assegnamento di valori ai parametri che rende falso l'enunciato.

- ✦ *Dimostrazioni induttive*: un enunciato che ha un parametro intero n può essere spesso dimostrato per induzione su n. Si dimostra che l'enunciato è vero per la base, cioè un numero finito di casi per determinati valori di n, e poi si dimostra il passo induttivo: che se l'enunciato è vero per valori fino a n allora è vero per $n + 1$.

- ✦ *Induzioni strutturali*: in alcune situazioni, comprese molte di quelle trattate in questo libro, il teorema che dev'essere dimostrato induttivamente riguarda un costrutto definito ricorsivamente, come gli alberi. Si può dimostrare un teorema su oggetti così costruiti per induzione sul numero di passi compiuti nella costruzione. Questo tipo di induzione è detto strutturale.

- ✦ *Alfabeti*: un alfabeto è un insieme finito di simboli.

- ✦ *Stringhe*: una stringa è una sequenza di simboli di lunghezza finita.

- *Linguaggi e problemi*: un linguaggio è un insieme (eventualmente infinito) di stringhe, formate da simboli tratti dallo stesso alfabeto. Quando le stringhe di un linguaggio devono essere interpretate in qualche modo, la questione se una stringa appartenga al linguaggio viene indicata talora con il termine problema.

1.7 Bibliografia

Per una trattazione approfondita del materiale presentato in questo capitolo, inclusi i concetti matematici che stanno alla base dell'informatica, si consiglia [1].

1. A. V. Aho, J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York 1994; in it. *Fondamenti di informatica*, Zanichelli Editore, Bologna, 1994.

Capitolo 2

Automi a stati finiti

Questo capitolo presenta la classe dei linguaggi detti "regolari". Tali linguaggi sono esattamente quelli che possono essere descritti dagli automi a stati finiti, e che abbiamo già brevemente trattato nel Paragrafo 1.1.1. Definiremo gli automi a stati finiti in termini formali dopo un dettagliato esempio che fornirà la motivazione per lo studio successivo.

Come detto precedentemente, un automa a stati finiti ha un insieme di stati e un "controllo" che si muove da stato a stato in risposta a input esterni. Una distinzione cruciale tra le classi di automi a stati finiti riguarda il controllo: se è deterministico, ossia se l'automa non può essere in più di uno stato per volta, oppure non deterministico, ossia se l'automa può trovarsi in più stati contemporaneamente. Vedremo che l'aggiunta del non determinismo non permette di allargare la classe dei linguaggi definibili da automi a stati finiti deterministici, ma può essere più efficace descrivere un'applicazione usando un automa non deterministico. In effetti il non determinismo permette di "programmare" soluzioni di un problema usando un linguaggio ad alto livello. L'automa a stati finiti non deterministico viene poi "compilato", attraverso un algoritmo che verrà spiegato in questo capitolo, in un automa deterministico che può essere "eseguito" da un computer convenzionale.

Il capitolo si chiude con lo studio di automi non deterministici estesi, che hanno la scelta supplementare di fare una transizione da uno stato all'altro spontaneamente, ossia sulla stringa vuota come input. Anche questi automi accettano solo linguaggi regolari, ma se ne vedrà l'importanza nel Capitolo 3, dove studieremo le espressioni regolari e la loro equivalenza agli automi.

Lo studio dei linguaggi regolari prosegue nel Capitolo 3. Qui viene introdotto un altro modo importante di descriverli: la notazione algebrica conosciuta come espressioni regolari. Dopo aver discusso le espressioni regolari e aver dimostrato la loro equivalenza agli automi a stati finiti, nel Capitolo 4 useremo sia gli automi sia le espressioni regolari come strumenti per mostrare alcune proprietà importanti dei linguaggi regolari. Esempi di tali proprietà sono le proprietà di "chiusura", che permettono di affermare che un linguaggio

è regolare perché sappiamo che uno o più altri linguaggi sono regolari, e le proprietà di "decisione". Queste ultime sono algoritmi per rispondere a domande sugli automi oppure sulle espressioni regolari, per esempio se due automi o due espressioni rappresentino lo stesso linguaggio.

2.1 Una descrizione informale degli automi a stati finiti

In questo paragrafo affronteremo un esempio di problema reale, nella cui soluzione gli automi a stati finiti svolgono un ruolo importante. Ci occupiamo di protocolli a supporto del "denaro elettronico", cioè file che possono essere usati da un cliente per pagare merci tramite Internet e che un venditore può accettare con la certezza che il "denaro" è reale. Il venditore deve essere sicuro che il file inviatogli non è stato né falsificato né copiato, mentre il cliente ne conserva una copia da riutilizzare.

La non falsificabilità del file deve essere assicurata da una banca e da un meccanismo crittografico: in altre parole, un terzo attore, la banca, deve emettere e cifrare i file che rappresentano il denaro in modo da evitare problemi. La banca ha un secondo compito importante: deve tenere un database di tutto il denaro valido emesso, in modo da poter verificare e dare conferma a un un negozio che il file ricevuto rappresenta denaro reale e può dunque essere accreditato sul conto dello stesso. Non ci occupiamo qui degli aspetti crittografici della questione, né di come la banca possa conservare ed estrarre quelli che potrebbero essere miliardi di banconote elettroniche. Questi problemi non dovrebbero costituire ostacoli a lungo termine per la diffusione del denaro elettronico; sebbene su scala limitata, abbiamo esempi del suo impiego sin dai tardi anni '90.

Nondimeno, al fine di poter usare denaro elettronico, si devono ideare protocolli in modo tale che il denaro possa essere manipolato come l'utente desidera. Dato che i sistemi monetari allettano sempre la frode, è necessario verificare i criteri da adottare riguardo all'impiego del denaro. Occorre cioè dimostrare che le sole cose che possono accadere sono quelle che noi vogliamo che accadano, ossia quelle che non consentono a un utente privo di scrupoli di sottrarre il denaro altrui o addirittura di "fabbricarlo".

Nel corso del paragrafo illustreremo un esempio molto semplice di protocollo (inadatto) per il denaro elettronico, lo modelleremo con automi a stati finiti e dimostreremo come applicare le costruzioni sugli automi per verificare protocolli (oppure, in questo caso, per scoprire che il protocollo contiene un errore).

2.1.1 Le regole fondamentali

Ci sono tre partecipanti: il cliente, il negozio e la banca. Per semplicità si suppone che esista un solo file che rappresenta il denaro. Il cliente può decidere di trasferire questo file di denaro virtuale al negozio, che lo riscatterà dalla banca (ossia farà sì che la banca emani un nuovo file per il negozio anziché per il cliente) e spedirà la merce al cliente. Inoltre il

cliente ha l'opzione di poter cancellare il file, cioè può chiedere alla banca di ricollocare il denaro nel suo conto, rendendolo non spendibile. L'interazione fra i tre partecipanti è dunque limitata a cinque eventi.

1. Il cliente può decidere di *pagare*, cioè di inviare il denaro al negozio.

2. Il cliente può decidere di *annullare*. Il denaro viene inviato alla banca attraverso un messaggio contenente le istruzioni di accredito dell'importo sul conto del cliente.

3. Il negozio può *consegnare* la merce al cliente.

4. Il negozio può *riscattare* il denaro, ossia il denaro viene mandato alla banca con la richiesta di trasferirne il valore al negozio stesso.

5. La banca può *trasferire* il denaro creando un nuovo file di denaro virtuale, adeguatamente cifrato, e inviandolo al negozio.

2.1.2 Il protocollo

I tre partecipanti devono decidere attentamente come comportarsi per evitare situazioni non desiderate. Nell'esempio facciamo una supposizione ragionevole: non si può dare per scontato che il cliente agisca onestamente. In particolare il cliente potrebbe tentare di copiare il file di denaro virtuale, usarlo per pagare più volte, oppure pagare e poi annullare il denaro, ottenendo così la merce gratis.

La banca, per sua natura, deve agire responsabilmente. In particolare deve assicurarsi che due negozi non possano riscattare lo stesso file di denaro virtuale e non deve permettere che il denaro venga contemporaneamente annullato e riscattato. Anche il negozio deve cautelarsi. Soprattutto non deve inoltrare la merce finché non è sicuro di aver ricevuto in cambio denaro valido.

Si possono rappresentare protocolli di questo tipo come automi a stati finiti. Ogni stato rappresenta una situazione in cui uno dei partecipanti può trovarsi. In altre parole lo stato "ricorda" che certi eventi importanti hanno avuto luogo e che altri non sono ancora avvenuti. Le transizioni tra gli stati avvengono quando si verifica uno dei cinque eventi descritti sopra. Penseremo questi eventi come "esterni" agli automi che rappresentano i tre partecipanti, sebbene ogni partecipante sia responsabile dell'avvio di uno o più eventi. Nell'ambito del problema è importante la sequenza degli eventi che possono accadere, non chi può avviarli.

La Figura 2.1 rappresenta i tre partecipanti per mezzo di automi. Nel diagramma illustriamo solo gli eventi che influenzano ciascun partecipante. Per esempio l'azione *pay* (pagare) riguarda soltanto il cliente e il negozio. La banca non sa che il denaro è stato mandato dal cliente al negozio; lo scopre quando il negozio compie l'azione *redeem* (riscattare).

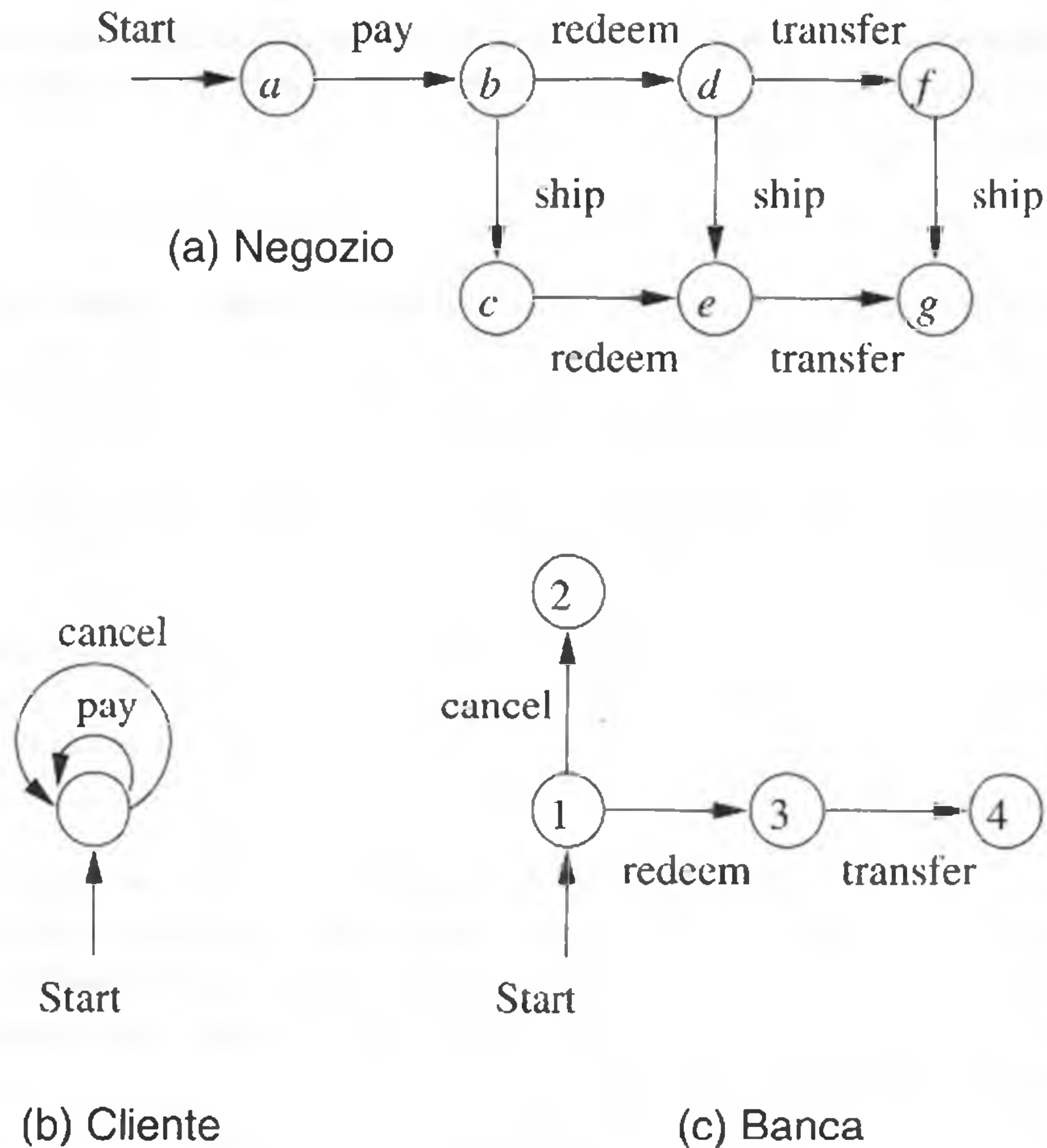


Figura 2.1 Automi a stati finiti che rappresentano un cliente, un negozio e una banca.

Esaminiamo per primo l'automa (c), che rappresenta la banca. Lo stato iniziale è lo stato 1, che corrisponde alla situazione in cui la banca ha emesso il file di denaro virtuale in questione, ma non ha ricevuto la richiesta di riscattarlo oppure di annullarlo. Se il cliente invia alla banca la richiesta *cancel* (annullare), allora la banca ricolloca il denaro nel conto del cliente ed entra nello stato 2. Questo stato denota la situazione in cui il denaro è stato annullato. Per la responsabilità che le compete, dal momento che non può permettere che il cliente annulli di nuovo oppure spenda lo stesso denaro, dopo esservi entrata la banca non lascerà lo stato 2.[1]

Quando la banca si trova nello stato 1 può ricevere anche una richiesta *redeem* dal

[1] Si badi che l'intera discussione ruota intorno a un unico file di denaro virtuale. In effetti la banca farà funzionare lo stesso protocollo con un gran numero di monete elettroniche. Poiché il meccanismo è lo stesso per ognuna, si può discutere il problema come se ne esistesse un solo esemplare.

negozio. In questo caso la banca passa allo stato 3 e, dopo un certo tempo, manda al negozio un messaggio *transfer* (trasferimento) con un nuovo file di denaro virtuale, che ora appartiene al negozio. Dopo aver spedito il messaggio di trasferimento, la banca passa allo stato 4, in cui non accetterà richieste *cancel* o *redeem* né svolgerà alcun'altra azione che riguardi il file di denaro virtuale in questione.

Consideriamo ora la Figura 2.1(a), l'automa che rappresenta le azioni del negozio. Mentre la banca fa sempre la cosa giusta, il sistema del negozio ha qualche difetto. Si immagini che la spedizione e le operazioni finanziarie siano svolte da processi distinti, cosicché c'è la possibilità che l'azione *ship* venga compiuta prima, dopo, oppure durante il riscatto del denaro elettronico. Questo tipo di meccanismo permette al negozio di trovarsi nella situazione di aver già inoltrato la merce e di scoprire solo in seguito che il denaro era falso.

Il negozio parte dallo stato a. Quando il cliente ordina la merce compiendo l'azione *pay*, il negozio entra nello stato b. In questo stato il negozio avvia sia il processo di spedizione sia quello di riscatto. Se la merce viene spedita prima, allora il negozio entra nello stato c, in cui deve ancora riscattare il denaro dalla banca e da questa ricevere il trasferimento di un file di denaro virtuale equivalente. In alternativa il negozio può mandare subito il messaggio *redeem*, entrando dunque nello stato d. Dallo stato d il negozio può effettuare la spedizione, entrando nello stato e, oppure ricevere il trasferimento di denaro dalla banca, entrando nello stato f. Dallo stato f ci si aspetta che il negozio spedisca infine la merce e passi allo stato g, dove la transazione è completata e non succede più niente. Nello stato e il negozio attende il trasferimento dalla banca. Disgraziatamente la merce è già stata spedita, e se il trasferimento non ha luogo il negozio subisce una perdita.

Da ultimo osserviamo l'automa che modella il cliente, Figura 2.1(b). Per indicare il fatto che il cliente "può fare qualunque cosa", questo automa ha solo uno stato. Il cliente può svolgere le azioni *pay* e *cancel* tutte le volte che vuole, in qualunque ordine, rimanendo nell'unico stato dopo ogni azione.

2.1.3 Automi che possono ignorare azioni

I tre automi nella Figura 2.1 rispecchiano il comportamento indipendente dei tre partecipanti, ma mancano alcune transizioni. Per esempio il negozio non viene influenzato da un messaggio *cancel*, così se il cliente esegue l'azione *cancel* il negozio dovrebbe rimanere nello stato in cui si trova. Nella definizione formale di automa a stati finiti, che vedremo nel Paragrafo 2.2, ogni volta che riceve un input X un automa deve seguire un arco etichettato X dallo stato in cui si trova a un nuovo stato. Perciò l'automa per il negozio ha bisogno di un ulteriore arco, da ogni stato a se stesso, etichettato *cancel*. In questo modo, quando si esegue l'azione *cancel*, l'automa del negozio può compiere una "transizione" su tale input, rimanendo nello stesso stato in cui si trovava. Senza questi archi supple-

mentari, a ogni occorrenza dell'azione *cancel*, l'automa negozio "morirebbe"; l'automa, cioè, non si troverebbe in alcuno stato, e ulteriori azioni gli sarebbero impossibili.

Un altro potenziale problema è che uno dei partecipanti possa, intenzionalmente o per errore, inviare un messaggio inaspettato: questo comportamento non deve far morire nessuno degli automi. Si supponga per esempio che il cliente decida di eseguire l'azione *pay* una seconda volta, mentre il negozio si trova nello stato *e*. Poiché lo stato non ha alcun arco in uscita con l'etichetta *pay*, l'automa del negozio morirebbe prima di poter ricevere il trasferimento dalla banca. Riassumendo, bisogna aggiungere agli automi della Figura 2.1 cicli su certi stati, con etichette per tutte le azioni che devono essere ignorate quando ci si trova in quegli stati. Gli automi completi sono rappresentati nella Figura 2.2. Per risparmiare spazio, si combinano le etichette su un unico arco invece di mostrare archi distinti con gli stessi estremi ma con etichette diverse. Si devono ignorare due tipi di azioni.

1. *Azioni che sono irrilevanti per il partecipante.* Come già visto, l'unica azione irrilevante per il negozio è *cancel*, dunque ognuno dei sette stati ha un ciclo etichettato *cancel*. Per la banca, sia *pay* sia *ship* sono irrilevanti, perciò si pone un arco etichettato *pay, ship* su ognuno degli stati della banca. Per il cliente, *ship*, *redeem* e *transfer* sono tutti irrilevanti, e pertanto si aggiungono archi con tali etichette. Di fatto il cliente rimane nel suo unico stato per ogni sequenza di input: l'automa non ha effetti sull'operazione del sistema globale. Il cliente è comunque un partecipante, dato che avvia le azioni *pay* e *cancel*. Tuttavia, come abbiamo detto, il fatto di provocare le azioni non ha nulla a che vedere con il comportamento degli automi.

2. *Azioni cui si deve impedire di uccidere un automa.* Come abbiamo visto, non possiamo permettere al cliente di uccidere l'automa del negozio eseguendo l'azione *pay* una seconda volta. Per questa ragione sono stati aggiunti cicli con l'etichetta *pay* a tutti gli stati, eccetto lo stato *a* (in cui l'azione è prevista e rilevante). Inoltre sono stati aggiunti cicli con l'etichetta *cancel* agli stati 3 e 4 della banca, allo scopo di impedire al cliente di uccidere l'automa di questa cancellando denaro già riscattato. Correttamente la banca ignora tale richiesta. In modo analogo gli stati 3 e 4 hanno un ciclo su *redeem*. Il negozio non dovrebbe cercare di riscattare lo stesso denaro due volte. Se lo fa, la banca ignora la seconda richiesta.

2.1.4 L'intero sistema come automa

Per ora abbiamo i modelli di comportamento dei tre partecipanti, ma non della loro interazione. Come detto sopra, dato che il cliente non ha vincoli sul comportamento, il suo automa ha un solo stato, e qualsiasi sequenza di eventi lo lascia in tale stato; in altre parole, non è possibile che il sistema nel suo insieme "muoia" perché l'automa del cliente non reagisce a un'azione. D'altra parte, sia il negozio sia la banca si comportano in modo

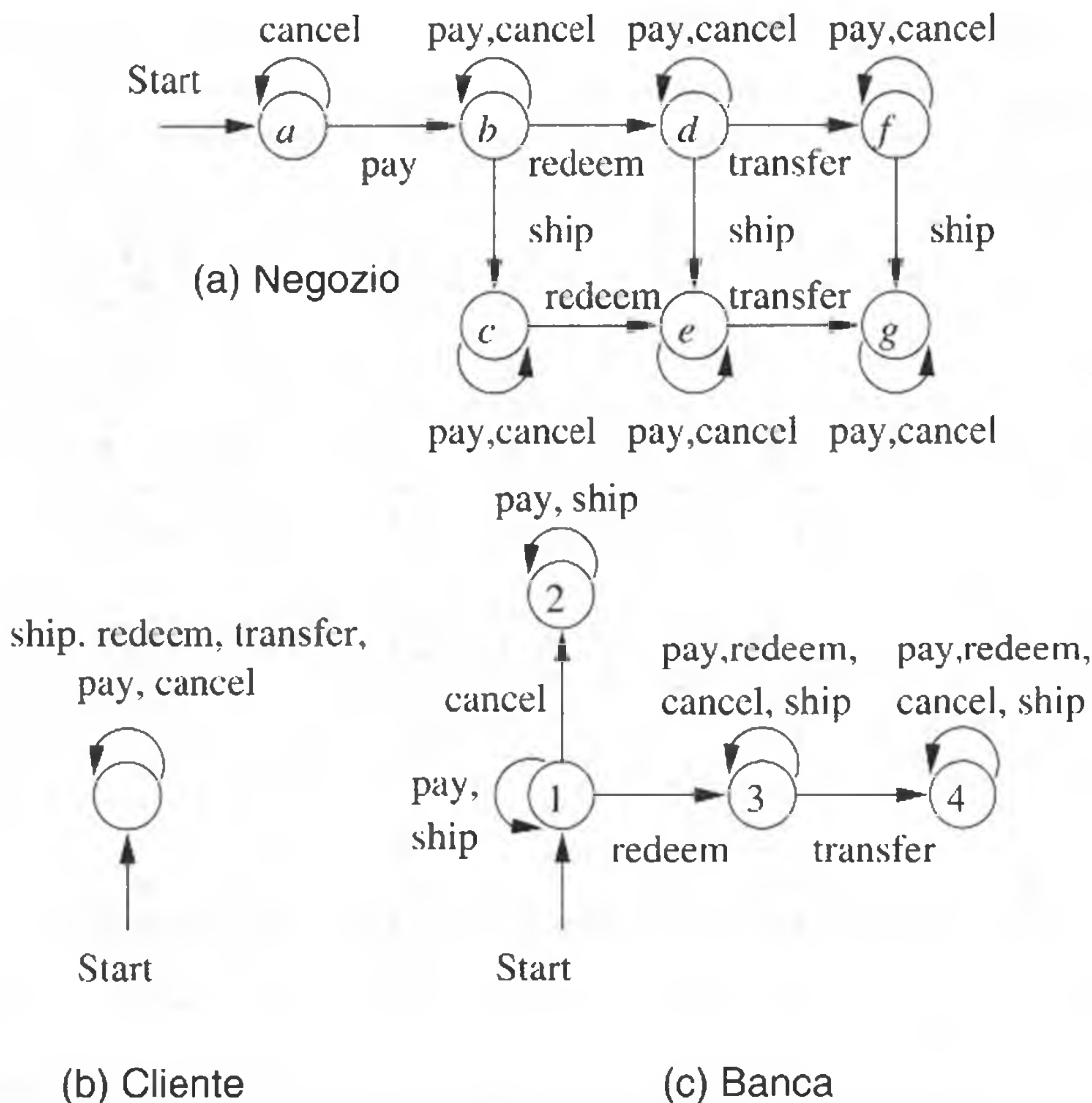


Figura 2.2 Gli insiemi completi di transizioni per i tre automi.

complesso, e non è immediatamente chiaro in quali combinazioni di stati possano trovarsi i loro automi.

Abitualmente si esplora l'interazione di automi come questi costruendo l'automa *prodotto*. Gli stati di tale automa rappresentano una coppia di stati, uno del negozio e uno della banca. Per esempio lo stato $(3, d)$ dell'automa prodotto rappresenta la situazione in cui la banca si trova nello stato 3 e il negozio nello stato d. Poiché la banca ha quattro stati e il negozio ne ha sette, l'automa prodotto ha $4 \times 7 = 28$ stati.

L'automa prodotto è illustrato nella Figura 2.3. Per chiarezza i 28 stati sono disposti in una griglia. La riga corrisponde allo stato della banca, la colonna a quello del negozio. Per risparmiare spazio abbiamo usato abbreviazioni per le etichette sugli archi: P, S, C, R e T stanno rispettivamente per *pay* (pagamento), *ship* (spedizione), *cancel* (cancellazione),

redeem (riscatto) e *transfer* (trasferimento).

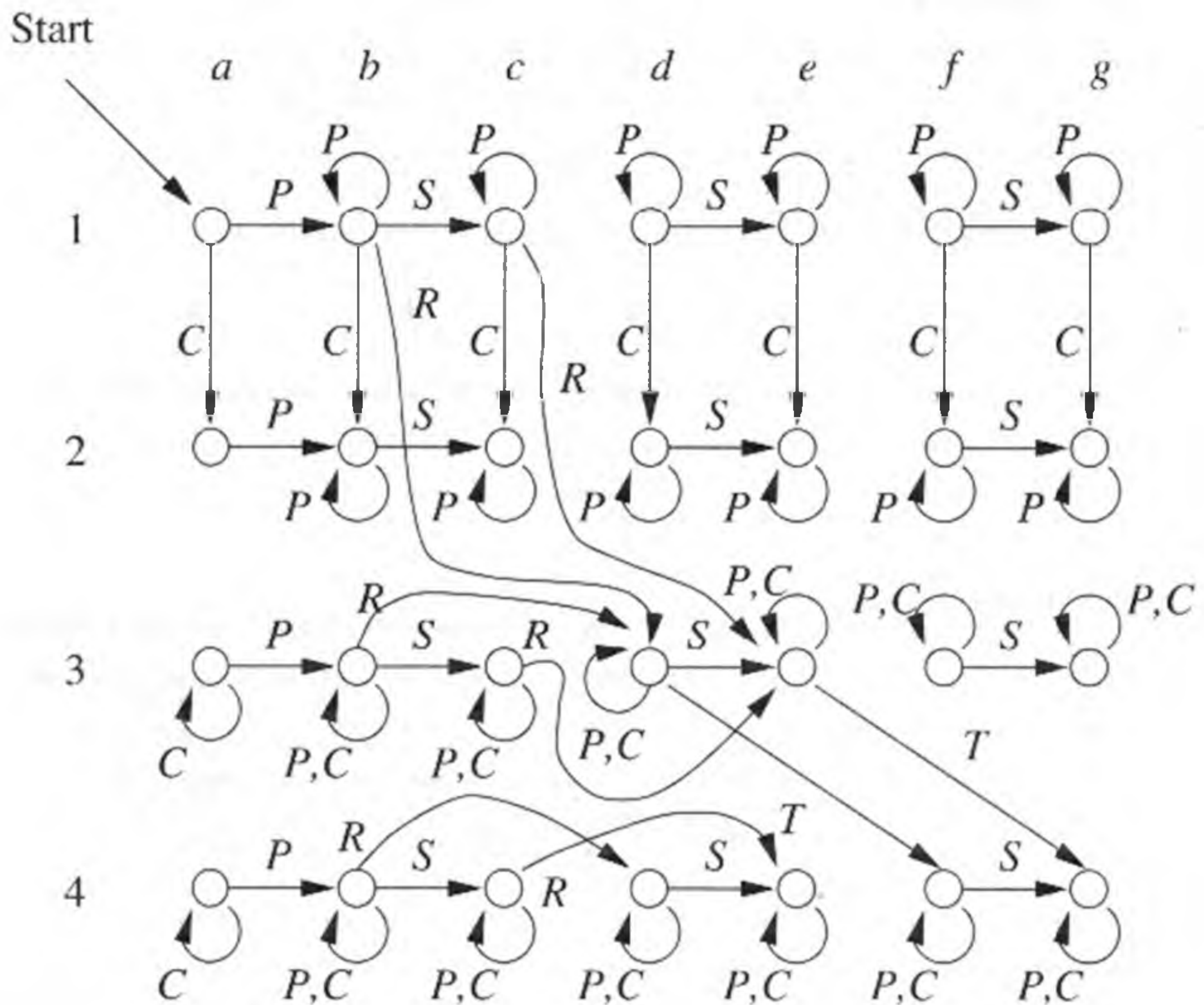


Figura 2.3 L'automa prodotto per il negozio e la banca.

Per costruire gli archi dell'automa prodotto bisogna far funzionare "in parallelo" l'automa della banca e quello del negozio. I due componenti dell'automa prodotto compiono transizioni sugli input autonomamente. Tuttavia è importante notare che se a fronte di un input uno dei due automi non ha uno stato a cui passare, anche l'automa prodotto non può compiere la transizione, e quindi "muore".

Per precisare la regola di transizione, supponiamo che l'automa prodotto si trovi nello stato (i, x). Questo stato corrisponde alla situazione in cui la banca si trova nello stato i e il negozio nello stato x. Sia Z una delle azioni di input. Nell'automa che rappresenta la banca cerchiamo una transizione a partire dallo stato i con etichetta Z. Immaginiamo che tale transizione ci sia e che conduca allo stato j (che potrebbe coincidere con i se la banca compie un ciclo sull'input Z). Nell'automa del negozio cerchiamo un arco etichettato Z che porti a uno stato y. Se esistono sia j sia y, l'automa prodotto ha un arco dallo stato (i, x) allo stato (j, y), etichettato Z. Se j oppure y non esiste (perché la banca o il negozio non hanno archi uscenti rispettivamente da i o da x per l'input Z), allora non esiste nessun

arco uscente da (i, x) etichettato Z.

Risulta chiaro come sono stati scelti gli archi nella Figura 2.3. Per esempio, a fronte dell'input *pay* il negozio passa dallo stato a allo stato b, ma resta fermo se si trova in qualunque altro stato che non sia a. Quando l'input è *pay* la banca permane in qualunque stato si trovi, perché dal suo punto di vista l'azione è irrilevante. Questa osservazione spiega i quattro archi etichettati P all'estrema sinistra delle quattro righe nella Figura 2.3, e i cicli etichettati P sugli altri stati.

Un altro esempio: consideriamo l'input *redeem*. Se la banca riceve un messaggio *redeem* quando si trova nello stato 1, passa nello stato 3. Se si trova nello stato 3 oppure 4, resta ferma, mentre se si trova nello stato 2 muore, cioè non ha nessuno stato dove andare. Il negozio, d'altro canto, sullo stesso input può fare transizioni dallo stato b allo stato d oppure da c a e. Nella Figura 2.3 vediamo sei archi etichettati *redeem*, che corrispondono alle sei combinazioni di tre stati della banca e due stati del negozio che hanno archi uscenti con l'etichetta R. Per esempio nello stato $(1, b)$ l'arco etichettato R porta l'automa allo stato $(3, d)$, perché *redeem* conduce la banca dallo stato 1 allo stato 3 e il negozio da b a d. Un altro esempio: esiste un arco etichettato R da $(4, c)$ a $(4, e)$, poiché *redeem* riporta la banca dallo stato 4 allo stato 4, mentre conduce il negozio dallo stato c allo stato e.

2.1.5 Validazione del protocollo mediante l'automa prodotto

La Figura 2.3 mostra alcune cose interessanti. Per esempio solo dieci dei 28 stati possono essere raggiunti a partire dallo stato iniziale, che è $(1, a)$, cioè la combinazione degli stati iniziali degli automi della banca e del negozio. Si noti che stati come $(2, e)$ e $(4, d)$ non sono *accessibili*, ossia non c'è un cammino che li raggiunga dallo stato iniziale. Non è necessario includere gli stati inaccessibili nell'automa; questi sono presenti nell'esempio solo per completezza.

Il vero scopo dell'analisi di un protocollo come questo per mezzo di automi è porre domande del tipo: "può verificarsi il seguente errore?", e dare una risposta. Nell'esempio possiamo chiederci se è possibile che il negozio inoltri la merce e non venga pagato. In altre parole: può l'automa prodotto entrare in uno stato in cui il negozio ha spedito la merce (ossia lo stato è nella colonna c, e o g), sebbene non sia stata fatta né mai si farà una transizione sull'input T?

Per esempio nello stato $(3, e)$ la merce è stata spedita, ma prima o poi ci sarà una transizione sull'input T verso lo stato $(4, g)$. Per quanto riguarda la banca, una volta raggiunto lo stato 3, essa ha ricevuto ed elaborato la richiesta *redeem*. Ciò significa che deve essersi trovata nello stato 1 prima di ricevere *redeem*, e perciò il messaggio *cancel* non era stato ricevuto e verrà ignorato se dovesse presentarsi in futuro. In questo modo la banca eseguirà il trasferimento di denaro al negozio.

Lo stato $(2, c)$, invece, costituisce un problema. Lo stato è accessibile, ma l'unico

arco uscente riconduce allo stesso stato. Tale stato corrisponde alla situazione in cui la banca ha ricevuto un messaggio *cancel* prima del messaggio *redeem*. Ciononostante il negozio ha ricevuto un messaggio *pay*, cioè il cliente ha fatto il doppio gioco e ha speso e annullato lo stesso denaro. Incautamente il negozio ha spedito la merce prima di tentare di riscattare il denaro, e quando eseguirà l'azione *redeem* la banca non riconoscerà il messaggio. Infatti essa si trova nello stato 2 in cui ha annullato il denaro e non è disposta a elaborare una richiesta *redeem*.

2.2 Automi a stati finiti deterministici

È arrivato il momento di presentare la nozione formale di automa a stati finiti, in modo da poter precisare alcune argomentazioni e descrizioni informali viste nei Paragrafi 1.1.1 e 2.1. Cominciamo dalla definizione di automa a stati finiti deterministico, un automa che dopo aver letto una qualunque sequenza di input si trova in un singolo stato. Il termine "deterministico" concerne il fatto che per ogni input esiste un solo stato verso il quale l'automa passa dal suo stato corrente. All'opposto, gli automi a stati finiti non deterministici, tema del Paragrafo 2.3, possono trovarsi in diversi stati contemporaneamente. Il termine "automa a stati finiti" farà riferimento alla varietà deterministica, anche se si farà uso di "deterministico" o dell'abbreviazione *DFA* (*Deterministic Finite Automaton*, automa a stati finiti deterministico) per ricordare al lettore di quale tipo di automa si sta parlando.

2.2.1 Definizione di automa a stati finiti deterministico

Un *automa a stati finiti deterministico* consiste dei seguenti componenti.

1. Un insieme finito di *stati*, spesso indicato con Q.

2. Un insieme finito di *simboli di input*, spesso indicato con Σ.

3. Una *funzione di transizione*, che prende come argomento uno stato e un simbolo di input e restituisce uno stato. La funzione di transizione sarà indicata comunemente con δ. Nella rappresentazione grafica informale di automi che abbiamo visto, δ è rappresentata dagli archi tra gli stati e dalle etichette sugli archi. Se q è uno stato e a è un simbolo di input, $\delta(q, a)$ è lo stato p tale che esiste un arco etichettato con a da q a p.[2]

4. Uno *stato iniziale*, uno degli stati in Q.

[2] Più precisamente, il grafo è la descrizione di una funzione di transizione δ, e gli archi del grafo sono costruiti per riflettere le transizioni specificate da δ.

5. Un insieme di stati *finali*, o *accettanti*, F. L'insieme F è un sottoinsieme di Q.

Un automa a stati finiti deterministico verrà spesso indicato con il suo acronimo *DFA*. La rappresentazione più concisa di un DFA è un'enumerazione dei suoi cinque componenti. Nelle dimostrazioni denotiamo un DFA come una quintupla:

$$A = (Q, \Sigma, \delta, q_0, F)$$

dove A è il nome del DFA, Q è l'insieme degli stati, Σ i suoi simboli di input, δ la sua funzione di transizione, q_0 il suo stato iniziale ed F il suo insieme di stati accettanti.

2.2.2 Elaborazione di stringhe in un DFA

La prima cosa che bisogna capire di un DFA è come decide se "accettare" o no una sequenza di simboli di input. Il "linguaggio" del DFA è l'insieme di tutte le stringhe che il DFA accetta. Supponiamo che $a_1a_2 \cdots a_n$ sia una sequenza di simboli di input. Si parte dal DFA nel suo stato iniziale, q_0. Consultando la funzione di transizione δ, per esempio $\delta(q_0, a_1) = q_1$, troviamo lo stato in cui il DFA entra dopo aver letto il primo simbolo di input, a_1. Il successivo simbolo di input, a_2, viene trattato valutando $\delta(q_1, a_2)$. Supponiamo che questo stato sia q_2. Continuiamo così, trovando gli stati $q_3, q_4, \ldots, q_n$ tali che $\delta(q_{i-1}, a_i) = q_i$ per ogni i. Se q_n è un elemento di F, allora l'input $a_1a_2 \cdots a_n$ viene accettato, altrimenti viene rifiutato.

Esempio 2.1 Specifichiamo formalmente un DFA che accetta tutte e sole le stringhe di 0 e di 1 in cui compare la sequenza 01. Possiamo scrivere il linguaggio L così:

$$\{w \mid w \text{ è della forma } x01y \text{ per stringhe } x \text{ e } y \text{ che consistono solamente di 0 e di 1}\}$$

Una descrizione equivalente, che usa i parametri x e y a sinistra della barra verticale, è:

$$\{x01y \mid x \text{ e } y \text{ sono stringhe qualsiasi di 0 e di 1}\}$$

Esempi di stringhe appartenenti al linguaggio includono 01, 11010 e 100011. Esempi di stringhe *non* appartenenti al linguaggio includono ϵ, 0, e 111000.

Che cosa sappiamo di un automa che accetta questo linguaggio L? In primo luogo il suo alfabeto di input è $\Sigma = \{0, 1\}$; ha un certo insieme di stati, Q, di cui uno, poniamo q_0, è lo stato iniziale. Quest'automa deve ricordare i fatti importanti relativi agli input già visti. Per decidere se 01 è una sottostringa dell'input, A deve ricordare quanto segue.

1. Ha già visto 01? In caso affermativo accetta ogni sequenza di ulteriori input, cioè da questo momento in poi si troverà solo in stati accettanti.

2. Pur non avendo ancora visto 01, l'input più recente è stato 0, cosicché se ora vede un 1 avrà visto 01. Da questo momento può accettare qualunque seguito?

3. Non ha ancora visto 01, ma l'input più recente è nullo (siamo ancora all'inizio) oppure come ultimo dato ha visto un 1? In tal caso A non accetta finché non vede uno 0 e subito dopo un 1.

Ognuna di queste tre condizioni può essere rappresentata da uno stato. La condizione (3) è rappresentata dallo stato iniziale q_0. All'inizio bisogna vedere uno 0 e poi un 1. Ma se nello stato q_0 si vede per primo un 1, allora non abbiamo fatto alcun passo verso 01, e dunque dobbiamo permanere nello stato q_0. In altri termini $\delta(q_0, 1) = q_0$.

D'altra parte, se ci troviamo nello stato q_0 e vediamo uno 0, siamo nella condizione (2). Vale a dire: non abbiamo ancora visto 01, ma ora abbiamo lo 0. Dunque usiamo q_2 per rappresentare la condizione (2). La transizione da q_0 sull'input 0 è $\delta(q_0, 0) = q_2$.

Ora consideriamo le transizioni dallo stato q_2. Se vediamo uno 0, non siamo in una situazione migliore di prima, ma neanche peggiore. Non abbiamo visto 01, ma 0 è stato l'ultimo simbolo incontrato: stiamo ancora aspettando un 1. Lo stato q_2 descrive questa situazione perfettamente, e dunque poniamo $\delta(q_2, 0) = q_2$. Se ci troviamo nello stato q_2 e vediamo un input 1, sappiamo che c'è uno 0 seguito da un 1. Possiamo passare a uno stato accettante, che si chiamerà q_1 e corrisponderà alla summenzionata condizione (1). Ossia $\delta(q_2, 1) = q_1$.

Infine dobbiamo determinare le transizioni per lo stato q_1. In questo stato abbiamo già incontrato una sequenza 01, quindi, qualsiasi cosa accada, saremo ancora in una situazione in cui abbiamo visto 01. In altri termini $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.

Da quanto detto risulta allora $Q = \{q_0, q_1, q_2\}$. Come affermato in precedenza, q_0 è lo stato iniziale e l'unico stato accettante è q_1. Quindi $F = \{q_1\}$. La definizione completa dell'automa A che accetta il linguaggio L delle stringhe che hanno una sottostringa 01 è

$$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

dove δ è la funzione di transizione descritta sopra. □

2.2.3 Notazioni più semplici per i DFA

Specificare un DFA attraverso una quintupla e una descrizione dettagliata della funzione di transizione δ è allo stesso tempo noioso e di difficile lettura. I sistemi preferibili per la descrizione degli automi sono due:

1. un *diagramma di transizione*, cioè un grafo come quelli presentati nel Paragrafo 2.1

2. una *tabella di transizione*, cioè la specificazione tabellare della funzione δ, da cui si deducono l'insieme degli stati e l'alfabeto di input.

Diagrammi di transizione

Un *diagramma di transizione* per un DFA $A = (Q, \Sigma, \delta, q_0, F)$ è un grafo definito come segue.

a) Per ogni stato in Q esiste un nodo.

b) Per ogni stato q in Q e ogni simbolo di input a in Σ, sia $\delta(q, a) = p$. Allora il diagramma ha un arco dal nodo q al nodo p etichettato a. Se esistono diversi simboli di input che causano transizioni da q a p, il diagramma può avere un arco etichettato dalla lista di tali simboli.

c) Una freccia etichettata *Start* entra nello stato iniziale q_0. Tale freccia non proviene da alcun nodo.

d) I nodi corrispondenti a stati accettanti (quelli in F) sono indicati da un doppio circolo. Gli stati non in F hanno un solo circolo.

Esempio 2.2 La Figura 2.4 mostra il diagramma di transizione per il DFA costruito nell'Esempio 2.1. Nel diagramma vediamo i tre nodi che corrispondono ai tre stati. C'è una freccia *Start* che entra nello stato iniziale q_0, e l'unico stato accettante, q_1, è rappresentato da un doppio circolo. Da ogni stato escono un arco etichettato 0 e un arco etichettato 1 (sebbene i due archi siano combinati in un unico arco con una doppia etichetta nel caso di q_1). Ogni arco corrisponde a uno dei fatti relativi a δ, stabiliti nell'Esempio 2.1. □

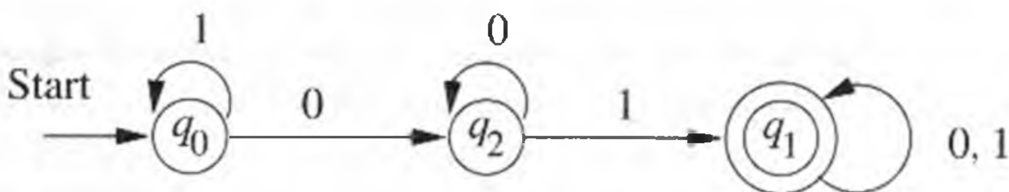


Figura 2.4 Il diagramma di transizione per il DFA che accetta tutte le stringhe con una sottostringa 01.

Tabelle di transizione

Una *tabella di transizione* è una comune rappresentazione tabellare di una funzione come δ, che ha due argomenti e restituisce un valore. Le righe della tabella corrispondono agli stati, le colonne agli input. La voce all'incrocio della riga corrispondente allo stato q e della colonna corrispondente all'input a è lo stato $\delta(q, a)$.

Esempio 2.3 La tabella di transizione relativa alla funzione δ dell'Esempio 2.1 è rappresentata dalla Figura 2.5, corredata di due informazioni specifiche: lo stato iniziale è indicato da una freccia, gli stati accettanti da un asterisco. Dato che possiamo dedurre gli insiemi degli stati e dei simboli di input dalle intestazioni delle righe e delle colonne, dalla tabella di transizione si ricavano tutte le informazioni necessarie per specificare l'automa a stati finiti in maniera univoca. □

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Figura 2.5 Tabella di transizione per il DFA dell'Esempio 2.1.

2.2.4 Estensione della funzione di transizione alle stringhe

Abbiamo spiegato in modo informale che un DFA definisce un linguaggio: l'insieme di tutte le stringhe che producono una sequenza di transizioni di stati dallo stato iniziale a uno stato accettante. Rispetto al diagramma di transizione, il linguaggio di un DFA è l'insieme delle etichette lungo i cammini che conducono dallo stato iniziale a un qualunque stato accettante.

È necessario ora precisare la nozione di linguaggio di un DFA. A questo scopo definiamo una *funzione di transizione estesa*, che descrive che cosa succede quando partiamo da uno stato e seguiamo una sequenza di input. Se δ è la funzione di transizione, allora la funzione di transizione estesa costruita da δ si chiamerà $\hat{\delta}$. La funzione di transizione estesa è una funzione che prende uno stato q e una stringa w e restituisce uno stato p: lo stato che l'automa raggiunge quando parte nello stato q ed elabora la sequenza di input w. Definiamo $\hat{\delta}$ per induzione sulla lunghezza della stringa di input come segue.

BASE $\hat{\delta}(q, \epsilon) = q$. In altre parole, se ci troviamo nello stato q e non leggiamo alcun input, allora rimaniamo nello stato q.

INDUZIONE Supponiamo che w sia una stringa della forma xa, ossia a è l'ultimo simbolo di w e x è la stringa che consiste di tutti i simboli eccetto l'ultimo.[3] Per esempio $w = 1101$ si scompone in $x = 110$ e $a = 1$. Allora

[3] Ricordiamo la convenzione che abbiamo fissato, secondo la quale le lettere all'inizio dell'alfabeto sono simboli, mentre quelle verso la fine dell'alfabeto sono stringhe. Tale convenzione è necessaria affinché la proposizione "della forma xa" abbia un senso.

$$\hat{\delta}(q, w) = \delta\big(\hat{\delta}(q, x), a\big) \tag{2.1}$$

La (2.1) può sembrare contorta, ma il concetto è semplice. Per computare $\hat{\delta}(q, w)$, calcoliamo prima $\hat{\delta}(q, x)$, lo stato in cui si trova l'automa dopo aver elaborato tutti i simboli di w eccetto l'ultimo. Supponiamo che questo stato sia p, ossia $\hat{\delta}(q, x) = p$. Allora $\hat{\delta}(q, w)$ è quanto si ottiene compiendo una transizione dallo stato p sull'input a, l'ultimo simbolo di w. In altri termini $\hat{\delta}(q, w) = \delta(p, a)$.

Esempio 2.4 Costruiamo un DFA che accetti il linguaggio

$$L = \{w \mid w \text{ ha un numero pari di 0 e un numero pari di 1}\}$$

Non dovrebbe sorprendere che il compito degli stati di questo DFA sia quello di contare il numero degli 0 e quello degli 1, ma di contarli modulo 2. In altre parole si usa lo stato per ricordare se il numero degli 0 e il numero degli 1 visti fino a quel momento sono pari o dispari. Quindi ci sono quattro stati, che possono essere interpretati come segue:

q_0: sia il numero degli 0 sia il numero degli 1 visti finora sono pari

q_1: il numero degli 0 visti finora è pari, ma il numero degli 1 è dispari

q_2: il numero degli 1 visti finora è pari, ma il numero degli 0 è dispari

q_3: sia il numero degli 0 sia il numero degli 1 visti finora sono dispari.

Lo stato q_0 è nello stesso tempo lo stato iniziale e l'unico stato accettante. È lo stato iniziale perché prima di leggere qualunque input il numero degli 0 e degli 1 visti sono entrambi zero, e zero è pari. È l'unico stato accettante perché descrive esattamente la condizione per la quale una sequenza di 0 e di 1 appartiene al linguaggio L.

Siamo ora in grado di specificare un DFA per il linguaggio L:

$$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

La funzione di transizione δ è descritta dal diagramma di transizione della Figura 2.6. Si noti che ogni input 0 fa sì che lo stato attraversi la linea tratteggiata orizzontale. Pertanto, dopo aver visto un numero pari di 0 ci troviamo al di sopra della linea, nello stato q_0 oppure q_1, mentre dopo averne visto un numero dispari ci troviamo al di sotto, nello stato q_2 oppure q_3. Analogamente, ogni 1 fa sì che lo stato attraversi la linea tratteggiata verticale. Perciò, dopo aver visto un numero pari di 1 siamo a sinistra, nello stato q_0 oppure q_2, mentre dopo averne visto un numero dispari siamo a destra, nello stato q_1 o q_3. Tali osservazioni sono una dimostrazione informale che i quattro stati hanno le

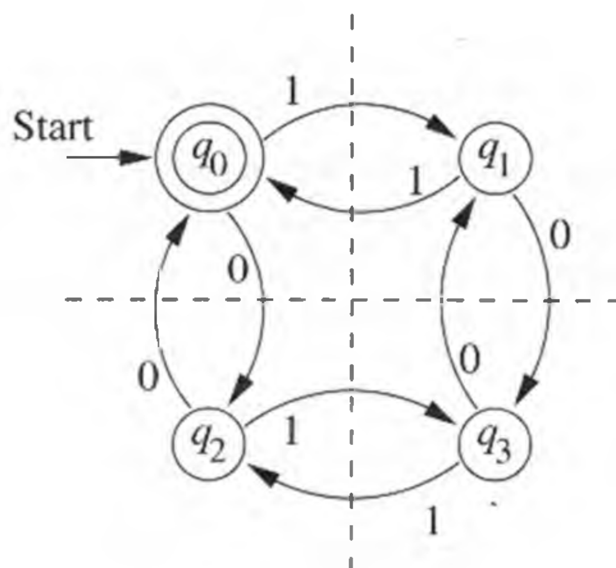


Figura 2.6 Diagramma di transizione per il DFA dell'Esempio 2.4.

interpretazioni a loro attribuite. Si potrebbe dimostrare in termini formali la correttezza delle nostre affermazioni sugli stati, per mutua induzione, sulla scorta dell'Esempio 1.23.

È possibile rappresentare questo DFA anche per mezzo di una tabella di transizione, come illustrato nella Figura 2.7. Noi però non siamo interessati solo all'ideazione di questo DFA; il nostro intento è di usarlo per illustrare la costruzione di $\hat{\delta}$ dalla funzione di transizione δ. Supponiamo che l'input sia 110101. Dato che questa stringa ha un numero pari sia di 0 sia di 1, ci aspettiamo che appartenga al linguaggio, cioè che $\hat{\delta}(q_0, 110101) = q_0$, dato che q_0 è l'unico stato accettante. Verifichiamo quest'asserzione.

	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figura 2.7 Tabella di transizione per il DFA dell'Esempio 2.4.

La verifica comporta il calcolo di $\hat{\delta}(q_0, w)$ per ogni prefisso w di 110101, a partire da ϵ e procedendo per aggiunte successive. Il riepilogo di questo calcolo è:

- $\hat{\delta}(q_0, \epsilon) = q_0$
- $\hat{\delta}(q_0, 1) = \delta\big(\hat{\delta}(q_0, \epsilon), 1\big) = \delta(q_0, 1) = q_1$

Notazione standard e variabili locali

Dopo aver letto questo paragrafo si potrebbe immaginare che la notazione abituale sia obbligatoria, cioè che si *debba* usare δ per la funzione di transizione, A per il nome di un DFA, e così via. In realtà tendiamo a usare le stesse variabili per denotare la stessa cosa in tutti gli esempi perché ciò aiuta a ricordare il tipo delle variabili; analogamente in un programma una variabile i è quasi sempre di tipo intero. Di fatto possiamo chiamare i componenti di un automa, o qualunque altro elemento, come vogliamo. Se lo desideriamo, possiamo chiamare un DFA M e la sua funzione di transizione T.

Non ci si deve inoltre stupire del fatto che le stesse variabili hanno significati diversi in contesti diversi. Per esempio a entrambi i DFA degli Esempi 2.1 e 2.4 è stata assegnata una funzione di transizione chiamata δ, ma ognuna delle due funzioni è una variabile locale, pertinente solo al suo esempio. Le due funzioni sono molto diverse e non hanno alcuna relazione l'una con l'altra.

- $\hat{\delta}(q_0, 11) = \delta\big(\hat{\delta}(q_0, 1), 1\big) = \delta(q_1, 1) = q_0$
- $\hat{\delta}(q_0, 110) = \delta\big(\hat{\delta}(q_0, 11), 0\big) = \delta(q_0, 0) = q_2$
- $\hat{\delta}(q_0, 1101) = \delta\big(\hat{\delta}(q_0, 110), 1\big) = \delta(q_2, 1) = q_3$
- $\hat{\delta}(q_0, 11010) = \delta\big(\hat{\delta}(q_0, 1101), 0\big) = \delta(q_3, 0) = q_1$
- $\hat{\delta}(q_0, 110101) = \delta\big(\hat{\delta}(q_0, 11010), 1\big) = \delta(q_1, 1) = q_0$.

□

2.2.5 Il linguaggio di un DFA

Possiamo ora definire il *linguaggio* di un DFA $A = (Q, \Sigma, \delta, q_0, F)$. Questo linguaggio è indicato con $L(A)$ ed è definito da

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ è in } F\}$$

In altre parole il linguaggio di A è l'insieme delle stringhe w che portano dallo stato iniziale q_0 a uno degli stati accettanti. Se L è uguale a $L(A)$ per un DFA A, allora diciamo che L è un *linguaggio regolare*.

Esempio 2.5 Come detto in precedenza, se A è il DFA dell'Esempio 2.1, allora $L(A)$ è l'insieme di tutte le stringhe di 0 e di 1 che contengono la sottostringa 01. Se invece A è il DFA dell'Esempio 2.4, allora $L(A)$ è l'insieme di tutte le stringhe di 0 e di 1 i cui numeri di 0 e di 1 sono entrambi pari. □

2.2.6 Esercizi

Esercizio 2.2.1 La Figura 2.8 rappresenta una pista per biglie. Una biglia viene lanciata a partire da A o da B. Le leve x_1, x_2 e x_3 fanno cadere la biglia a sinistra oppure a destra. Ogni volta che una biglia si imbatte in una leva, dopo che la biglia è passata la leva si ribalta, cosicché la biglia successiva prende l'altra diramazione.

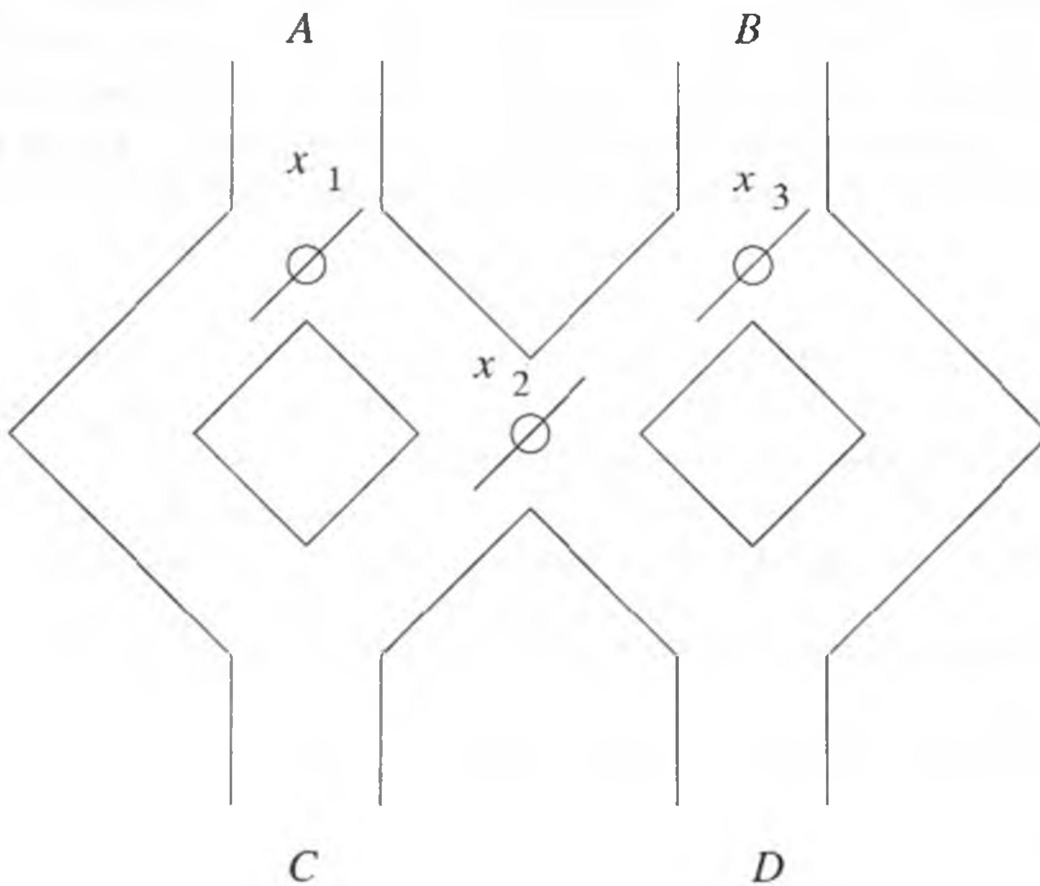


Figura 2.8 Una pista per biglie.

* a) Modellate questa pista con un automa a stati finiti. Gli input A e B rappresentano la via su cui la biglia viene fatta cadere. Supponete che l'accettazione corrisponda all'uscita della biglia da D, la non accettazione all'uscita da C.

! b) Descrivete informalmente il linguaggio dell'automa.

c) Supponete che le leve si ribaltino *prima* di permettere il passaggio della biglia. Come cambiano le risposte a (a) e (b)?

*! **Esercizio 2.2.2** Abbiamo definito $\hat{\delta}$ scomponendo la stringa di input in una stringa seguita da un singolo simbolo (nella parte induttiva, Equazione 2.1). Informalmente possiamo pensare invece che $\hat{\delta}$ descriva ciò che capita lungo un cammino con una certa stringa di etichette; in tal caso non dovrebbe essere rilevante in che modo scomponiamo la stringa di input nella definizione di $\hat{\delta}$. Mostrate che in effetti $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ per qualunque stato q e qualunque coppia di stringhe x e y. *Suggerimento*: svolgete un'induzione su $|y|$.

! **Esercizio 2.2.3** Dimostrate che, per qualunque stato q, stringa x e simbolo di input a, $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$. *Suggerimento*: usate l'Esercizio 2.2.2.

Esercizio 2.2.4 Definite tre DFA che accettino i seguenti linguaggi sull'alfabeto $\{0, 1\}$:

* a) l'insieme di tutte le stringhe che finiscono per 00

b) l'insieme di tutte le stringhe con tre 0 consecutivi (non necessariamente alla fine)

c) l'insieme delle stringhe con 011 come sottostringa.

! **Esercizio 2.2.5** Definite quattro DFA che accettino i seguenti linguaggi sull'alfabeto $\{0, 1\}$:

a) l'insieme di tutte le stringhe tali che ogni blocco di cinque simboli consecutivi contenga almeno due 0

b) l'insieme di tutte le stringhe il cui decimo simbolo a partire da destra sia 1

c) l'insieme delle stringhe che cominciano o finiscono (o entrambe le cose) per 01

d) l'insieme delle stringhe tali che il numero di 0 sia divisibile per cinque e il numero di 1 sia divisibile per 3.

!! **Esercizio 2.2.6** Definite due DFA che accettino i seguenti linguaggi sull'alfabeto $\{0, 1\}$:

* a) l'insieme di tutte le stringhe che cominciano con un 1 e che, interpretate come interi binari, siano multipli di 5 (per esempio le stringhe 101, 1010 e 1111 sono nel linguaggio; 0, 100 e 111 invece no)

b) l'insieme di tutte le stringhe che lette al contrario come un intero binario siano divisibili per 5 (esempi di stringhe nel linguaggio sono 0, 10011, 1001100 e 0101).

Esercizio 2.2.7 Sia A un DFA e q uno stato di A tale che $\delta(q, a) = q$ per tutti i simboli di input a. Dimostrate per induzione sulla lunghezza dell'input che, per tutte le stringhe di input w, $\hat{\delta}(q, w) = q$.

Esercizio 2.2.8 Sia A un DFA e a un simbolo di input di A tale che, per tutti gli stati q di A, valga $\delta(q, a) = q$.

a) Dimostrate per induzione su n che, per ogni $n \geq 0$, $\hat{\delta}(q, a^n) = q$, dove a^n è la stringa formata da n ripetizioni di a.

b) Dimostrate che o $\{a\}^* \subseteq L(A)$ o $\{a\}^* \cap L(A) = \emptyset$.

*! **Esercizio 2.2.9** Sia $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ un DFA e supponiamo che per tutti gli a in Σ si abbia $\delta(q_0, a) = \delta(q_f, a)$.

a) Dimostrate che, per ogni $w \neq \epsilon$, $\hat{\delta}(q_0, w) = \hat{\delta}(q_f, w)$.

b) Dimostrate che se x è una stringa non vuota in $L(A)$, allora per ogni $k > 0$ anche x^k (cioè x scritta un numero k di volte) è in $L(A)$.

*! **Esercizio 2.2.10** Considerate il DFA con la seguente tabella di transizione:

	0	1
$\rightarrow A$	A	B
$*B$	B	A

Descrivete in termini informali il linguaggio accettato da questo DFA e dimostrate per induzione sulla lunghezza della stringa di input che la descrizione è corretta. *Suggerimento*: nel porre l'ipotesi induttiva è consigliabile formulare un enunciato su quali input portano a ciascuno stato, e non solo su quali input portano allo stato accettante.

! **Esercizio 2.2.11** Ripetete l'Esercizio 2.2.10 per la seguente tabella di transizione:

	0	1
$\rightarrow *A$	B	A
$*B$	C	A
C	C	C

2.3 Automi a stati finiti non deterministici

Un automa a stati finiti non deterministico (*NFA, Nondeterministic Finite Automaton*) può trovarsi contemporaneamente in diversi stati. Questa caratteristica viene sovente espressa come capacità di "scommettere" su certe proprietà dell'input. Per esempio, quando un automa viene usato per cercare determinate sequenze di caratteri (come: parole chiave) in una lunga porzione di testo, è utile "scommettere" che ci si trova all'inizio di una di tali

sequenze e usare una sequenza di stati per verificare, carattere per carattere, che compaia la stringa cercata. Vedremo un esempio di questo tipo di applicazione nel Paragrafo 2.4.

Prima di esaminare le applicazioni, è necessario definire gli automi a stati finiti non deterministici e mostrare che accettano gli stessi linguaggi accettati dai DFA. Come i DFA, gli NFA accettano proprio i linguaggi regolari. Tuttavia ci sono buone ragioni per occuparsi degli NFA. Spesso sono più succinti e più facili da definire rispetto ai DFA; inoltre, anche se è sempre possibile convertire un NFA in un DFA, quest'ultimo può avere esponenzialmente più stati di un NFA. Per fortuna casi di questo tipo sono rari.

2.3.1 Descrizione informale degli automi a stati finiti non deterministici

Come un DFA, un NFA ha un insieme finito di stati, un insieme finito di simboli di input, uno stato iniziale e un insieme di stati accettanti. Ha anche una funzione di transizione che chiameremo δ. La differenza tra DFA ed NFA sta nel tipo di δ. Per gli NFA, δ è una funzione che ha come argomenti uno stato e un simbolo di input (come quella dei DFA), ma restituisce un insieme di zero o più stati (invece di un solo stato, come nel caso dei DFA). Cominceremo da un esempio di NFA e poi passeremo a precisare le definizioni.

Esempio 2.6 La Figura 2.9 mostra un automa a stati finiti non deterministico con il compito di accettare tutte e sole le stringhe di 0 e di 1 che finiscono per 01. Lo stato q_0 è lo stato iniziale e possiamo pensare che l'automa si trovi nello stato q_0 (eventualmente insieme ad altri stati) quando non ha ancora "scommesso" che il finale 01 è cominciato. È sempre possibile che il simbolo successivo non sia il primo del suffisso 01, anche se quel simbolo è proprio 0. Dunque lo stato q_0 può operare una transizione verso se stesso sia su 0 sia su 1.

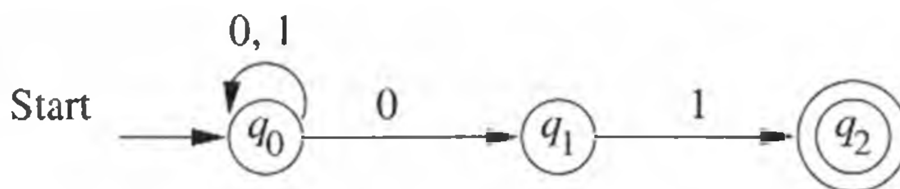


Figura 2.9 Un NFA che accetta tutte le stringhe che finiscono per 01.

Se però il simbolo successivo è 0, l'NFA scommette anche che è iniziato il suffisso 01. Un altro arco etichettato 0 conduce dunque da q_0 allo stato q_1. Si noti che ci sono due archi etichettati 0 in uscita da q_0. L'NFA ha l'opzione di andare verso q_0 oppure verso q_1, ed effettivamente fa entrambe le cose, come si vedrà quando si preciseranno le definizioni. Nello stato q_1 l'NFA verifica che il simbolo successivo sia 1, e se è così, passa allo stato q_2 e accetta.

Si osservi che non ci sono archi uscenti da q_1 etichettati 0 e non ci sono affatto archi uscenti da q_2. In tali situazioni il processo attivo dell'NFA corrispondente a quegli stati semplicemente "muore", sebbene altri processi possano continuare a esistere. Mentre un DFA ha soltanto un unico arco uscente da ogni stato per ogni simbolo di input, un NFA non ha questi vincoli; nella Figura 2.9, per esempio, abbiamo visto casi in cui il numero di archi è zero, uno e due.

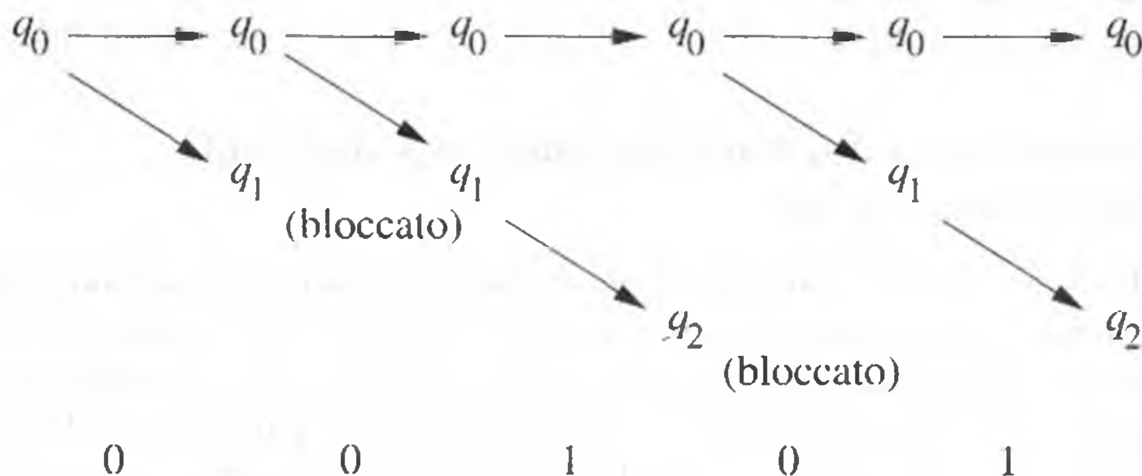


Figura 2.10 Gli stati in cui si trova un NFA mentre viene elaborata la sequenza di input 00101.

La Figura 2.10 illustra come un NFA elabora gli input. Abbiamo mostrato che cosa succede quando l'automa della Figura 2.9 riceve la sequenza di input 00101. Esso parte dal solo stato iniziale, q_0. Quando viene letto il primo 0, l'NFA può passare allo stato q_0 e allo stato q_1. In questo modo fa entrambe le cose. I due processi sono raffigurati nella seconda colonna della Figura 2.10.

A questo punto viene letto il secondo 0. Lo stato q_0 può nuovamente andare sia verso q_0 sia verso q_1. Lo stato q_1, invece, non ha transizioni su 0, perciò "muore". Quando si presenta il terzo input, un 1, bisogna considerare le transizioni sia da q_0 sia da q_1. Troviamo che q_0 va solamente verso q_0 su 1, mentre q_1 va solamente verso q_2. Dunque, dopo aver letto 001, l'NFA si trova negli stati q_0 e q_2. Poiché q_2 è uno stato accettante, l'NFA accetta 001.

Ma l'input non è finito. Il quarto simbolo, uno 0, fa morire il processo associato a q_2, mentre q_0 va sia in q_0 sia in q_1. L'ultimo input, un 1, manda q_0 in q_0 e q_1 in q_2. Dato che ci troviamo di nuovo in uno stato accettante, 00101 viene accettato. □

2.3.2 Definizione di automa a stati finiti non deterministico

Presentiamo ora le nozioni formali relative agli automi a stati finiti non deterministici. Verranno sottolineate le differenze tra i DFA e gli NFA. Un NFA si rappresenta

essenzialmente come un DFA:

$$A = (Q, \Sigma, \delta, q_0, F)$$

dove:

1. Q è un insieme finito di *stati*

2. Σ è un insieme finito di *simboli di input*

3. q_0, elemento di Q, è lo *stato iniziale*

4. F, un sottoinsieme di Q, è l'insieme degli stati *finali* (o *accettanti*)

5. δ, la *funzione di transizione*, è la funzione che ha come argomenti uno stato in Q e un simbolo di input in Σ, e restituisce un sottoinsieme di Q. Si noti che l'unica differenza tra un NFA e un DFA è nel tipo di valore restituito da δ: un insieme di stati nel caso di un NFA e un singolo stato nel caso di un DFA.

Esempio 2.7 L'NFA della Figura 2.9 può essere specificato formalmente come

$$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

dove la funzione di transizione δ è data dalla tabella di transizione della Figura 2.11. □

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Figura 2.11 Tabella di transizione di un NFA che accetta tutte le stringhe che finiscono per 01.

Si osservi che la tabella di transizione può specificare la funzione di transizione tanto per un NFA quanto per un DFA. L'unica differenza è che ogni voce nella tabella di un NFA è un insieme, anche se l'insieme è un *singoletto*, ossia ha un solo membro. Inoltre, se non c'è alcuna transizione da uno stato su un dato simbolo di input, la voce adeguata è $\emptyset$, l'insieme vuoto.

2.3.3 La funzione di transizione estesa

Come per i DFA, bisogna estendere la funzione di transizione δ di un NFA a una funzione $\hat{\delta}$ che prende uno stato q e una stringa di simboli di input w, e restituisce l'insieme degli stati in cui si trova l'NFA quando parte dallo stato q ed elabora la stringa w. L'idea è suggerita nella Figura 2.10; in sostanza, se q è l'unico stato nella prima colonna, $\hat{\delta}(q, w)$ è la colonna di stati successivi alla lettura di w. Per esempio la Figura 2.10 indica che $\hat{\delta}(q_0, 001) = \{q_0, q_2\}$. In termini formali definiamo $\hat{\delta}$ per la funzione di transizione δ di un NFA come segue.

BASE $\hat{\delta}(q, \epsilon) = \{q\}$. Se nessun simbolo di input è stato letto, ci troviamo nel solo stato da cui siamo partiti.

INDUZIONE Supponiamo che w sia della forma $w = xa$, dove a è il simbolo finale di w e x è la parte restante. Supponiamo altresì che $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$. Sia

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

Allora $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, r_m\}$. In termini meno formali computiamo $\hat{\delta}(q, w)$ calcolando inizialmente $\hat{\delta}(q, x)$, e poi seguendo le transizioni etichettate a da tutti questi stati.

Esempio 2.8 Usiamo $\hat{\delta}$ per descrivere come l'NFA della Figura 2.9 elabora l'input 00101. Un compendio dei passi è:

1. $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

Il punto (1) è la regola di base. Il punto (2) si ottiene applicando δ all'unico stato, q_0, che si trova nell'insieme precedente: il risultato è $\{q_0, q_1\}$. Il punto (3) si ottiene prendendo l'unione, sui due stati dell'insieme precedente, di ciò che si ottiene quando si applica loro δ con input 0. In altre parole $\delta(q_0, 0) = \{q_0, q_1\}$, mentre $\delta(q_1, 0) = \emptyset$. Per il punto (4) prendiamo l'unione di $\delta(q_0, 1) = \{q_0\}$ e $\delta(q_1, 1) = \{q_2\}$. I punti (5) e (6) sono simili a (3) e (4). □

2.3.4 Il linguaggio di un NFA

Come abbiamo suggerito, un NFA accetta una stringa w se, mentre si leggono i caratteri di w, è possibile fare una sequenza di scelte dello stato successivo che porta dallo stato iniziale a uno stato accettante. Il fatto che altre scelte per i simboli di input di w conducano a uno stato non accettante, oppure non conducano ad alcuno stato (cioè che una sequenza di stati muoia), non impedisce a w di venire accettato dall'NFA nel suo insieme. Formalmente, se $A = (Q, \Sigma, \delta, q_0, F)$ è un NFA, allora

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

In altre parole $L(A)$ è l'insieme delle stringhe w in Σ^* tali che $\hat{\delta}(q_0, w)$ contenga almeno uno stato accettante.

Esempio 2.9 Per esemplificare, proviamo in termini formali che l'NFA della Figura 2.9 accetta il linguaggio $L = \{w \mid w \text{ termina per } 01\}$. La dimostrazione è un'induzione mutua dei seguenti tre enunciati, che caratterizzano i tre stati:

1. $\hat{\delta}(q_0, w)$ contiene q_0 per ogni w

2. $\hat{\delta}(q_0, w)$ contiene q_1 se e solo se w finisce per 0

3. $\hat{\delta}(q_0, w)$ contiene q_2 se e solo se w finisce per 01.

Per dimostrare questi enunciati, bisogna considerare in che modo A può raggiungere ciascuno stato; ossia: qual è stato l'ultimo simbolo di input, e in quale stato si trovava A prima di leggere quel simbolo?

Dato che il linguaggio di quest'automa è l'insieme delle stringhe w tali che $\hat{\delta}(q_0, w)$ contiene q_2 (perché q_2 è l'unico stato accettante), la dimostrazione dei tre enunciati, in particolare di (3), garantisce che il linguaggio dell'NFA è l'insieme delle stringhe che finiscono per 01. La dimostrazione del teorema è un'induzione su $|w|$, la lunghezza di w, a partire dalla lunghezza 0.

BASE Se $|w| = 0$, allora $w = \epsilon$. L'enunciato (1) dice che $\hat{\delta}(q_0, \epsilon)$ contiene q_0; ciò segue dalla base della definizione di $\hat{\delta}$. Circa l'enunciato (2) sappiamo che ϵ non finisce per 0 e sappiamo inoltre che $\hat{\delta}(q_0, \epsilon)$ non contiene q_1, ancora una volta per la base della definizione di $\hat{\delta}$. Dunque le ipotesi di entrambe le direzioni dell'enunciato se-e-solo-se sono false, e di conseguenza entrambe le direzioni dell'enunciato sono vere. La dimostrazione dell'enunciato (3) per $w = \epsilon$ è essenzialmente la stessa della dimostrazione dell'enunciato (2) presentata sopra.

INDUZIONE Assumiamo che $w = xa$, dove a è un simbolo, 0 oppure 1. Possiamo supporre che gli enunciati dall'(1) al (3) siano validi per x e dobbiamo dimostrarli per w. Vale a dire, assumiamo $|w| = n + 1$, dunque $|x| = n$. Assumiamo l'ipotesi induttiva per n e dimostriamola per $n + 1$.

1. Sappiamo che $\hat{\delta}(q_0, x)$ contiene q_0. Poiché esistono transizioni sia su 0 sia su 1 da q_0 verso se stesso, ne consegue che anche $\hat{\delta}(q_0, w)$ contiene q_0; dunque l'enunciato (1) è dimostrato per w.

2. (Se) Assumiamo che w finisca per 0, ossia $a = 0$. Per l'enunciato (1) applicato a x, sappiamo che $\hat{\delta}(q_0, x)$ contiene q_0. Poiché esiste una transizione da q_0 a q_1 su input 0, concludiamo che $\hat{\delta}(q_0, w)$ contiene q_1.

   (Solo-se) Supponiamo che $\hat{\delta}(q_0, w)$ contenga q_1. Dal diagramma della Figura 2.9 si ricava che c'è un solo modo di raggiungere lo stato q_1: che la sequenza di input w sia della forma $x0$. Ciò è sufficiente per dimostrare la parte solo-se dell'enunciato (2).

3. (Se) Assumiamo che w finisca per 01. Allora, se $w = xa$, sappiamo che $a = 1$ e x finisce per 0. Per l'enunciato (2) applicato a x, sappiamo che $\hat{\delta}(q_0, x)$ contiene q_1. Poiché esiste una transizione da q_1 a q_2 su input 1, concludiamo che $\hat{\delta}(q_0, w)$ contiene q_2.

   (Solo-se) Supponiamo che $\hat{\delta}(q_0, w)$ contenga q_2. Dal diagramma della Figura 2.9 si deduce che c'è un solo modo di giungere nello stato q_2: che w sia della forma $x1$, dove $\hat{\delta}(q_0, x)$ contiene q_1. Per l'enunciato (2) applicato a x, sappiamo che x finisce per 0. Dunque w finisce per 01, e abbiamo dimostrato l'enunciato (3).

□

2.3.5 Equivalenza di automi a stati finiti deterministici e non deterministici

Ci sono molti linguaggi per i quali è più facile costruire un NFA anziché un DFA, come il linguaggio (Esempio 2.6) delle stringhe che finiscono per 01; eppure, per quanto sorprendente, ogni linguaggio che può essere descritto da un NFA può essere descritto anche da un DFA. Inoltre il DFA ha in pratica circa tanti stati quanti l'NFA, sebbene abbia spesso più transizioni. Nel peggiore dei casi, tuttavia, il più piccolo DFA può avere 2^n stati, mentre il più piccolo NFA per lo stesso linguaggio ha solo n stati.

La dimostrazione che i DFA possono fare le stesse cose degli NFA si serve di un'importante costruzione, detta "costruzione per sottoinsiemi", in quanto comporta la costruzione di tutti i sottoinsiemi dell'insieme di stati dell'NFA. In generale molte dimostrazioni

sugli automi richiedono la costruzione di un automa a partire da un altro. È importante considerare la costruzione per sottoinsiemi come esempio di descrizione formale di un automa nei termini degli stati e delle transizioni di un altro, senza conoscere i particolari del secondo automa.

La costruzione per sottoinsiemi parte da un NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Il suo fine è la descrizione di un DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tale che $L(D) = L(N)$. Si noti che gli alfabeti di input dei due automi sono gli stessi e lo stato iniziale di D è l'insieme che contiene solo lo stato iniziale di N. Gli altri componenti di D sono costruiti come segue.

- Q_D è l'insieme dei sottoinsiemi di Q_N; vale a dire, Q_D è l'*insieme potenza* di Q_N. Osserviamo che se Q_N ha n stati, allora Q_D avrà 2^n stati. Spesso dallo stato iniziale di Q_D non è possibile raggiungere tutti questi stati. Gli stati inaccessibili possono essere eliminati. Dunque, in effetti, il numero di stati di D può essere molto inferiore a 2^n.

- F_D è l'insieme dei sottoinsiemi S di Q_N tali che $S \cap F_N \neq \emptyset$. In altre parole F_D è formato dagli insiemi di stati di N che includono almeno uno stato accettante.

- Per ogni insieme $S \subseteq Q_N$ e per ogni simbolo di input a in Σ,

$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

  Cioè, per computare $\delta_D(S, a)$ consideriamo tutti gli stati p in S, rileviamo in quali insiemi di stati l'automa N va a finire partendo da p e leggendo a, e prendiamo infine l'unione di tutti questi insiemi.

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\to \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Figura 2.12 La costruzione completa per sottoinsiemi dalla Figura 2.9.

Esempio 2.10 Sia N l'automa della Figura 2.9, che accetta tutte le stringhe che finiscono per 01. Dato che l'insieme di stati di N è $\{q_0, q_1, q_2\}$, la costruzione per sottoinsiemi produce un DFA con $2^3 = 8$ stati, corrispondenti a tutti i sottoinsiemi dei tre stati. La Figura 2.12 mostra la tabella di transizione per questi otto stati; vedremo tra breve i dettagli su come le singole voci vengano computate.

Notiamo che la tabella di transizione appartiene a un automa a stati finiti deterministico. Anche se le voci nella tabella sono insiemi, gli stati del DFA così costruito *sono* insiemi. Per chiarire questo punto, possiamo inventare nomi nuovi per questi stati, per esempio A per $\emptyset$, B per $\{q_0\}$, e così via. La tabella di transizione della Figura 2.13 definisce esattamente lo stesso automa della Figura 2.12, ma chiarisce che le voci della tabella sono singoli stati del DFA.

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Figura 2.13 Ridenominazione degli stati della Figura 2.12.

Degli otto stati della Figura 2.13, partendo dallo stato iniziale B possiamo raggiungere solo gli stati B, E ed F. Gli altri cinque stati sono inaccessibili dallo stato iniziale e potrebbero anche non esserci. Spesso si può evitare il passo di costruire voci della tabella di transizione per ogni sottoinsieme di stati (cosa che richiede un tempo esponenziale) se si compie una "valutazione differita" dei sottoinsiemi. Vediamo come.

BASE Il singoletto formato dal solo stato iniziale di N è certamente accessibile.

INDUZIONE Supponiamo di aver determinato che l'insieme S è accessibile. Allora per ogni simbolo di input a computiamo l'insieme di stati $\delta_D(S, a)$; sappiamo che anche questi insiemi di stati saranno accessibili.

Per l'esempio in esame, sappiamo che $\{q_0\}$ è uno stato del DFA D. Troviamo che $\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ e $\delta_D(\{q_0\}, 1) = \{q_0\}$. Ambedue questi fatti si deducono dal diagramma di transizione della Figura 2.9 e osservando che su 0 esistono archi uscenti da q_0 verso sia q_0 sia q_1, mentre su 1 esiste un arco diretto solo verso q_0. Abbiamo dunque una riga della tabella di transizione per il DFA: la seconda riga nella Fig. 2.12.

Uno dei due insiemi computati è "vecchio"; $\{q_0\}$ è già stato considerato. $\{q_0, q_1\}$, invece, è nuovo e le sue transizioni devono essere calcolate. Troviamo $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ e $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$. A titolo di esempio esplicitiamo il secondo calcolo; sappiamo che

$$\delta_D(\{q_0, q_1\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$

Ora abbiamo la quinta riga della Figura 2.12 e abbiamo scoperto un nuovo stato di D, cioè $\{q_0, q_2\}$. Un calcolo simile ci dice che

$$\delta_D(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$
$$\delta_D(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \emptyset = \{q_0\}$$

Questi calcoli forniscono la sesta riga della Figura 2.12, ma producono solo insiemi di stati che sono già stati esaminati.

Quindi la costruzione per sottoinsiemi è arrivata a un punto fermo; ora conosciamo tutti gli stati accessibili e le loro transizioni. L'intero DFA è rappresentato dalla Figura 2.14. Osserviamo che ha solo tre stati, cioè, casualmente, lo stesso numero di stati dell'NFA della Figura 2.9 dal quale è stato costruito. D'altra parte il DFA della Figura 2.14 ha sei transizioni, a fronte delle quattro della Figura 2.9. □

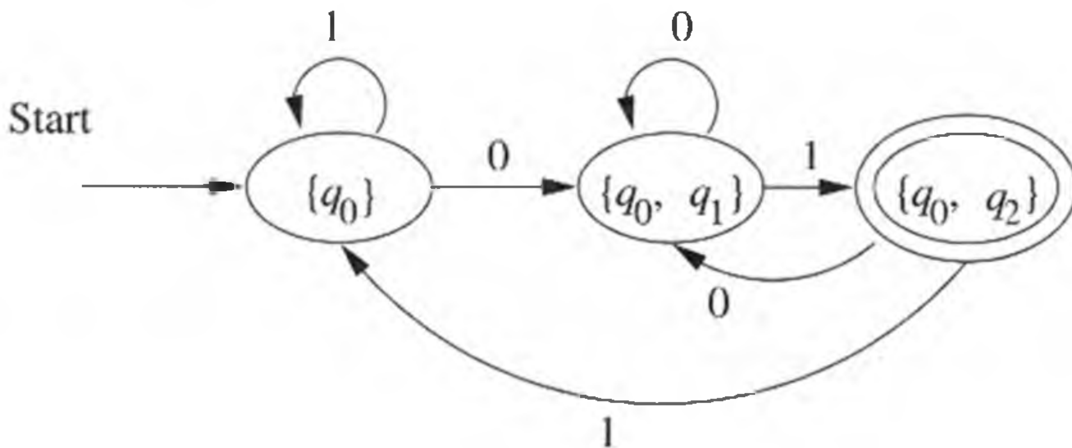


Figura 2.14 Il DFA costruito dall'NFA della Figura 2.9.

Dobbiamo dimostrare formalmente che la costruzione per sottoinsiemi è corretta, sebbene l'intuizione di ciò sia suffragata dagli esempi. Dopo aver letto la sequenza di simboli di input w, il DFA costruito si trova in un unico stato, che è l'insieme degli stati in cui si troverebbe l'NFA dopo aver letto w. Dato che gli stati accettanti del DFA sono quegli stati che includono almeno uno stato accettante dell'NFA, e anche l'NFA accetta se giunge in almeno uno dei suoi stati accettanti, possiamo concludere che il DFA e l'NFA accettano esattamente le stesse stringhe e dunque lo stesso linguaggio.

Teorema 2.11 Se $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ è il DFA costruito dall'NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ mediante la costruzione per sottoinsiemi, allora $L(D) = L(N)$.

DIMOSTRAZIONE In primo luogo dimostriamo, per induzione su $|w|$, che

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Si noti che ognuna delle funzioni $\hat{\delta}$ restituisce un insieme di stati di Q_N, ma $\hat{\delta}_D$ interpreta quest'insieme come uno degli stati di Q_D (che è l'insieme potenza di Q_N), mentre $\hat{\delta}_N$ lo interpreta come un sottoinsieme di Q_N.

BASE Sia $|w| = 0$, ossia $w = \epsilon$. Per le definizioni della base di $\hat{\delta}$ per i DFA e gli NFA, sia $\hat{\delta}_D(\{q_0\}, \epsilon)$ sia $\hat{\delta}_N(q_0, \epsilon)$ sono $\{q_0\}$.

INDUZIONE Sia w di lunghezza $n + 1$ e supponiamo vero l'enunciato per la lunghezza n. Scomponiamo w in $w = xa$, dove a è il simbolo finale di w. Per l'ipotesi induttiva, $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Questi due insiemi di stati di N siano $\{p_1, p_2, \ldots, p_k\}$.

La parte induttiva della definizione di $\hat{\delta}$ per gli NFA ci dice

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

La costruzione per sottoinsiemi, d'altra parte, dice che

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Serviamoci ora della (2.3) e del fatto che $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ nella parte induttiva della definizione di $\hat{\delta}$ per i DFA:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D\big(\hat{\delta}_D(\{q_0\}, x), a\big) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.4}$$

Dunque le Equazioni (2.2) e (2.4) dimostrano che $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. Osservando che sia D sia N accettano w se e solo se, rispettivamente, $\hat{\delta}_D(\{q_0\}, w)$ e $\hat{\delta}_N(q_0, w)$ contengono uno stato in F_N, abbiamo portato a compimento la dimostrazione che $L(D) = L(N)$. □

Teorema 2.12 Un linguaggio L è accettato da un DFA se e solo se L è accettato da un NFA.

DIMOSTRAZIONE (Se) La parte "se" è costituita dalla costruzione per sottoinsiemi e dal Teorema 2.11.

(Solo-se) Questa parte è facile; dobbiamo soltanto convertire un DFA in un NFA identico. In modo intuitivo, se abbiamo il diagramma di transizione di un DFA, possiamo anche interpretarlo come il diagramma di transizione di un NFA, in cui c'è esattamente una scelta di transizione in qualunque caso. In termini più formali, sia $D = (Q, \Sigma, \delta_D, q_0, F)$ un DFA. Definiamo l'NFA equivalente $N = (Q, \Sigma, \delta_N, q_0, F)$, dove δ_N è definita dalla seguente regola.

- Se $\delta_D(q, a) = p$, allora $\delta_N(q, a) = \{p\}$.

È facile dimostrare, per induzione su $|w|$, che se $\hat{\delta}_D(q_0, w) = p$ allora

$$\hat{\delta}_N(q_0, w) = \{p\}$$

Lasciamo che sia il lettore a farlo. Di conseguenza la stringa w viene accettata da D se e solo se viene accettata da N; cioè $L(D) = L(N)$. □

2.3.6 Un caso sfavorevole di costruzione per sottoinsiemi

Nell'Esempio 2.10 abbiamo visto che il DFA non aveva più stati di quanti ne avesse l'NFA equivalente. Come già detto, nella pratica è normale che il DFA abbia approssimativamente lo stesso numero di stati dell'NFA a partire dal quale è stato costruito. Tuttavia è possibile una crescita esponenziale del numero degli stati; tutti i 2^n stati del DFA che si possono costruire da un NFA di n stati potrebbero risultare accessibili. Il seguente esempio non raggiunge il limite, ma illustra un caso in cui il più piccolo DFA equivalente a un NFA di $n + 1$ stati ha 2^n stati.

Esempio 2.13 Consideriamo l'NFA N della Figura 2.15. $L(N)$ è l'insieme di tutte le stringhe di 0 e di 1 tali che l'n-esimo simbolo dalla fine sia 1. Intuitivamente un DFA D che accetti questo linguaggio deve ricordare gli ultimi n simboli che ha letto.

Poiché ci sono 2^n sequenze distinte di lunghezza n, se D avesse meno di 2^n stati, ci sarebbe uno stato q raggiunto dopo aver letto due sequenze distinte di n bit, poniamo $a_1a_2 \cdots a_n$ e $b_1b_2 \cdots b_n$.

Dato che le sequenze sono diverse, devono differire in una posizione, per esempio $a_i \neq b_i$. Supponiamo (per simmetria) che $a_i = 1$ e $b_i = 0$. Se $i = 1$, allora q deve essere sia uno stato accettante sia uno stato non accettante, visto che $a_1a_2 \cdots a_n$ è accettato (l'n-esimo simbolo dalla fine è 1), e invece $b_1b_2 \cdots b_n$ no. Se $i > 1$, allora consideriamo lo stato p in cui D entra dopo aver letto $i - 1$ simboli 0 da q. Allora p deve essere sia accettante sia non accettante, dato che $a_ia_{i+1} \cdots a_n00 \cdots 0$ è accettato e $b_ib_{i+1} \cdots b_n00 \cdots 0$ no.

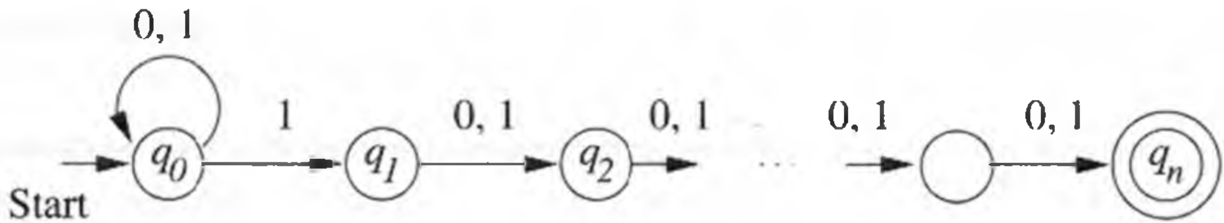


Figura 2.15 Questo NFA non ha alcun DFA equivalente con meno di 2^n stati.

Passiamo ora al funzionamento dell'NFA N della Figura 2.15. Esiste uno stato q_0 in cui l'NFA si trova sempre, a prescindere dagli input letti. Se l'input successivo è 1, N può anche "scommettere" che questo 1 sarà l'n-esimo simbolo dalla fine, quindi passa allo stato q_1 così come a q_0. Dallo stato q_1 qualunque input porta N in q_2, l'input successivo lo porta in q_3, e così via, finché, $n - 1$ input dopo, si trova nello stato accettante q_n. Esprimiamo ora l'enunciato formale relativo alla condotta degli stati di N.

1. N si trova nello stato q_0 dopo aver letto una qualunque sequenza di input w.

2. N si trova nello stato q_i, per $i = 1, 2, \ldots, n$, dopo aver letto la sequenza di input w se e solo se l'i-esimo simbolo dalla fine di w è 1; in altre parole w è della forma $x1a_1a_2 \cdots a_{i-1}$, dove ogni a_j è un simbolo di input.

Non dimostreremo questi enunciati formalmente; la dimostrazione è una facile induzione su $|w|$, del tipo illustrato nell'Esempio 2.9. Per completare la dimostrazione che l'automa accetta esattamente le stringhe con un 1 nell'n-esima posizione dalla fine, consideriamo l'enunciato (2) con $i = n$, secondo il quale N si trova nello stato q_n se e solo se l'n-esimo simbolo dalla fine è 1. Essendo però q_n l'unico stato accettante, tale condizione caratterizza precisamente l'insieme delle stringhe accettate da N. □

2.3.7 Esercizi

* **Esercizio 2.3.1** Convertite in un DFA il seguente NFA:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$*s$	$\{s\}$	$\{s\}$

Esercizio 2.3.2 Convertite in un DFA il seguente NFA:

Il principio della piccionaia

Nell'Esempio 2.13 abbiamo impiegato un'importante tecnica di ragionamento, detta *principio della piccionaia*. In parole povere, se si hanno più piccioni che cellette e ogni piccione si ricovera in una celletta, allora ci deve essere almeno una celletta contenente più di un piccione. Nel nostro caso i "piccioni" sono le sequenze di n bit e le "cellette" sono gli stati. Poiché esistono meno stati che sequenze, a uno stato devono essere assegnate due sequenze.

Il principio della piccionaia può sembrare ovvio, ma dipende dal fatto che il numero delle cellette è finito. Funziona perciò per automi a stati finiti, in cui gli stati corrispondono alle cellette della piccionaia. Non si può applicare invece ad altri tipi di automi che hanno un numero infinito di stati.

Per capire perché è essenziale che il numero delle cellette sia finito, si consideri la situazione infinita in cui le cellette corrispondono agli interi $1, 2, \ldots$. Numeriamo i piccioni $0, 1, 2, \ldots$, in modo che ci sia un piccione in più rispetto alle cellette. Possiamo allora mandare il piccione i nella celletta $i + 1$ per ogni $i \geq 0$. Così ogni piccione ha la sua celletta e non ci sono due piccioni obbligati a condividere lo stesso spazio.

	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\emptyset$	$\{p\}$

! **Esercizio 2.3.3** Convertite il seguente NFA in un DFA e descrivete informalmente il linguaggio che accetta.

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r, s\}$	$\{t\}$
r	$\{p, r\}$	$\{t\}$
$*s$	$\emptyset$	$\emptyset$
$*t$	$\emptyset$	$\emptyset$

! **Esercizio 2.3.4** Definite automi a stati finiti non deterministici che accettino i seguenti linguaggi. Cercate di sfruttare il più possibile il non determinismo.

Stati trappola e DFA cui mancano alcune transizioni

Abbiamo formalmente definito un DFA in modo che per ogni stato e per ogni simbolo di input ci sia esattamente una transizione verso un altro stato. A volte, però, in situazioni in cui sappiamo che nessuna estensione della sequenza di input può essere accettata, è più opportuno fare in modo che il DFA "muoia". Per esempio osserviamo l'automa della Figura 1.2, che svolge il proprio compito riconoscendo una sola parola, `then`, e nient'altro. Quest'automa non è tecnicamente un DFA, in quanto gli mancano le transizioni sulla maggior parte dei simboli da ciascuno dei suoi stati.

Tuttavia tale automa è un NFA. Se utilizziamo la costruzione per sottoinsiemi in modo da convertirlo in un DFA, l'automa sembra quasi lo stesso, ma include uno *stato trappola*, ossia uno stato non accettante che conduce a se stesso su ogni possibile simbolo di input. Lo stato trappola corrisponde a $\emptyset$, l'insieme vuoto di stati dell'automa della Figura 1.2.

In generale possiamo aggiungere uno stato trappola a qualunque automa che abbia *non più* di una transizione per ogni stato e simbolo di input: si aggiunge una transizione verso lo stato trappola da uno stato q, su tutti i simboli di input per i quali q non ha transizioni. Il risultato sarà un DFA nel senso più stretto. A volte, perciò, faremo riferimento a un automa come un DFA se ha *al massimo* una transizione uscente da ciascuno stato su ogni simbolo, piuttosto che se ha *esattamente una* transizione.

* a) L'insieme delle stringhe sull'alfabeto $\{0, 1, \ldots, 9\}$ tali che la cifra finale sia comparsa in precedenza.

b) L'insieme delle stringhe sull'alfabeto $\{0, 1, \ldots, 9\}$ tali che la cifra finale *non* sia comparsa in precedenza.

c) L'insieme delle stringhe di 0 e di 1 tali che esistano due 0 separati da un numero di posizioni multiplo di 4. Si noti che 0 è un multiplo di 4.

Esercizio 2.3.5 Nella parte solo-se del Teorema 2.12 abbiamo omesso la dimostrazione per induzione su $|w|$ che se $\hat{\delta}_D(q_0, w) = p$ allora $\hat{\delta}_N(q_0, w) = \{p\}$. Fornite tale dimostrazione.

! **Esercizio 2.3.6** Nel riquadro su "Stati trappola e DFA cui mancano alcune transizioni" sosteniamo che, se N è un NFA che ha al massimo una scelta di stato per ogni stato e simbolo di input (ossia $\delta(q, a)$ non ha mai più di un elemento), allora il DFA D costruito

da N con la costruzione per sottoinsiemi ha esattamente gli stati e le transizioni di N, più le transizioni verso un nuovo stato trappola ogni volta che a N manca una transizione per uno stato e un simbolo di input dati. Dimostrate questa affermazione.

Esercizio 2.3.7 Nell'Esempio 2.13 abbiamo affermato che l'NFA N si trova nello stato q_i, per $i = 1, 2, \ldots, n$, dopo aver letto la sequenza di input w, se e solo se l'i-esimo simbolo dalla fine di w è 1. Dimostrate quest'asserzione.

2.4 Un'applicazione: ricerche testuali

In questo paragrafo vedremo che la trattazione teorica presentata nel paragrafo precedente, in cui abbiamo considerato il problema di decidere se una sequenza di bit finisce per 01, è in effetti un modello appropriato per molti problemi reali che si presentano in applicazioni come le ricerche nel Web e il prelievo di informazioni da un testo.

2.4.1 Ricerca di stringhe in un testo

Un problema comune nell'era del Web e di altre raccolte di testi on-line è il seguente: dato un insieme di parole, trovare tutti i documenti che ne contengono una (o tutte). Un motore di ricerca è un tipico esempio di tale procedura. Un motore di ricerca usa una tecnica particolare, detta *indici invertiti*, in cui per ogni parola che compare nel Web (ci sono 100.000.000 di parole diverse) viene memorizzata una lista di tutti i luoghi in cui questa si incontra. Una macchina con una memoria centrale molto grande può tenere a disposizione le liste più frequenti, consentendo ricerche simultanee da parte di più utenti.

Le tecniche a indici invertiti non utilizzano gli automi a stati finiti, ma richiedono molto tempo per la copia delle pagine e la preparazione degli indici. Esistono svariate applicazioni affini per le quali gli indici invertiti non sono adatti, mentre le tecniche fondate sugli automi risultano appropriate. Le caratteristiche che rendono un'applicazione adeguata alle ricerche che impiegano gli automi sono due.

1. Il deposito sul quale viene condotta la ricerca cambia rapidamente. Per esempio:

   (a) gli analisti della comunicazione esplorano quotidianamente le notizie on-line del giorno in cerca di argomenti specifici; per esempio un analista finanziario potrebbe cercare le sigle di particolari azioni o nomi di aziende

   (b) un "robot della spesa" potrebbe cercare i prezzi correnti degli articoli richiesti dai suoi clienti; il robot può recuperare dal Web pagine di catalogo e cercarvi le parole che indicano il prezzo di un particolare articolo.

2. I documenti da esaminare non possono essere catalogati. Per esempio su Amazon.com un *crawler* difficilmente troverà tutte le pagine relative a ogni libro in

vendita. Tali pagine sono generate dinamicamente in risposta a un'interrogazione. Si potrebbero allora cercare i libri su un certo argomento, per esempio "automi a stati finiti", e poi nelle pagine trovate cercare determinate parole, per esempio "eccellente", nelle recensioni.

2.4.2 Automi a stati finiti non deterministici per ricerche testuali

Supponiamo che ci venga dato un insieme di parole, che chiameremo *parole chiave*; vogliamo trovare le occorrenze di una qualunque di queste parole. In applicazioni come queste è utile ideare un automa a stati finiti non deterministico che segnali, entrando in uno stato accettante, di aver letto una delle parole chiave. Il testo di un documento viene passato, un carattere alla volta, all'NFA, che vi riconosce le occorrenze delle parole chiave. Descriviamo una forma semplice di NFA che riconosce un insieme di parole chiave.

1. C'è uno stato iniziale con una transizione verso se stesso su ogni simbolo di input, per esempio ogni carattere ASCII stampabile se si sta esaminando un testo. Intuitivamente lo stato iniziale rappresenta la "congettura" che non abbiamo ancora cominciato a vedere una delle parole chiave, anche se abbiamo visto alcune lettere di una di queste parole.

2. Per ogni parola chiave $a_1a_2 \cdots a_k$, ci sono k stati, $q_1, q_2, \ldots, q_k$. Una transizione porta dallo stato iniziale verso q_1 sul simbolo a_1, una transizione da q_1 verso q_2 sul simbolo a_2, e così via. Lo stato q_k è uno stato accettante e indica che è stata trovata la parola chiave $a_1a_2 \cdots a_k$.

Esempio 2.14 Facciamo l'ipotesi di voler ideare un NFA che riconosca le occorrenze delle parole `web` e `ebay`. Il diagramma di transizione per l'NFA costruito secondo le regole di cui sopra è illustrato nella Figura 2.16. Lo stato 1 è lo stato iniziale, e usiamo Σ per indicare l'insieme di tutti i caratteri ASCII stampabili. Gli stati dal 2 al 4 hanno il compito di riconoscere `web`, mentre gli stati dal 5 all'8 riconoscono `ebay`. □

Ovviamente un NFA non è un programma. Per implementare un NFA ci sono in sostanza due possibilità.

1. Scrivere un programma che simuli l'NFA computando l'insieme degli stati in cui si trova dopo aver letto ogni simbolo di input. La simulazione è illustrata nella Figura 2.10.

2. Convertire l'NFA in un DFA equivalente usando la costruzione per sottoinsiemi, quindi simulare direttamente il DFA.

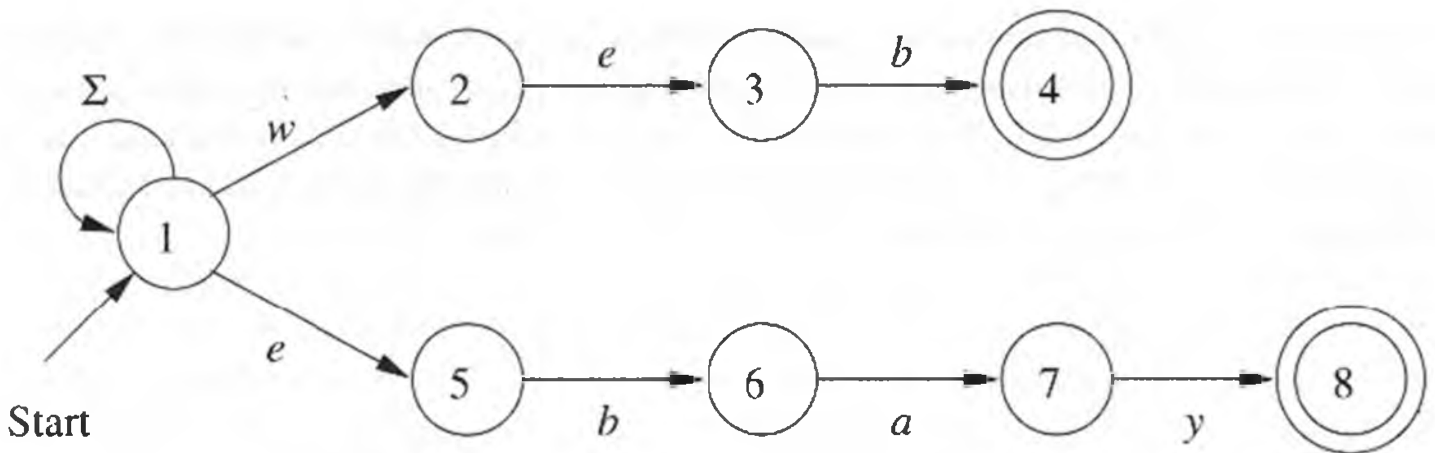


Figura 2.16 Un NFA che cerca le parole `web` e `ebay`.

Alcuni programmi che elaborano testi, come le forme avanzate del comando UNIX `grep` (`egrep` e `fgrep`), in realtà uniscono le due soluzioni. Per i nostri scopi la conversione in un DFA è facile e siamo sicuri che il numero di stati non cresce.

2.4.3 Un DFA per riconoscere un insieme di parole chiave

La costruzione per sottoinsiemi si può applicare a qualunque NFA. Quando la applichiamo a un NFA costruito per un insieme di parole chiave, secondo la strategia del Paragrafo 2.4.2, troviamo che il numero di stati del DFA non è mai maggiore del numero di stati dell'NFA. Poiché invece, nel caso peggiore, il numero di stati cresce esponenzialmente passando a un DFA, quello preso in esame è un caso favorevole, e spiega per quale ragione si usa spesso il metodo di determinare un NFA per le parole chiave e di costruire da questo un DFA. Ecco le regole per costruire l'insieme degli stati del DFA.

a) Se q_0 è lo stato iniziale dell'NFA, allora $\{q_0\}$ è uno degli stati del DFA.

b) Supponiamo che p sia uno degli stati dell'NFA e che venga raggiunto dallo stato iniziale lungo un cammino i cui simboli sono $a_1a_2 \cdots a_m$. Allora uno degli stati del DFA è l'insieme degli stati dell'NFA formato da:

   1. q_0

   2. p

   3. ogni altro stato dell'NFA raggiungibile da q_0 seguendo un cammino le cui etichette sono un suffisso di $a_1a_2 \cdots a_m$, cioè qualunque sequenza di simboli della forma $a_ja_{j+1} \cdots a_m$.

Si noti che in generale nel DFA ci sarà uno stato per ogni stato p dell'NFA. Tuttavia nel passo (b) due stati possono dare come risultato lo stesso insieme di stati, e dunque diventare un solo stato del DFA. Per esempio, se due delle parole chiave iniziano con la stessa lettera, poniamo a, allora i due stati dell'NFA raggiunti da q_0 con un arco etichettato a daranno lo stesso insieme di stati e quindi verranno fusi nel DFA.

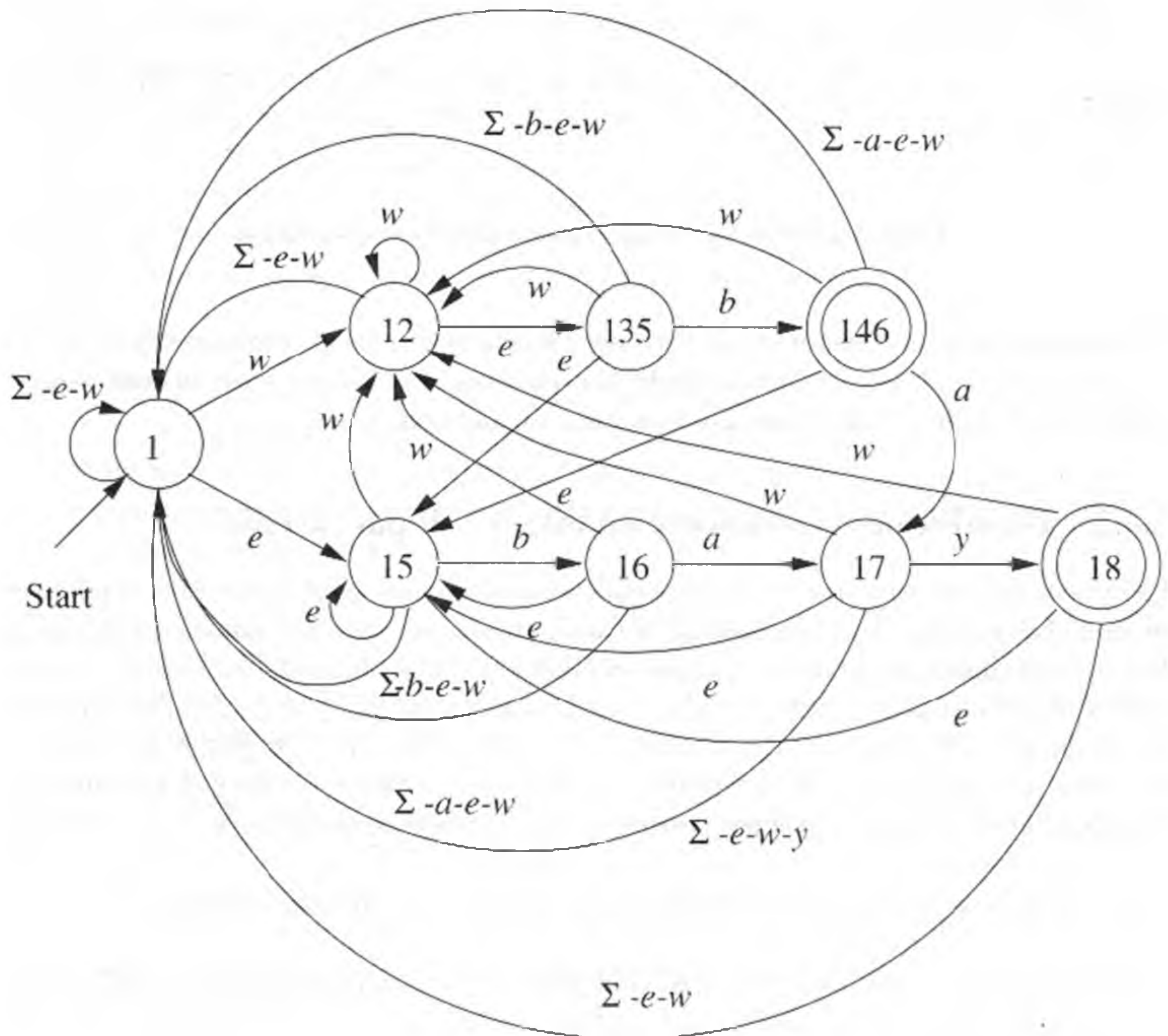


Figura 2.17 Conversione in DFA dell'NFA nella Figura 2.16.

Esempio 2.15 La costruzione di un DFA dall'NFA della Figura 2.16 è illustrata nella Figura 2.17. Ogni stato del DFA è collocato nella stessa posizione dello stato p da cui è ricavato usando la regola (b). Consideriamo per esempio lo stato 135, che è l'abbreviazione per $\{1, 3, 5\}$. Tale stato è stato costruito dallo stato 3; come ogni altro insieme di

stati del DFA, include lo stato iniziale 1; include inoltre lo stato 5, perché a questo stato si perviene dallo stato 1 attraverso un suffisso, e, della stringa we, che raggiunge lo stato 3 nella Figura 2.16.

Le transizioni per ognuno degli stati del DFA si possono calcolare conformemente alla costruzione per sottoinsiemi. La regola è comunque semplice. Da qualunque insieme di stati che includa lo stato iniziale q_0 e altri stati $\{p_1, p_2, \ldots, p_n\}$, si determina dove si sposta l'NFA dai p_i per ogni simbolo x, e si fa compiere al DFA una transizione etichettata x verso lo stato che consiste di q_0 e di tutti gli stati raggiunti dai p_i sul simbolo x. Per ogni simbolo x tale che non esistano transizioni uscenti da nessun p_i, si fa compiere a questo stato del DFA una transizione su x verso lo stato che consiste di q_0 e di tutti gli stati che vengono raggiunti da q_0 nell'NFA seguendo un arco etichettato x.

Consideriamo per esempio lo stato 135 della Figura 2.17. L'NFA della Figura 2.16 ha transizioni sul simbolo b dagli stati 3 e 5 rispettivamente verso gli stati 4 e 6. Perciò, sul simbolo b, 135 va verso 146. Sul simbolo e non esistono transizioni dell'NFA in uscita da 3 o 5, ma c'è una transizione da 1 a 5. Quindi, nel DFA, 135 va verso 15 su input e. Analogamente, su input w, 135 va verso 12.

Su ogni altro simbolo x non ci sono transizioni uscenti da 3 o 5 e lo stato 1 va solamente verso se stesso. Dunque esistono transizioni da 135 a 1 su ogni simbolo in Σ che non sia b, e o w. Applichiamo la notazione $\Sigma - b - e - w$ per rappresentare quest'insieme e usiamo rappresentazioni simili per gli altri insiemi ottenuti rimuovendo simboli da Σ. □

2.4.4 Esercizi

Esercizio 2.4.1 Ideate gli NFA che riconoscono i seguenti insiemi di stringhe.

* a) `abc`, `abd` e `aacd`. Assumete che l'alfabeto sia $\{a, b, c, d\}$.

b) `0101`, `101` e `011`.

c) `ab`, `bc` e `ca`. Assumete che l'alfabeto sia $\{a, b, c\}$.

Esercizio 2.4.2 Convertite ogni NFA dell'Esercizio 2.4.1 in un DFA.

2.5 Automi a stati finiti con epsilon-transizioni

Presentiamo ora un'altra estensione degli automi a stati finiti. La novità consiste nell'ammettere transizioni sulla stringa vuota ϵ. È come se l'NFA potesse compiere una transizione spontaneamente, senza aver ricevuto un simbolo di input. Come il non determinismo, introdotto nel Paragrafo 2.3, anche questa nuova possibilità non amplia la classe dei linguaggi accettati dagli automi a stati finiti, ma offre una certa comodità notazionale.

Nel Paragrafo 3.1 vedremo che gli NFA con ϵ-transizioni, che chiameremo *ϵ-NFA*, sono strettamente legati alle espressioni regolari e sono utili nel dimostrare l'equivalenza fra la classe dei linguaggi accettati dagli automi a stati finiti e quella dei linguaggi specificati da espressioni regolari.

2.5.1 Uso delle ϵ-transizioni

Cominceremo da una trattazione informale degli ϵ-NFA, servendoci di diagrammi di transizione con etichette ϵ. Negli esempi che seguono, gli automi accettano le sequenze di etichette lungo cammini dallo stato iniziale a uno stato accettante, considerando come invisibili le occorrenze di ϵ, che quindi non contribuiscono a formare le stringhe associate ai cammini.

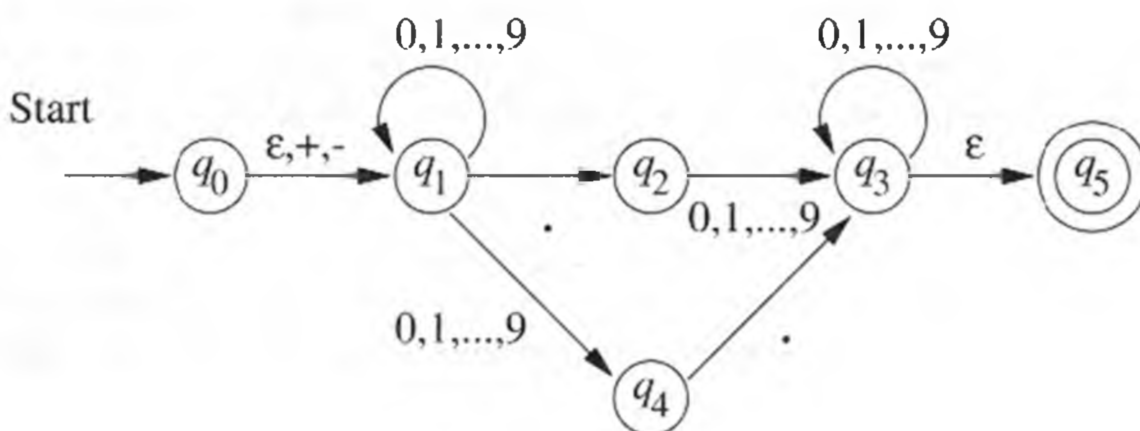


Figura 2.18 Un ϵ-NFA che accetta numeri decimali.

Esempio 2.16 La Figura 2.18 mostra un ϵ-NFA che accetta numeri decimali (in notazione anglosassone) formati da:

1. un segno $+$ o $-$, facoltativo
2. una sequenza di cifre
3. un punto decimale
4. una seconda sequenza di cifre.

Una delle due sequenze di cifre (punti (2) e (4)), ma non entrambe, può essere vuota. Di particolare interesse è la transizione da q_0 a q_1 su ϵ, $+$ o $-$. Lo stato q_1 rappresenta quindi la situazione in cui sono stati letti il segno, se presente, ed eventualmente delle cifre, ma non il punto decimale. Lo stato q_2 rappresenta la situazione in cui è stato letto il punto decimale, preceduto o no da cifre già lette. In q_4 abbiamo letto almeno una cifra, ma non

ancora il punto decimale. Lo stato q_3 si interpreta così: sono stati letti il punto decimale e almeno una cifra, prima o dopo di quello. L'automa può restare in q_3 e leggere nuove cifre, oppure può "scommettere" che le cifre sono terminate e spostarsi autonomamente nello stato accettante q_5. □

Esempio 2.17 La strategia delineata nell'Esempio 2.14 per costruire un NFA che riconosca un insieme di parole si può semplificare se si ammettono le ϵ-transizioni. Per esempio l'NFA che riconosce le parole `web` e `ebay`, illustrato nella Figura 2.16, si può anche realizzare mediante ϵ-transizioni, come nella Figura 2.19. In generale si costruisce una sequenza completa di stati per ogni parola come se questa fosse l'unica da riconoscere, e si aggiunge un nuovo stato iniziale (lo stato 9 nella Figura 2.19) da cui si raggiungono, con ϵ-transizioni, gli stati iniziali degli automi per le singole parole. □

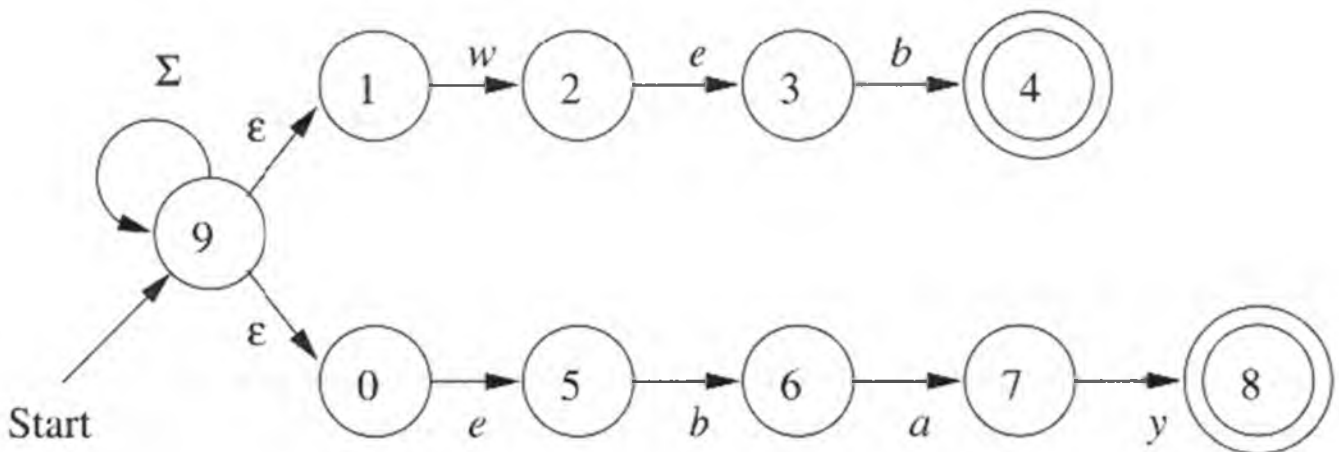


Figura 2.19 Uso di ϵ-transizioni per riconoscere parole.

2.5.2 La notazione formale per gli ϵ-NFA

Possiamo rappresentare un ϵ-NFA esattamente come un NFA, salvo che per un aspetto: la funzione di transizione deve incorporare informazioni sulle transizioni etichettate da ϵ. Formalmente denotiamo un ϵ-NFA con $A = (Q, \Sigma, \delta, q_0, F)$, dove tutti i componenti si interpretano come per gli NFA, mentre δ è una funzione che richiede come argomenti:

1. uno stato in Q

2. un elemento di $\Sigma \cup \{\epsilon\}$, cioè un simbolo di input o il simbolo ϵ. Per non fare confusione, si richiede che il simbolo di stringa vuota ϵ non sia un elemento dell'alfabeto Σ.

Esempio 2.18 L'ϵ-NFA della Figura 2.18 è specificato formalmente da

$$E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}, \delta, q_0, \{q_5\})$$

dove δ è definita dalla tabella della Figura 2.20. □

	ϵ	$+, -$	$.$	$0, 1, \ldots, 9$
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figura 2.20 Tabella di transizione per la Figura 2.18.

2.5.3 Epsilon-chiusure

Procediamo nelle definizioni formali di una funzione di transizione estesa per gli ϵ-NFA, così da stabilire le stringhe e i linguaggi accettati da tali automi, per poi spiegare in che modo un ϵ-NFA può essere simulato da un DFA. Prima, però, dobbiamo introdurre una nozione cruciale: la cosiddetta *ϵ-chiusura* di uno stato. Informalmente, per ϵ-chiudere uno stato q si seguono tutte le ϵ-transizioni uscenti da q, ripetendo poi l'operazione da tutti gli stati raggiunti via via, fino a trovare tutti gli stati raggiungibili da q attraverso cammini etichettati solo da ϵ-transizioni. Definiamo formalmente l'ϵ-chiusura, ECLOSE(q), in modo ricorsivo, come segue.

BASE Lo stato q appartiene a ECLOSE(q).

INDUZIONE Se lo stato p appartiene a ECLOSE(q) ed esiste una transizione, etichettata ϵ, da p a r, allora r appartiene a ECLOSE(q). Più precisamente, sia δ la funzione di transizione di un ϵ-NFA; se p appartiene a ECLOSE(q), allora ECLOSE(q) contiene anche tutti gli stati in $\delta(p, \epsilon)$.

Esempio 2.19 Nell'automa della Figura 2.18 ogni stato coincide con la propria ϵ-chiusura, con due eccezioni: ECLOSE(q_0) $= \{q_0, q_1\}$ e ECLOSE(q_3) $= \{q_3, q_5\}$. Il motivo è che ci sono soltanto due ϵ-transizioni, la prima che aggiunge q_1 a ECLOSE(q_0) e la seconda che aggiunge q_5 a ECLOSE(q_3).

La Figura 2.21 illustra un esempio più complesso. Per questa famiglia di stati, che potrebbe far parte di un ϵ-NFA, possiamo concludere che

$$\text{ECLOSE}(1) = \{1, 2, 3, 4, 6\}$$

Ognuno di questi stati si può raggiungere dallo stato 1 lungo un cammino etichettato solo da ϵ. Lo stato 6, per esempio, si raggiunge con il cammino $1 \to 2 \to 3 \to 6$. Lo stato 7 non appartiene a ECLOSE(1), pur essendo raggiungibile da 1, perché il cammino deve passare per l'arco $4 \to 5$, che non è etichettato ϵ. Il fatto che lo stato 6 sia raggiungibile anche lungo il cammino $1 \to 4 \to 5 \to 6$, che comprende transizioni con etichette diverse da ϵ, non è rilevante: l'esistenza di un cammino fatto di sole ϵ-transizioni è sufficiente a porre lo stato 6 in ECLOSE(1). □

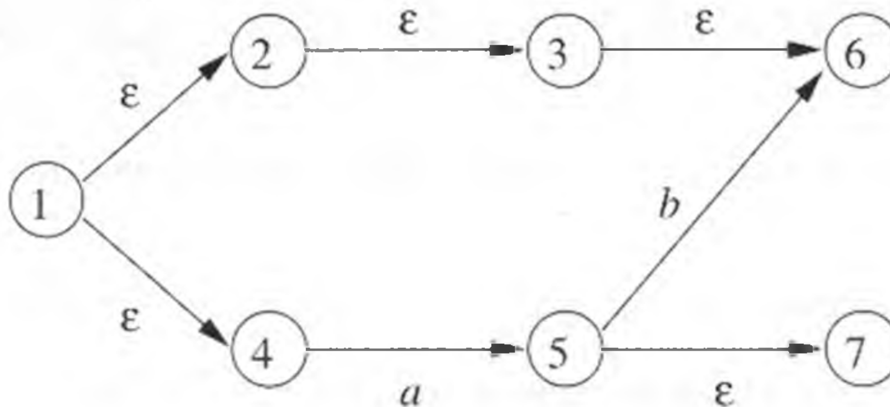


Figura 2.21 Esempi di stati e transizioni.

2.5.4 Transizioni estese e linguaggi per gli ϵ-NFA

Tramite l'ϵ-chiusura è facile spiegare il comportamento di un ϵ-NFA a fronte di una sequenza di input diversi da ϵ; quindi possiamo definire che cosa significa "accettare un input" per un ϵ-NFA.

Supponiamo che $E = (Q, \Sigma, \delta, q_0, F)$ sia un ϵ-NFA. Definiamo anzitutto $\hat{\delta}$, la funzione di transizione estesa, che descrive l'evoluzione per una sequenza di valori d'ingresso. Vogliamo che $\hat{\delta}(q, w)$ sia l'insieme degli stati raggiungibili attraverso percorsi le cui etichette, concatenate, formano la stringa w. Come sempre, le ϵ lungo un cammino non contribuiscono a w. Ecco un'adeguata definizione ricorsiva di $\hat{\delta}$:

BASE $\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$. Se l'etichetta del cammino è ϵ, si possono seguire solo archi etichettati ϵ uscenti da q; questo è proprio il modo in cui ECLOSE viene definito.

INDUZIONE Supponiamo che w abbia la forma xa, dove a è l'ultimo simbolo. Si noti che a appartiene a Σ, e quindi non può essere ϵ, che non appartiene a Σ. Calcoliamo $\hat{\delta}(q, w)$ come segue.

1. Poniamo $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$. In tal modo i p_i sono tutti e soli gli stati raggiungibili da q lungo cammini etichettati x. Un tale cammino può terminare con una o più ϵ-transizioni e può contenere altre ϵ-transizioni.

2. Sia $\bigcup_{i=1}^{k} \delta(p_i, a)$ l'insieme $\{r_1, r_2, \ldots, r_m\}$. In altre parole si seguono tutte le transizioni etichettate a dagli stati che si raggiungono da q con cammini etichettati x. Gli r_j sono *alcuni* degli stati raggiungibili da q con cammini etichettati w. Gli altri stati raggiungibili si trovano seguendo ϵ-archi uscenti dagli r_j, nel passo (3), più avanti.

3. Infine $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$. Questo ulteriore passo di chiusura include tutti i cammini da q etichettati w, tenendo conto di eventuali ϵ-archi dopo l'ultimo simbolo effettivo a.

Esempio 2.20 Calcoliamo $\hat{\delta}(q_0, 5.6)$ per l'ϵ-NFA della Figura 2.18. Ricapitoliamo i passi necessari.

- $\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$.

- Il valore di $\hat{\delta}(q_0, 5)$ si calcola come segue.

  1. Si determinano per prima cosa le transizioni su input 5 dagli stati q_0 e q_1, ottenuti in precedenza nel calcolo di $\hat{\delta}(q_0, \epsilon)$; in altre parole si calcola $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.
  2. Si applica quindi la ϵ-chiusura ai membri dell'insieme calcolato al passo (1). Si ottiene così $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$. Questo insieme è $\hat{\delta}(q_0, 5)$. Per i due simboli successivi si ripetono gli stessi due passi.

- Si calcola $\hat{\delta}(q_0, 5.)$ come segue.

  1. Dapprima si calcola $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Poi

  $$\hat{\delta}(q_0, 5.) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}.$$

- Si calcola $\hat{\delta}(q_0, 5.6)$ come segue.

  1. Dapprima si calcola $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.

2. Poi $\hat{\delta}(q_0, 5.6) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$.

□

A questo punto possiamo definire il linguaggio di un ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ come il lettore può immaginare: $L(E) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$. Il linguaggio di E è l'insieme delle stringhe w che portano dallo stato iniziale ad almeno uno stato finale. Nell'Esempio 2.20 abbiamo infatti osservato che $\hat{\delta}(q_0, 5.6)$ contiene lo stato accettante q_5; ne deriva che la stringa 5 . 6 appartiene al linguaggio di quell'ϵ-NFA.

2.5.5 Eliminazione di ϵ-transizioni

Dato un ϵ-NFA E, possiamo trovare un DFA D che accetta lo stesso linguaggio di E. La costruzione impiegata qui è molto simile a quella per sottoinsiemi: gli stati di D sono sottoinsiemi di stati di E. C'è una sola differenza: dobbiamo incorporare le ϵ-transizioni di E mediante il meccanismo di ϵ-chiusura.

Sia $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Il DFA equivalente

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

è definito come segue.

1. Q_D è l'insieme dei sottoinsiemi di Q_E. Più esattamente scopriremo che tutti gli stati accessibili di D sono sottoinsiemi *ϵ-chiusi* di Q_E, cioè insiemi $S \subseteq Q_E$ tali che $S = \text{ECLOSE}(S)$. In altre parole un insieme di stati S è ϵ-chiuso se ogni ϵ-transizione uscente da uno stato di S porta a uno stato di S. Si noti che $\emptyset$ è un insieme ϵ-chiuso.

2. $q_D = \text{ECLOSE}(q_0)$; lo stato iniziale di D si ottiene chiudendo l'insieme composto dal solo stato iniziale di E. Si osservi che questa regola si distingue dalla costruzione originale per sottoinsiemi, nella quale lo stato iniziale dell'automa costruito è l'insieme contenente solo lo stato iniziale dell'NFA dato.

3. F_D è formato dagli insiemi di stati che contengono almeno uno stato accettante di E. Quindi $F_D = \{S \mid S \text{ è in } Q_D \text{ e } S \cap F_E \neq \emptyset\}$.

4. Per ogni a in Σ e per ogni insieme S in Q_D, $\delta_D(S, a)$ si calcola come segue:

   (a) sia $S = \{p_1, p_2, \ldots, p_k\}$

   (b) si determini l'insieme $\{r_1, r_2, \ldots, r_m\} = \bigcup_{i=1}^{k} \delta_E(p_i, a)$

   (c) allora $\delta_D(S, a) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

Esempio 2.21 Vogliamo eliminare le ϵ-transizioni dall'ϵ-NFA della Figura 2.18, che in seguito chiameremo E. Da E costruiamo un DFA D, illustrato nella Figura 2.22. Per chiarezza abbiamo omesso dalla figura lo stato trappola $\emptyset$ e le transizioni verso di esso. Il lettore deve immaginare che per ogni stato rappresentato ci siano transizioni supplementari verso $\emptyset$ su ogni simbolo di input per il quale non c'è una transizione esplicita. Inoltre dallo stato $\emptyset$ parte una transizione verso lo stesso stato per ogni simbolo di input.

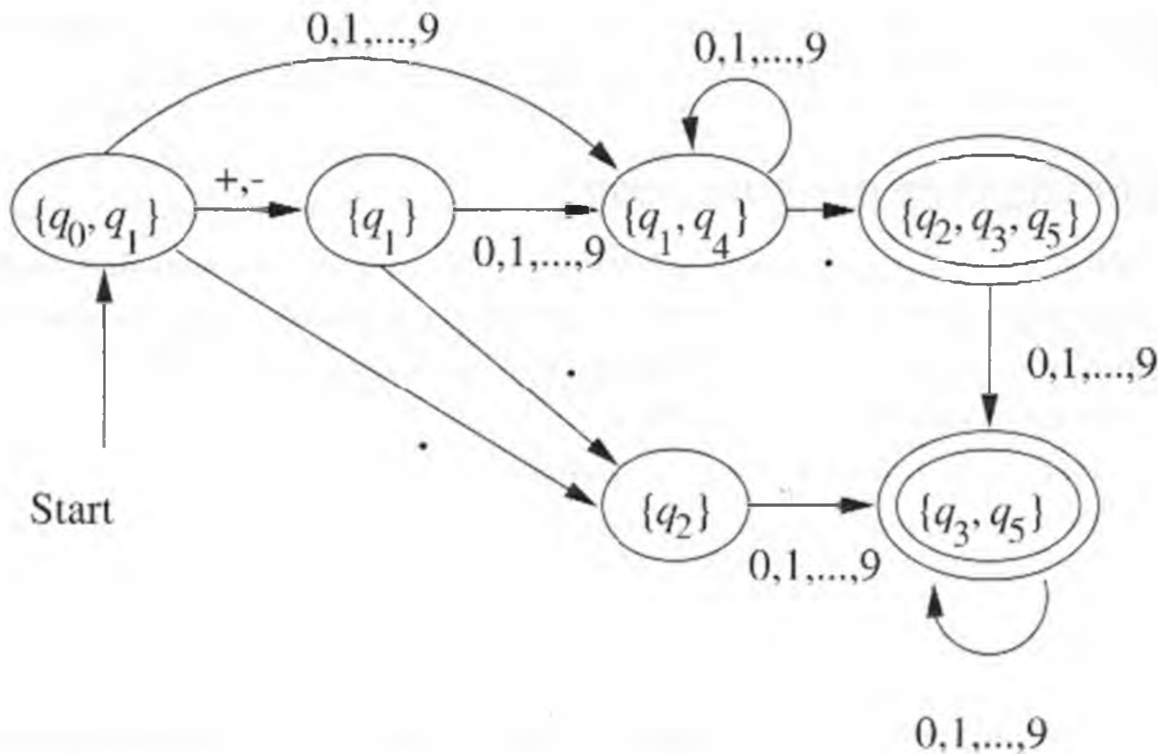


Figura 2.22 Il DFA D che elimina le ϵ-transizioni dalla Figura 2.18.

Poiché lo stato iniziale di E è q_0, lo stato iniziale di D è $\text{ECLOSE}(q_0)$, cioè $\{q_0, q_1\}$. Dobbiamo anzitutto trovare i successori di q_0 e q_1 rispetto ai simboli di Σ; tali simboli sono i segni "più" e "meno", il punto e le cifre da 0 a 9. Nella Figura 2.18 osserviamo che q_1 non ha archi uscenti per $+$ e $-$, mentre q_0 va in q_1. Dunque per calcolare $\delta_D(\{q_0, q_1\}, +)$ applichiamo la ϵ-chiusura a $\{q_1\}$. Non essendoci ϵ-transizioni uscenti da q_1, abbiamo $\delta_D(\{q_0, q_1\}, +) = \{q_1\}$. Analogamente $\delta_D(\{q_0, q_1\}, -) = \{q_1\}$. Le due transizioni sono rappresentate da un solo arco nella Figura 2.22.

Dobbiamo ora calcolare $\delta_D(\{q_0, q_1\}, .)$. Poiché q_0 non ha archi uscenti per il punto, e q_1 va in q_2 (Figura 2.18), dobbiamo ϵ-chiudere $\{q_2\}$. Non ci sono ϵ-transizioni uscenti da q_2, quindi il solo q_2 forma la propria chiusura: $\delta_D(\{q_0, q_1\}, .) = \{q_2\}$.

Infine calcoliamo $\delta_D(\{q_0, q_1\}, 0)$ come esempio delle transizioni da $\{q_0, q_1\}$ sulle cifre. Scopriamo che q_0 non ha archi uscenti sulle cifre, mentre q_1 va in q_1 e in q_4. Poiché nessuno di tali stati ha ϵ-transizioni uscenti, concludiamo che $\delta_D(\{q_0, q_1\}, 0) = \{q_1, q_4\}$; per le altre cifre si procede nello stesso modo.

Abbiamo così giustificato gli archi uscenti da $\{q_0, q_1\}$ della Figura 2.22. Le altre transizioni si determinano in modo simile; i dettagli sono lasciati al lettore. Poiché q_5 è l'unico stato accettante di E, gli stati accettanti di D sono gli stati accessibili che contengono q_5, cioè $\{q_3, q_5\}$ e $\{q_2, q_3, q_5\}$, indicati da doppi circoli nella Figura 2.22. □

Teorema 2.22 Un linguaggio L è accettato da un ϵ-NFA se e solo se L è accettato da un DFA.

DIMOSTRAZIONE (Se) La prova in questa direzione è facile. Supponiamo che esista un DFA D tale che $L = L(D)$. Trasformiamo D in un ϵ-NFA E aggiungendo la transizione $\delta(q, \epsilon) = \emptyset$ per ogni stato q di D. Tecnicamente dobbiamo anche trasformare le transizioni di D sui simboli di input in NFA-transizioni; per esempio $\delta_D(q, a) = p$ diventa $\delta_E(q, a) = \{p\}$. Perciò le transizioni di E e quelle di D sono le stesse, ma in E si stabilisce esplicitamente che non ci sono ϵ-transizioni.

(Solo se) Sia $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ un ϵ-NFA. Applichiamo la costruzione per sottoinsiemi, modificata come sopra, per generare il DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

Dobbiamo dimostrare che $L(D) = L(E)$; a tale scopo proviamo che le funzioni di transizione estese di E e D coincidono. Formalmente, dimostriamo $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ per induzione sulla lunghezza di w.

BASE Se $|w| = 0$, allora $w = \epsilon$. Sappiamo che $\hat{\delta}_E(q_0, \epsilon) = \text{ECLOSE}(q_0)$ e che $q_D = \text{ECLOSE}(q_0)$, perché questa è la definizione dello stato iniziale di D. Infine per un DFA sappiamo che $\hat{\delta}(p, \epsilon) = p$ per ogni stato p; in particolare $\hat{\delta}_D(q_D, \epsilon) = \text{ECLOSE}(q_0)$. Abbiamo così provato che $\hat{\delta}_E(q_0, \epsilon) = \hat{\delta}_D(q_D, \epsilon)$.

INDUZIONE Supponiamo $w = xa$, dove a è l'ultimo simbolo di w, e assumiamo che l'enunciato valga per x, cioè che $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$. Siano questi due insiemi di stati uguali a $\{p_1, p_2, \ldots, p_k\}$.

Per la definizione di $\hat{\delta}$ negli ϵ-NFA, $\hat{\delta}_E(q_0, w)$ si calcola nel modo seguente.

1. Sia $\bigcup_{i=1}^{k} \delta_E(p_i, a)$ uguale a $\{r_1, r_2, \ldots, r_m\}$.

2. Ne segue che $\hat{\delta}_E(q_0, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

Esaminando la costruzione del DFA D per sottoinsiemi, modificata come sopra, notiamo che $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ si ottiene dagli stessi passi (1) e (2) appena illustrati. Perciò $\hat{\delta}_D(q_D, w)$, che è $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$, coincide con $\hat{\delta}_E(q_0, w)$. Abbiamo così provato che $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ e completato la parte induttiva. □

2.5.6 Esercizi

* **Esercizio 2.5.1** Considerate il seguente ϵ-NFA.

	ϵ	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

a) Calcolate l'ϵ-chiusura di ciascuno stato.

b) Elencate tutte le stringhe di lunghezza minore o uguale a tre accettate dall'automa.

c) Trasformate l'automa in un DFA.

Esercizio 2.5.2 Ripetete l'Esercizio 2.5.1 per il seguente ϵ-NFA:

	ϵ	a	b	c
$\rightarrow p$	$\{q, r\}$	$\emptyset$	$\{q\}$	$\{r\}$
q	$\emptyset$	$\{p\}$	$\{r\}$	$\{p, q\}$
$*r$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Esercizio 2.5.3 Definite gli ϵ-NFA per i seguenti linguaggi, sfruttando le ϵ-transizioni per semplificarli.

a) L'insieme delle stringhe composte da zero o più a, seguite da zero o più b, seguite da zero o più c.

! b) L'insieme delle stringhe formate da 01 ripetuto una o più volte, o da 010 ripetuto una o più volte.

! c) L'insieme delle stringhe di 0 e di 1 tali che almeno uno degli ultimi dieci simboli sia 1.

2.6 Riepilogo

✦ *Automi a stati finiti deterministici (DFA)*: un DFA ha un insieme finito di stati e un insieme finito di simboli di input. Uno degli stati è iniziale; zero o più stati sono accettanti. Una funzione di transizione stabilisce come cambia lo stato a fronte di un simbolo di input.

- ✦ *Diagrammi di transizione*: è utile rappresentare gli automi tramite grafi i cui nodi corrispondono agli stati e i cui archi, che denotano le transizioni, sono etichettati da simboli di input. Lo stato iniziale è indicato da una freccia, gli stati accettanti da doppi circoli.

- ✦ *Linguaggio di un automa*: un automa accetta stringhe. Una stringa è accettata se, a partire dallo stato iniziale, le transizioni corrispondenti all'elaborazione in sequenza dei simboli della stringa portano a uno stato accettante. Rispetto a un diagramma di transizione, una stringa è accettata se è l'etichetta di un cammino dallo stato iniziale a uno stato accettante.

- ✦ *Automi a stati finiti non deterministici (NFA)*: gli NFA si distinguono dai DFA perché possono avere un numero arbitrario, anche zero, di transizioni con la stessa etichetta uscenti dallo stesso stato.

- ✦ *Costruzione per sottoinsiemi*: trattando gli insiemi di stati di un NFA come stati di un DFA, si può trasformare un NFA in un DFA che accetta lo stesso linguaggio.

- ✦ *ϵ-transizioni*: si può estendere la nozione di NFA ammettendo transizioni su input vuoto, cioè senza alcun simbolo di input. Un NFA esteso in questo modo si può trasformare in un DFA che accetta lo stesso linguaggio.

- ✦ *Applicazioni alla ricerca in testi*: gli automi a stati finiti non deterministici sono utili per rappresentare macchine (*pattern-matcher*) che ricercano una o più parole-chiave in una collezione di testi. Questi automi si possono simulare direttamente in un programma, oppure si possono prima convertire in un DFA, che viene poi simulato.

2.7 Bibliografia

Di solito si ritiene che lo studio formale dei sistemi a stati finiti prenda le mosse da [2]. In realtà questo lavoro si fondava su un modello di computazione a "reti neuronali" anziché sugli automi a stati finiti come li intendiamo oggi. I DFA usuali furono proposti indipendentemente, in diverse varianti, da [1], [3] e [4]. Gli automi non deterministici e la costruzione per sottoinsiemi provengono da [5].

1. D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.* **257**:3-4 (1954), pp. 161–190 e 275–303.

2. W. S. McCulloch, W. Pitts, "A logical calculus of the ideas immanent in nervious activity," *Bull. Math. Biophysics* **5** (1943), pp. 115–133.

3. G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal* **34**:5 (1955), pp. 1045–1079.

4. E. F. Moore, "Gedanken experiments on sequential machines," in [6], pp. 129–153.

5. M. O. Rabin, D. Scott, "Finite automata and their decision problems," *IBM J. Research and Development* **3**:2 (1959), pp. 115–125.

6. C. E. Shannon, J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956.

Capitolo 3

Espressioni e linguaggi regolari

Apriamo questo capitolo presentando le cosiddette espressioni regolari, un tipo di notazione per definire linguaggi già brevemente esemplificato nel Paragrafo 1.1.2. Le espressioni regolari si possono anche considerare come un linguaggio di programmazione in cui esprimere applicazioni rilevanti, relative per esempio all'ambito delle ricerche in testi o come componenti di un compilatore. Le espressioni regolari sono strettamente legate agli automi a stati finiti non deterministici e possono rappresentare una comoda alternativa alla notazione degli NFA per descrivere componenti software.

In questo capitolo, dopo averle definite, dimostriamo che le espressioni regolari sono in grado di definire tutti e soli i linguaggi regolari, discutiamo il modo in cui esse sono utilizzate in diversi sistemi software, quindi esaminiamo le leggi algebriche che si applicano loro. Nonostante differenze rilevanti, tali leggi somigliano in modo significativo a quelle dell'aritmetica.

3.1 Espressioni regolari

Da una descrizione dei linguaggi fondata su un tipo di macchina, gli automi a stati finiti deterministici e no, volgiamo ora l'attenzione a una descrizione algebrica: le "espressioni regolari". Scopriremo che le espressioni regolari definiscono esattamente gli stessi linguaggi descritti dalle varie forme di automa: i linguaggi regolari. Le espressioni regolari offrono comunque qualcosa in più rispetto agli automi: un modo dichiarativo di esprimere le stringhe da accettare. Perciò le espressioni regolari servono da linguaggio di input per molti sistemi che trattano stringhe. Vediamo alcuni esempi.

1. I comandi di ricerca, come `grep` nei sistemi UNIX, o i comandi equivalenti per la ricerca di stringhe in un browser o in programmi per la composizione di testi. In tutti questi casi si impiega una notazione simile alle espressioni regolari per

descrivere i *pattern* che l'utente vuole cercare in un file. Un sistema di ricerca converte l'espressione regolare in un DFA o in un NFA e simula l'automa sul file da esaminare.

2. I generatori di analizzatori lessicali, come Lex e Flex. Un analizzatore lessicale è quel componente di un compilatore che spezza un programma sorgente in unità logiche dette *token*, composte da uno o più caratteri e aventi un significato preciso. Esempi di *token* sono le "parole chiave" o *keyword* (ad es., `while`), gli identificatori (ad es., una lettera seguita da zero o più lettere o cifre) e certi simboli, come + o <=. Un generatore di analizzatori lessicali accetta descrizioni della forma dei *token*, in sostanza espressioni regolari, e produce un DFA che riconosce la sequenza dei *token* nell'input.

3.1.1 Gli operatori delle espressioni regolari

Le espressioni regolari denotano linguaggi. L'espressione regolare $\mathbf{01^*} + \mathbf{10^*}$ denota per esempio il linguaggio formato da tutte le stringhe composte da uno 0 seguito da qualsiasi numero di 1, oppure da un 1 seguito da qualsiasi numero di 0. Poiché non abbiamo ancora spiegato come interpretare le espressioni regolari, per il momento l'affermazione relativa al linguaggio di tale espressione deve essere accettata sulla parola. Tra breve definiremo tutti i simboli usati nell'espressione, in modo da chiarire perché l'interpretazione proposta è quella corretta. Prima di descrivere la notazione delle espressioni regolari, dobbiamo conoscere le tre operazioni sui linguaggi rappresentate dagli operatori delle espressioni regolari. Tali operazioni sono le seguenti.

1. L'*unione* di due linguaggi L ed M, indicata con $L \cup M$, è l'insieme delle stringhe che sono in L oppure in M, oppure in entrambi. Per esempio se $L = \{001, 10, 111\}$ e $M = \{\epsilon, 001\}$, allora $L \cup M = \{\epsilon, 10, 001, 111\}$.

2. La *concatenazione* dei linguaggi L ed M è l'insieme delle stringhe che si possono formare prendendo una qualunque stringa in L e concatenandola con qualsiasi stringa in M. Abbiamo definito la concatenazione di una coppia di stringhe nel Paragrafo 1.5.2; una stringa è seguita dall'altra per formare il risultato della concatenazione. Indichiamo la concatenazione di linguaggi con un punto oppure senza alcun operatore; l'operatore di concatenazione è spesso chiamato "punto" (*dot*). Per esempio, se $L = \{001, 10, 111\}$ e $M = \{\epsilon, 001\}$, allora $L \cdot M$, o semplicemente LM, è $\{001, 10, 111, 001001, 10001, 111001\}$. Le prime tre stringhe in LM sono le stringhe in L concatenate con ϵ. Dato che ϵ è l'identità per la concatenazione, le stringhe risultanti sono quelle in L. Le ultime tre stringhe in LM sono invece formate prendendo ogni stringa in L e concatenandola con la seconda stringa in M, che è 001. Per esempio 10 da L concatenato con 001 da M dà 10001 per LM.

3. La *chiusura* (o *star* o *chiusura di Kleene*[1]) di un linguaggio L viene indicata con L^* e rappresenta l'insieme delle stringhe che possono essere formate prendendo un numero qualsiasi di stringhe da L, eventualmente con ripetizioni (la stessa stringa può essere selezionata più di una volta), e concatenandole tutte. Per esempio, se $L = \{0, 1\}$, allora L^* consiste di tutte le stringhe di 0 e 1. Se $L = \{0, 11\}$, allora L^* consiste di quelle stringhe di 0 e 1 tali che gli 1 compaiono a coppie, per esempio 011, 11110 e ϵ, ma non 01011 o 101. In termini più formali, L^* è l'unione infinita $\cup_{i \geq 0} L^i$, dove $L^0 = \{\epsilon\}$, $L^1 = L$, e L^i, per $i > 1$ è $LL \cdots L$ (la concatenazione di i copie di L).

Esempio 3.1 Dato che l'idea di chiusura di un linguaggio non è immediatamente comprensibile, ne esaminiamo qualche esempio. In primo luogo, sia $L = \{0, 11\}$. $L^0 = \{\epsilon\}$, indipendentemente da L: la potenza di ordine 0 rappresenta la selezione di zero stringhe da L. Abbiamo poi $L^1 = L$, ovvero la scelta di una stringa di L. Dunque i primi due termini nell'espansione di L^* danno $\{\epsilon, 0, 11\}$.

Consideriamo poi L^2. Prendiamo due stringhe da L, con ripetizioni consentite, per cui ci sono quattro scelte, che danno $L^2 = \{00, 011, 110, 1111\}$. Analogamente L^3 è l'insieme delle stringhe che possono essere formate compiendo tre scelte delle due stringhe in L, cioè

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

Per calcolare L^* dobbiamo computare L^i per ogni i, e prendere l'unione di tutti questi linguaggi. L^i ha 2^i membri. Sebbene ogni L^i sia finito, l'unione degli infiniti termini L^i è generalmente un linguaggio infinito, come nel nostro esempio.

Sia ora L l'insieme di tutte le stringhe di 0. Si noti che L è infinito, a differenza dell'esempio precedente, in cui L era un linguaggio finito. Tuttavia non è difficile scoprire cos'è L^*. $L^0 = \{\epsilon\}$, come sempre, mentre $L^1 = L$. L^2 è l'insieme di stringhe formato prendendo una stringa di 0 e concatenandola con un'altra stringa di 0. Il risultato è ancora una stringa di 0. In effetti ogni stringa di 0 può essere scritta come la concatenazione di due stringhe di 0 (non si dimentichi che ϵ è una "stringa di 0" e può sempre essere una delle due stringhe che vengono concatenate). Dunque $L^2 = L$. Analogamente $L^3 = L$, e così di seguito. Perciò l'unione infinita $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$ è L nel caso particolare che il linguaggio L sia l'insieme di tutte le stringhe di 0.

Come esempio finale, $\emptyset^* = \{\epsilon\}$. Notiamo che $\emptyset^0 = \{\epsilon\}$, mentre $\emptyset^i$ è vuoto per ogni $i \geq 1$, dato che nessuna stringa può essere selezionata dall'insieme vuoto. Effettivamente $\emptyset$ è uno degli unici due linguaggi la cui chiusura *non* è infinita. □

---

[1] La locuzione "chiusura di Kleene" fa riferimento a S. C. Kleene, che ideò la notazione delle espressioni regolari e quest'operatore.

Uso dell'operatore asterisco

Abbiamo incontrato per la prima volta l'operatore asterisco nel Paragrafo 1.5.2, dove l'abbiamo applicato a un alfabeto, per esempio Σ^*. Tale operatore forma tutte le stringhe i cui simboli sono scelti dall'alfabeto Σ. L'operatore di chiusura è essenzialmente lo stesso, sebbene vi sia una sottile differenza di tipo.

Supponiamo che L sia il linguaggio contenente stringhe di lunghezza 1, e che per ogni simbolo a in Σ esista una stringa a in L. Allora, sebbene L e Σ abbiano lo stesso "aspetto", sono di tipo diverso: L è un insieme di stringhe, mentre Σ è un insieme di simboli. D'altra parte L^* indica lo stesso linguaggio di Σ^*.

3.1.2 Costruzione di espressioni regolari

Qualsiasi algebra si forma a partire da alcune espressioni elementari, di solito costanti e variabili. Le algebre permettono poi di costruire ulteriori espressioni applicando un determinato insieme di operatori alle espressioni elementari e alle espressioni costruite in precedenza. Abitualmente è necessario anche un metodo per raggruppare gli operatori con i loro operandi, ad esempio attraverso le parentesi. Per esemplificare, l'algebra aritmetica comunemente nota parte da costanti come gli interi e i numeri reali, più le variabili, e costruisce espressioni più complesse con operatori aritmetici come $+$ e $\times$.

L'algebra delle espressioni regolari segue questo schema, usando costanti e variabili che indicano linguaggi e operatori per le tre operazioni del Paragrafo 3.1.1: unione, punto e star. Possiamo descrivere le espressioni regolari in maniera ricorsiva, come segue. In questa definizione, non solo descriviamo che cosa sono le espressioni regolari lecite, ma per ogni espressione regolare E descriviamo il linguaggio che rappresenta, indicandolo con $L(E)$.

BASE La base consiste di tre parti.

1. Le costanti ϵ e $\emptyset$ sono espressioni regolari, indicanti rispettivamente i linguaggi $\{\epsilon\}$ e $\emptyset$. In altre parole $L(\epsilon) = \{\epsilon\}$, e $L(\emptyset) = \emptyset$.

2. Se a è un simbolo qualsiasi, allora $\mathbf{a}$ è un'espressione regolare. Tale espressione denota il linguaggio $\{a\}$. Cioè $L(\mathbf{a}) = \{a\}$. Si noti che si usa il grassetto per indicare un'espressione corrispondente a un simbolo. La corrispondenza, per esempio che $\mathbf{a}$ si riferisce ad a, dovrebbe essere evidente.

3. Una variabile, generalmente una lettera maiuscola e in corsivo, come L, rappresentante un linguaggio arbitrario.

INDUZIONE Il passo induttivo consta di quattro parti, una per ognuno dei tre operatori e una per l'introduzione delle parentesi.

1. Se E ed F sono espressioni regolari, allora $E + F$ è un'espressione regolare che indica l'unione di $L(E)$ e $L(F)$. In altri termini $L(E + F) = L(E) \cup L(F)$.

2. Se E ed F sono espressioni regolari, allora EF è un'espressione regolare che indica la concatenazione di $L(E)$ e $L(F)$. Ossia $L(EF) = L(E)L(F)$. Si osservi che il punto può essere usato facoltativamente per indicare l'operatore di concatenazione, visto come operatore su linguaggi oppure come operatore in un'espressione regolare. Per esempio $\mathbf{0{\cdot}1}$ è un'espressione regolare che ha lo stesso significato di $\mathbf{01}$ e rappresenta il linguaggio $\{01\}$. Tuttavia si eviterà di usare il punto come concatenazione nelle espressioni regolari.[2]

3. Se E è un'espressione regolare, allora E^* è un'espressione regolare che indica la chiusura di $L(E)$. Cioè $L(E^*) = \big(L(E)\big)^*$.

4. Se E è un'espressione regolare, allora anche (E), un E tra parentesi, è un'espressione regolare, e indica lo stesso linguaggio di E. In termini formali $L\big((E)\big) = L(E)$.

Esempio 3.2 Scriviamo un'espressione regolare per l'insieme delle stringhe che consistono di 0 e 1 alternati. Dapprima sviluppiamo un'espressione regolare per il linguaggio che consta di una singola stringa 01. Possiamo poi usare l'operatore asterisco per ottenere un'espressione per tutte le stringhe della forma $0101 \cdots 01$.

La regola di base per le espressioni regolari ci dice che $\mathbf{0}$ e $\mathbf{1}$ sono espressioni che indicano rispettivamente i linguaggi $\{0\}$ e $\{1\}$. Se concateniamo le due espressioni, otteniamo un'espressione regolare per il linguaggio $\{01\}$; questa espressione è $\mathbf{01}$. Come regola generale, se vogliamo un'espressione regolare per il linguaggio che consiste solamente della stringa w, usiamo w stesso come espressione regolare. Si noti che nell'espressione regolare i simboli di w saranno scritti di norma in grassetto, ma il cambiamento di carattere serve solo a distinguere le espressioni dalle stringhe e non dovrebbe essere considerato significativo.

Per ottenere tutte le stringhe di zero o più occorrenze di 01, usiamo ora l'espressione regolare $(\mathbf{01})^*$. Si osservi che mettiamo prima tra parentesi $\mathbf{01}$ per evitare di confonderci con l'espressione $\mathbf{01}^*$, il cui linguaggio è formato da tutte le stringhe consistenti di uno 0 e di un qualunque numero di 1. La ragione di questa interpretazione verrà chiarita nel Paragrafo 3.1.3, ma si può anticipare brevemente che lo star ha la precedenza sul punto e perciò l'argomento dello star viene selezionato prima di compiere concatenazioni.

---

[2] In realtà le espressioni regolari in UNIX impiegano il punto per uno scopo completamente diverso, ossia per rappresentare qualunque carattere ASCII.

Le espressioni e i loro linguaggi

Per essere precisi, un'espressione regolare E è appunto un'espressione, non un linguaggio. Quando vogliamo riferirci al linguaggio denotato da E, dovremmo usare $L(E)$. Tuttavia è consuetudine fare riferimento a E quando effettivamente si intende $L(E)$. Ricorreremo a tale convenzione finché sarà chiaro che si tratta di un linguaggio e non di un'espressione regolare.

Tuttavia $L\big((\mathbf{01})^*\big)$ non è esattamente il linguaggio che vogliamo, in quanto include solo le stringhe di 0 e 1 alternati che cominciano per 0 e finiscono per 1, mentre dobbiamo anche considerare la possibilità che ci sia un 1 all'inizio e uno 0 alla fine. Un modo può essere quello di costruire altre tre espressioni regolari che trattino le altre tre possibilità. In altre parole $(\mathbf{10})^*$ rappresenta le stringhe alternate che cominciano per 1 e finiscono per 0, $\mathbf{0}(\mathbf{10})^*$ può essere usato per stringhe che cominciano e finiscono per 0, e $\mathbf{1}(\mathbf{01})^*$ si impiega per stringhe che cominciano e finiscono per 1. L'espressione regolare completa è

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Si faccia attenzione all'uso dell'operatore $+$ per effettuare l'unione dei quattro linguaggi, che nel complesso danno tutte le stringhe con alternanze di 0 e 1.

Esiste però un altro modo, che produce un'espressione regolare diversa e più concisa. Ricominciamo dall'espressione $(\mathbf{01})^*$. Se facciamo una concatenazione a sinistra con l'espressione $\epsilon + \mathbf{1}$, possiamo aggiungere un 1 facoltativo all'inizio. Analogamente aggiungiamo uno 0 facoltativo alla fine con l'espressione $\epsilon + \mathbf{0}$. Per esempio, ricorrendo alla definizione dell'operatore $+$:

$$L(\epsilon + \mathbf{1}) = L(\epsilon) \cup L(\mathbf{1}) = \{\epsilon\} \cup \{1\} = \{\epsilon, 1\}$$

Se concateniamo questo linguaggio con un qualsiasi altro linguaggio L, la scelta ϵ dà tutte le stringhe in L, mentre la scelta 1 dà $1w$ per ogni stringa w in L. Perciò un'altra espressione per l'insieme di stringhe che alternano 0 e 1 è:

$$(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0})$$

Si noti che, per garantire che gli operatori siano raggruppati opportunamente, sono necessarie le parentesi prima e dopo ogni espressione aggiunta. □

3.1.3 Precedenza degli operatori delle espressioni regolari

Come altre algebre, gli operatori delle espressioni regolari hanno un ordine di precedenza, il che significa che gli operatori sono associati ai loro operandi in un ordine particolare. La

nozione di precedenza ci è familiare dalle espressioni aritmetiche ordinarie. Per esempio sappiamo che $xy + z$ raggruppa il prodotto xy prima della somma, dunque è equivalente all'espressione tra parentesi $(xy) + z$ e non all'espressione $x(y + z)$. In aritmetica raggruppiamo similmente da sinistra due operatori uguali, così $x - y - z$ è equivalente a $(x - y) - z$ e non a $x - (y - z)$. Per le espressioni regolari l'ordine di precedenza degli operatori è il seguente.

1. L'operatore star ha il livello più alto di priorità. Vale a dire, si applica solamente alla sequenza più breve di simboli alla sua sinistra che sia un'espressione regolare ben formata.

2. Il successivo, nell'ordine di precedenza, è la concatenazione o operatore "punto". Dopo aver raggruppato tutti gli star con i loro operandi, passiamo agli operatori di concatenazione con i loro operandi. In altre parole vengono raggruppate insieme tutte le espressioni che sono *giustapposte* (adiacenti, senza alcun operatore interposto). Dato che la concatenazione è un operatore associativo, non ha importanza in quale ordine raggruppiamo le concatenazioni consecutive, sebbene, dovendo scegliere, sia consigliabile farlo a partire da sinistra. Per esempio **012** viene raggruppato così: (**01**)**2**.

3. Per ultime vengono raggruppate le unioni (gli operatori +) con i loro operandi. Dato che anche l'unione è associativa, non è importante in quale ordine vengono raggruppate unioni consecutive, anche se si assumerà che il raggruppamento avvenga a partire da sinistra.

Ovviamente a volte non si desidera che in un'espressione regolare il raggruppamento segua l'ordine di precedenza degli operatori. In tal caso siamo liberi di usare le parentesi per raggruppare gli operandi come ci pare più opportuno. Oltre a ciò non è scorretto neppure mettere fra parentesi gli operatori che si vogliono raggruppare, anche se il raggruppamento desiderato è conforme alle regole di precedenza.

Esempio 3.3 L'espressione $\mathbf{01}^* + \mathbf{1}$ corrisponde a $\big(\mathbf{0}(\mathbf{1}^*)\big) + \mathbf{1}$. Per primo viene raggruppato l'operatore star. Poiché il simbolo **1** immediatamente alla sua sinistra è un'espressione regolare lecita, esso da solo è l'operando dello star. Raggruppiamo poi la concatenazione tra **0** e $(\mathbf{1}^*)$, che produce l'espressione $\big(\mathbf{0}(\mathbf{1}^*)\big)$. Infine l'operatore unione connette questa espressione e quella alla sua destra, che è **1**.

Si noti che il linguaggio dell'espressione data, raggruppata secondo le regole di precedenza, è la stringa 1 più tutte le stringhe formate da uno 0 seguito da un qualunque numero di 1 (incluso nessuno). Se avessimo scelto di raggruppare il punto prima dello star, avremmo potuto usare le parentesi così: $(\mathbf{01})^* + \mathbf{1}$. Il linguaggio di quest'espressione è la stringa 1 e tutte le stringhe che ripetono 01, zero oppure più volte. Se volessimo raggruppare prima l'unione, potremmo metterla tra parentesi per produrre l'espressione

$\mathbf{0}(\mathbf{1}^* + \mathbf{1})$. Il linguaggio di quest'espressione è l'insieme di stringhe formate da uno 0 seguito da un numero arbitrario di 1. □

3.1.4 Esercizi

Esercizio 3.1.1 Scrivete le espressioni regolari per i seguenti linguaggi.

* a) L'insieme delle stringhe sull'alfabeto $\{a, b, c\}$ che contengano almeno una a e almeno una b.

b) L'insieme delle stringhe di 0 e 1 il cui decimo simbolo a partire da destra sia 1.

c) L'insieme delle stringhe di 0 e 1 con al massimo una coppia di 1 consecutivi.

! **Esercizio 3.1.2** Scrivete le espressioni regolari per i seguenti linguaggi.

* a) L'insieme di tutte le stringhe di 0 e 1 tali che ogni coppia di 0 adiacenti compaia prima di qualunque coppia di 1 adiacenti.

b) L'insieme delle stringhe di 0 e 1 il cui numero di 0 sia divisibile per cinque.

!! **Esercizio 3.1.3** Scrivete le espressioni regolari per i seguenti linguaggi.

a) L'insieme di tutte le stringhe di 0 e 1 che non contengano 101 come sottostringa.

b) L'insieme di tutte le stringhe con un numero uguale di 0 e 1 tali che in ogni prefisso la differenza tra il numero di 0 e il numero di 1 sia minore di 2.

c) L'insieme delle stringhe di 0 e 1 il cui numero di 0 sia divisibile per cinque e il numero di 1 sia pari.

! **Esercizio 3.1.4** Descrivete in italiano i linguaggi delle seguenti espressioni regolari:

* a) $(\mathbf{1} + \epsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$

b) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0} + \mathbf{1})^*$

c) $(\mathbf{0} + \mathbf{10})^*\mathbf{1}^*$.

*! **Esercizio 3.1.5** Nell'Esempio 3.1 abbiamo sottolineato che $\emptyset$ è uno dei linguaggi la cui chiusura è finita. Qual è l'altro?

3.2 Automi a stati finiti ed espressioni regolari

Descrivere un linguaggio per mezzo di un'espressione regolare è fondamentalmente diverso dal farlo per mezzo di un automa finito; eppure le due notazioni rappresentano di fatto la stessa classe di linguaggi, quelli che abbiamo denominato "regolari". Abbiamo già dimostrato che gli automi a stati finiti deterministici e i due tipi di automa a stati finiti non deterministici, con e senza ϵ-transizioni, accettano la stessa classe di linguaggi. Per provare che le espressioni regolari definiscono la stessa classe, dobbiamo dimostrare due cose.

1. Ogni linguaggio definito da uno di questi automi è definito anche da un'espressione regolare. Per questa dimostrazione possiamo assumere che il linguaggio sia accettato da un DFA.

2. Ogni linguaggio definito da un'espressione regolare è definito da uno di questi automi. Per questa parte della dimostrazione è più semplice mostrare che esiste un NFA con ϵ-transizioni che accetta lo stesso linguaggio.

La Figura 3.1 illustra tutte le equivalenze che abbiamo dimostrato o che dimostreremo. Un arco dalla classe X alla classe Y significa che ogni linguaggio definito dalla classe X è definito anche dalla classe Y. Poiché il grafo è fortemente connesso (cioè possiamo andare da ognuno dei quattro nodi a qualunque altro nodo) concludiamo che le quattro classi in realtà coincidono.

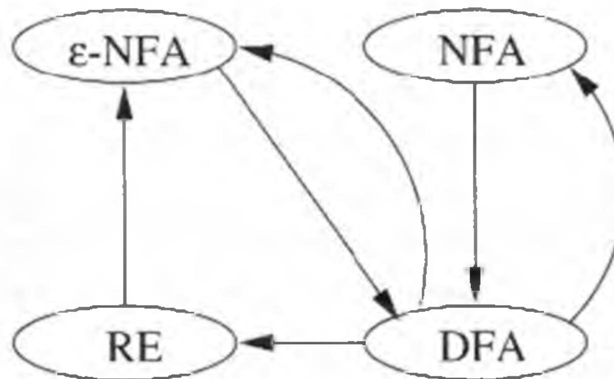


Figura 3.1 Piano per la dimostrazione dell'equivalenza di quattro diverse notazioni per i linguaggi regolari.

3.2.1 Dai DFA alle espressioni regolari

La costruzione di un'espressione regolare che definisca il linguaggio di un DFA dato è sorprendentemente intricata. Detto a grandi linee, si devono formare espressioni che descrivano insiemi di stringhe che etichettano certi cammini nel diagramma di transizione

del DFA, considerando però solo cammini che attraversano un sottoinsieme limitato degli stati. In una definizione induttiva di queste espressioni si comincia da quelle più semplici, che descrivono i cammini che non possono passare attraverso *nessuno* stato (vale a dire, nodi singoli oppure archi singoli), e per via induttiva si costruiscono le espressioni che fanno passare i cammini attraverso insiemi di stati via via più grandi. Alla fine i cammini possono passare attraverso qualunque stato, in modo che le espressioni generate rappresentino tutti i cammini possibili. Questi concetti sono presenti nella dimostrazione del teorema seguente.

Teorema 3.4 Se $L = L(A)$ per un DFA A, allora esiste un'espressione regolare R tale che $L = L(R)$.

DIMOSTRAZIONE Supponiamo che gli stati di A siano $\{1, 2, \ldots, n\}$ per un intero n. Comunque siano definiti gli stati di A, ce ne saranno n per un n finito, e rinominandoli ci si può riferire agli stati in questo modo, come se fossero i primi n interi positivi. Il nostro primo e più difficile compito consiste nel costruire una collezione di espressioni regolari che descrivano insiemi via via più ampi di cammini nel diagramma di transizione di A.

Usiamo $R_{ij}^{(k)}$ come nome dell'espressione regolare il cui linguaggio è l'insieme di stringhe w tale che w sia l'etichetta di un cammino dallo stato i allo stato j in A, e il cammino non abbia alcun nodo intermedio il cui numero sia maggiore di k. Si noti che i punti di partenza e di arrivo del cammino non sono intermedi; dunque non si richiede che i e j siano inferiori o uguali a k.

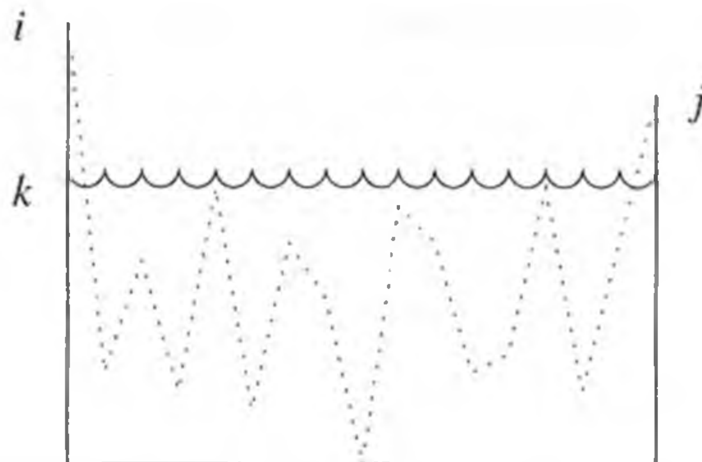


Figura 3.2 Un cammino la cui etichetta è nel linguaggio dell'espressione regolare $R_{ij}^{(k)}$.

La Figura 3.2 suggerisce il requisito sui cammini rappresentati da $R_{ij}^{(k)}$. La dimensione verticale rappresenta lo stato, da 1 in basso fino a n in alto; la dimensione orizzontale rappresenta il percorso lungo il cammino. Si noti che nel diagramma sia i sia j sono più grandi di k, anche se uno di loro o entrambi potrebbero essere uguali a k oppure minori.

Si osservi anche che il cammino passa due volte attraverso il nodo k, ma non attraversa mai uno stato più alto di k, eccetto che agli estremi.

Per costruire l'espressione $R_{ij}^{(k)}$ usiamo la seguente definizione induttiva, a partire da $k = 0$ per raggiungere infine $k = n$. Si noti che quando $k = n$, poiché non esistono stati più grandi di n, non c'è alcuna restrizione sui cammini rappresentati.

BASE La base è $k = 0$. Dato che tutti gli stati sono numerati da 1 in su, la restrizione comporta che il cammino non abbia nessuno stato intermedio. Ci sono solo due tipi di cammino che soddisfano tale condizione:

1. un arco dal nodo (stato) i al nodo j
2. un cammino di lunghezza 0 che consiste solamente di un nodo i.

Se $i \neq j$, allora è possibile solo il caso (1). Dobbiamo esaminare il DFA A e trovare i simboli di input a tali che esista una transizione dallo stato i allo stato j su a.

a) Se non esiste un tale simbolo a, allora $R_{ij}^{(0)} = \emptyset$.

b) Se esiste esattamente un tale simbolo a, allora $R_{ij}^{(0)} = \mathbf{a}$.

c) Se esistono simboli $a_1, a_2, \ldots, a_k$ che etichettano archi dallo stato i allo stato j, allora $R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

Se invece $i = j$, allora i cammini leciti sono il cammino di lunghezza 0 e tutti i cicli da i a se stesso. Poiché non ha simboli lungo il suo percorso, il cammino di lunghezza 0 è rappresentato dall'espressione regolare ϵ. Aggiungiamo perciò ϵ alle varie espressioni stabilite dal punto (a) fino al punto (c). Così nel caso (a) [nessun simbolo a] l'espressione diventa ϵ, nel caso (b) [un simbolo a] l'espressione diventa $\epsilon + \mathbf{a}$, e nel caso (c) [simboli multipli] l'espressione diventa $\epsilon + \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

INDUZIONE Supponiamo che esista un cammino dallo stato i allo stato j che non attraversa nessuno stato superiore a k. Bisogna prendere in considerazione due casi.

1. Il cammino non passa per lo stato k. In questo caso l'etichetta del cammino appartiene al linguaggio di $R_{ij}^{(k-1)}$.
2. Il cammino passa per lo stato k almeno una volta. Allora possiamo scomporlo in segmenti, come suggerito dalla Figura 3.3. Il primo segmento va dallo stato i allo stato k senza passare per k, l'ultimo va da k a j senza passare per k, e tutti i segmenti nel mezzo vanno da k a se stesso senza passare per k. Si osservi che se il cammino passa attraverso k solo una volta, allora non esistono segmenti mediani, ma solo un cammino da i a k e uno da k a j. L'insieme di etichette per tutti i cammini di

questo tipo è rappresentato dall'espressione regolare $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$. In altre parole la prima espressione rappresenta la parte del cammino che giunge allo stato k la prima volta, la seconda rappresenta la porzione che va da k a se stesso per zero, una o più volte, e la terza espressione rappresenta la parte del cammino che lascia k per l'ultima volta e va nello stato j.

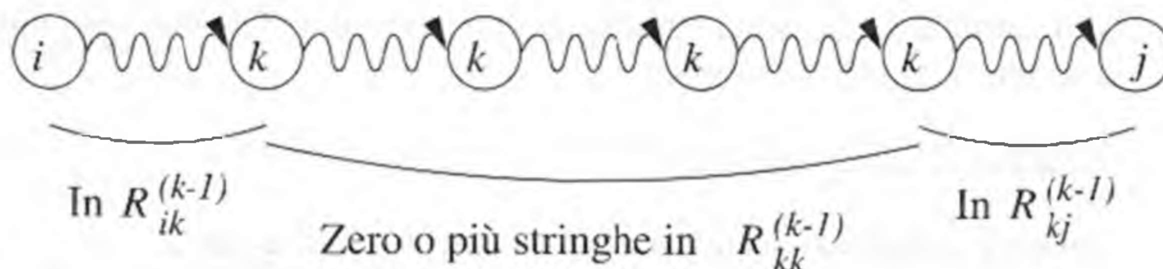


Figura 3.3 Un cammino da i a j può essere scomposto in segmenti determinati dai passaggi per lo stato k.

Quando combiniamo le espressioni per i cammini dei due tipi descritti sopra, otteniamo l'espressione

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$$

per le etichette di tutti i cammini dallo stato i allo stato j che non passano per stati più alti di k. Se costruiamo queste espressioni in ordine rispetto al valore dell'apice (k), allora, dato che ogni $R_{ij}^{(k)}$ dipende solamente dalle espressioni con un apice più piccolo, tutte le espressioni sono disponibili quando ne abbiamo bisogno.

Alla fine abbiamo $R_{ij}^{(n)}$ per ogni i e j. Possiamo assumere che lo stato 1 sia lo stato iniziale, mentre gli stati accettanti possono essere qualunque insieme di stati. L'espressione regolare per il linguaggio dell'automa è allora la somma (unione) di tutte le espressioni $R_{1j}^{(n)}$ tali che lo stato j sia uno stato accettante. □

Esempio 3.5 Convertiamo il DFA della Figura 3.4 in un'espressione regolare. Questo DFA accetta tutte le stringhe che contengono almeno uno 0. Per capirne la ragione, si noti che l'automa va dallo stato iniziale 1 allo stato accettante 2 non appena vede l'input 0. L'automa rimane allora nello stato 2 su tutte le sequenze di input.

Ecco le espressioni di base per la costruzione del Teorema 3.4.

$R_{11}^{(0)}$	$\epsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\epsilon + \mathbf{0} + \mathbf{1})$

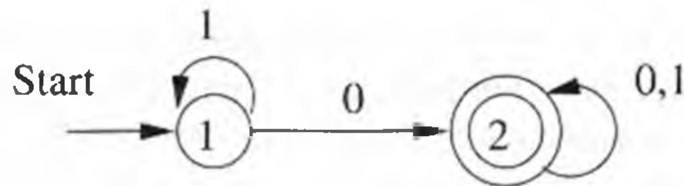


Figura 3.4 Un DFA che accetta tutte le stringhe con almeno uno 0.

Per esempio $R_{11}^{(0)}$ ha il termine ϵ perché gli stati iniziale e finale sono gli stessi: lo stato 1. Ha il termine **1** perché esiste un arco dallo stato 1 allo stato 1 su input 1. Come ulteriore esempio, $R_{12}^{(0)}$ è **0** perché esiste un arco etichettato 0 dallo stato 1 allo stato 2. Non ci sono termini ϵ in quanto gli stati iniziale e finale sono diversi. Come terzo esempio, abbiamo che $R_{21}^{(0)} = \emptyset$ perché non esiste nessun arco dallo stato 2 allo stato 1.

Ora dobbiamo passare alla parte induttiva, ossia dobbiamo costruire espressioni più complesse che prima prendono in considerazione cammini che passano attraverso lo stato 1 e poi cammini che passano attraverso gli stati 1 e 2, cioè cammini arbitrari. La regola per il calcolo delle espressioni $R_{ij}^{(1)}$ è un caso della regola generale data nella parte induttiva del Teorema 3.4:

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

Il prospetto della Figura 3.5 riporta le espressioni computate per sostituzione diretta nella formula sopra, seguite da espressioni semplificate di cui possiamo dimostrare, attraverso un ragionamento *ad hoc*, che rappresentano lo stesso linguaggio dell'espressione più complessa.

	Per sostituzione diretta	Semplificata
$R_{11}^{(1)}$	$\epsilon + \mathbf{1} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*\mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\emptyset + \emptyset(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon + \mathbf{0} + \mathbf{1} + \emptyset(\epsilon + \mathbf{1})^*\mathbf{0}$	$\epsilon + \mathbf{0} + \mathbf{1}$

Figura 3.5 Espressioni regolari per i cammini che passano solo attraverso lo stato 1.

Per esempio consideriamo $R_{12}^{(1)}$. La sua espressione è $R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$, che si ottiene dalla (3.1) sostituendo $i = 1$ e $j = 2$.

Per comprendere la semplificazione, si osservi il principio generale che se R è una qualunque espressione regolare, allora $(\epsilon + R)^* = R^*$. La giustificazione è che ambedue i membri dell'equazione descrivono il linguaggio costituito da qualsiasi concatenazione

di zero o più stringhe di $L(R)$. Nel nostro caso abbiamo $(\epsilon + \mathbf{1})^* = \mathbf{1}^*$; ambedue le espressioni denotano un numero arbitrario di 1. Inoltre $(\epsilon + \mathbf{1})\mathbf{1}^* = \mathbf{1}^*$. Ancora una volta si può osservare che entrambe le espressioni denotano "qualunque numero di 1". Dunque l'espressione originale $R_{12}^{(1)}$ è equivalente a $\mathbf{0} + \mathbf{1}^*\mathbf{0}$. Tale espressione denota il linguaggio che contiene la stringa 0 e tutte le stringhe formate da uno 0 preceduto da un numero arbitrario di 1. Questo linguaggio è indicato anche dall'espressione più semplice $\mathbf{1}^*\mathbf{0}$.

La semplificazione di $R_{11}^{(1)}$ è simile alla semplificazione di $R_{12}^{(1)}$ appena considerata. La semplificazione di $R_{21}^{(1)}$ e $R_{22}^{(1)}$ dipende da due regole relative a $\emptyset$. Per qualunque espressione regolare R, vale quanto segue.

1. $\emptyset R = R\emptyset = \emptyset$. In altre parole, $\emptyset$ è un *annichilatore* per la concatenazione; è esso stesso il risultato della concatenazione con una qualsiasi espressione a sinistra o a destra. Tale regola ha un senso perché, se vogliamo che una stringa sia nel risultato di una concatenazione, dobbiamo trovare stringhe da entrambi gli argomenti della concatenazione. Se uno degli argomenti è $\emptyset$, è impossibile trovare una stringa per quell'argomento.

2. $\emptyset + R = R + \emptyset = R$. Ossia $\emptyset$ è l'identità per l'unione; quando compare in un'unione, il risultato è l'altra espressione.

Di conseguenza un'espressione come $\emptyset(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$ può essere sostituita da $\emptyset$. Le ultime due semplificazioni dovrebbero essere chiare.

Calcoliamo ora l'espressione $R_{ij}^{(2)}$. La regola induttiva applicata con $k = 2$ dà:

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^* R_{2j}^{(1)} \tag{3.2}$$

Se sostituiamo nella (3.2) le espressioni semplificate della Figura 3.5, otteniamo le espressioni della Figura 3.6. Tale figura mostra anche le semplificazioni che seguono gli stessi principi descritti per la Figura 3.5.

	Per sostituzione diretta	Semplificata
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\mathbf{0}(\epsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\epsilon + \mathbf{0} + \mathbf{1})^*(\epsilon + \mathbf{0} + \mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + \mathbf{0} + \mathbf{1})(\epsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon + \mathbf{0} + \mathbf{1} + (\epsilon + \mathbf{0} + \mathbf{1})(\epsilon + \mathbf{0} + \mathbf{1})^*(\epsilon + \mathbf{0} + \mathbf{1})$	$(\mathbf{0} + \mathbf{1})^*$

Figura 3.6 Espressioni regolari per i cammini che possono passare attraverso qualunque stato.

L'espressione regolare finale equivalente all'automa della Figura 3.4 è costruita prendendo l'unione di tutte le espressioni in cui il primo stato è lo stato iniziale e il secondo stato è quello accettante. In questo esempio, con 1 come stato iniziale e 2 come unico stato accettante, abbiamo bisogno solamente dell'espressione $R_{12}^{(2)}$. Tale espressione è $\mathbf{1}^*\mathbf{0}(\mathbf{0}+\mathbf{1})^*$, ed è di facile interpretazione. Il suo linguaggio consiste di tutte le stringhe che cominciano con zero o più 1, poi presentano uno 0, e poi qualunque stringa di 0 e 1. In altri termini, il linguaggio è "tutte le stringhe di 0 e 1 con almeno uno 0". □

3.2.2 Conversione di DFA in espressioni regolari per eliminazione di stati

Il metodo per convertire un DFA in un'espressione regolare, che abbiamo visto nel Paragrafo 3.2.1, funziona sempre. Come il lettore avrà notato, esso non dipende dal fatto che il DFA è deterministico, e si può applicare anche a un NFA o a un ϵ-NFA. Tuttavia la costruzione dell'espressione regolare è dispendiosa. Non solo dobbiamo costruire circa n^3 espressioni per un automa di n stati, ma, se le espressioni non vengono semplificate, la lunghezza dell'espressione può crescere in media di un fattore 4 in ognuno degli n passi induttivi. Le espressioni stesse possono perciò raggiungere un numero di simboli dell'ordine di 4^n.

Esiste una soluzione simile che evita di svolgere un doppio lavoro in determinati punti. Per esempio tutte le espressioni con apice $(k+1)$ nella costruzione del Teorema 3.4 usano la stessa sottoespressione $(R_{kk}^{(k)})^*$; la scrittura di tale espressione viene dunque ripetuta n^2 volte.

La costruzione che mostreremo ora comporta l'eliminazione di stati. Quando eliminiamo uno stato s, tutti i cammini che passano per s non esistono più nell'automa. Per non cambiare il linguaggio dell'automa dobbiamo includere, su un arco che va direttamente da q a p, le etichette di cammini che vanno da q a p attraverso s. Poiché l'etichetta di un tale arco può ora recare stringhe anziché simboli, e può esserci anche un numero infinito di tali stringhe, per specificare l'etichetta non possiamo semplicemente elencarle. Fortunatamente abbiamo a disposizione un modo facile e finito di rappresentare tutte queste stringhe: usare un'espressione regolare.

Siamo dunque portati a considerare automi che hanno come etichette espressioni regolari. Il linguaggio dell'automa è l'unione, su tutti i cammini dallo stato iniziale a uno stato accettante, dei linguaggi formati concatenando i linguaggi delle espressioni regolari lungo un cammino. Si noti che questa regola è coerente con la definizione di linguaggio associato a un qualunque tipo di automa trattato fin qui. Ogni simbolo a, oppure ϵ se è consentito, può essere pensato come un'espressione regolare il cui linguaggio è una stringa singola, $\{a\}$ oppure $\{\epsilon\}$. Si può considerare quest'osservazione come la base della procedura di eliminazione di stati che passeremo ora a descrivere.

La Figura 3.7 illustra uno stato s generico che sta per essere eliminato. Supponiamo

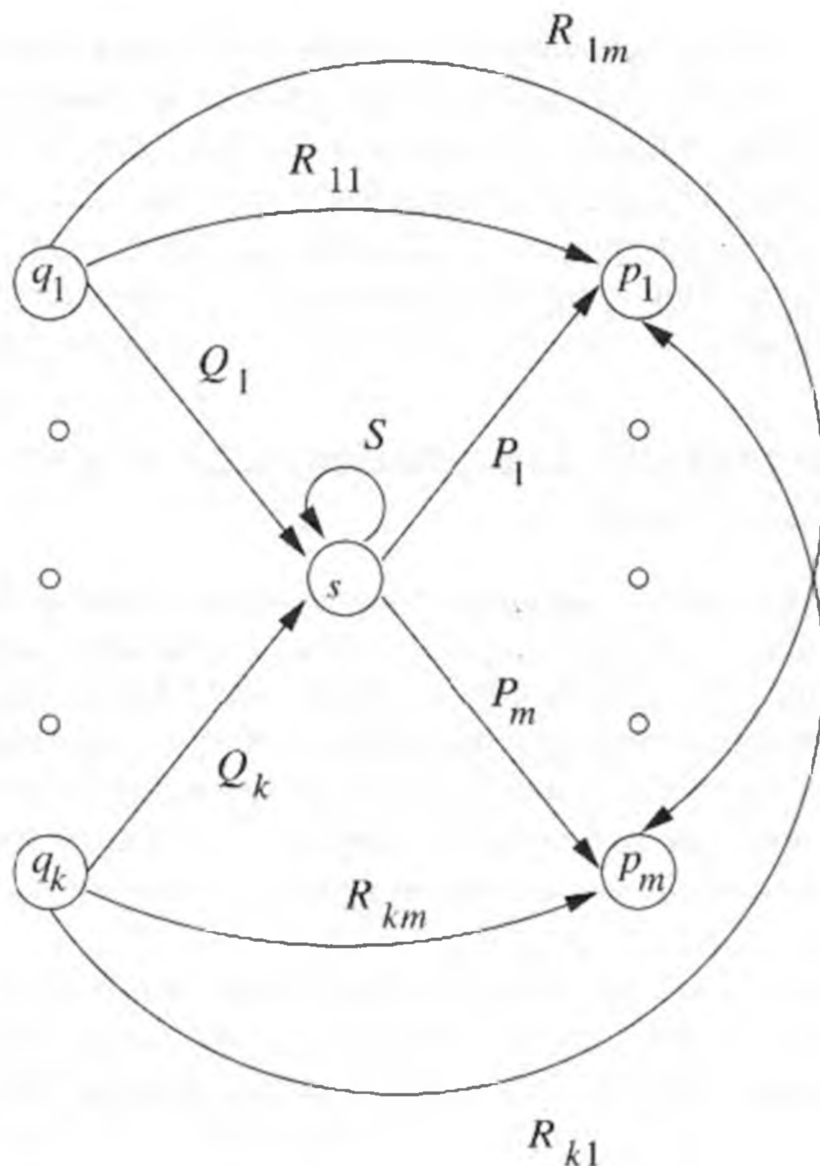


Figura 3.7 Uno stato s che sta per essere eliminato.

che nell'automa di cui s è uno stato, gli stati $q_1, q_2, \ldots, q_k$ siano predecessori di s e gli stati $p_1, p_2, \ldots, p_m$ siano successori di s. È possibile che alcuni dei q siano anche fra i p, ma supponiamo che s non si trovi tra i q o i p, anche se esiste un ciclo da s a se stesso, come suggerito nella Figura 3.7. Mostriamo anche un'espressione regolare su ciascun arco da uno dei q a s; l'espressione Q_i etichetta l'arco uscente da q_i. In modo analogo mostriamo l'espressione regolare P_i che etichetta l'arco da s a p_i, per ogni i. Mostriamo un ciclo su s con etichetta S. Infine esiste una espressione regolare R_{ij} sull'arco da q_i a p_j, per tutti i valori i e j. Si osservi che alcuni di questi archi possono non esistere nell'automa. In questo caso possiamo pensare che l'espressione su tali archi sia $\emptyset$.

La Figura 3.8 illustra che cosa succede quando si elimina lo stato s: tutti gli archi che lo toccano sono cancellati. Per compensare, introduciamo per ogni predecessore q_i e ogni successore p_j di s un'espressione regolare, che rappresenta tutti i cammini che partono

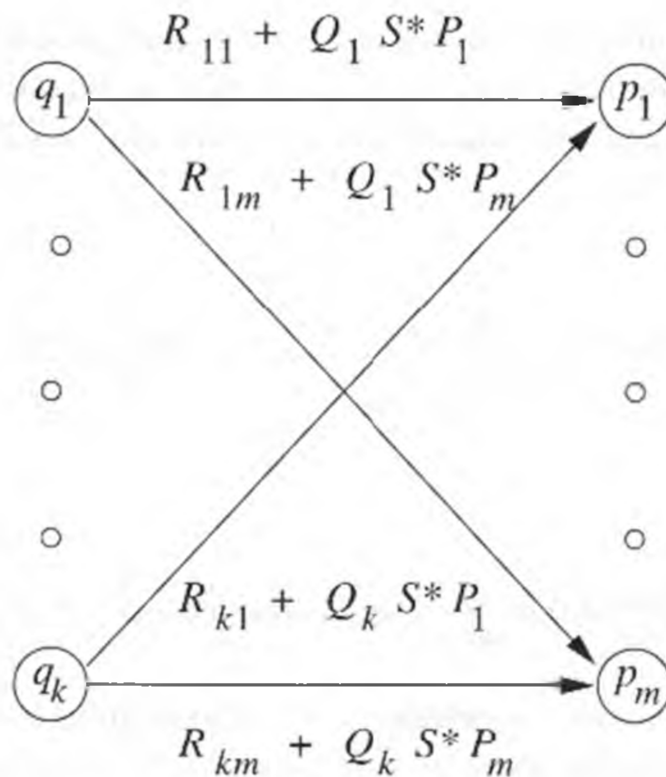


Figura 3.8 Risultato dell'eliminazione di s dalla Figura 3.7.

da q_i, vanno a s, eventualmente compiono un ciclo attorno a s zero o più volte, e infine vanno a p_j. L'espressione per questi cammini è $Q_i S^* P_j$. Tale espressione viene aggiunta (con l'operatore di unione) all'arco da q_i a p_j. Se non esiste alcun arco $q_i \rightarrow p_j$, ne introduciamo prima uno con l'espressione regolare $\emptyset$.

Descriviamo la strategia per costruire un'espressione regolare a partire da un automa a stati finiti.

1. Per ogni stato accettante q applichiamo il processo di riduzione menzionato sopra per produrre un automa equivalente con le etichette sugli archi costituite da espressioni regolari. Eliminiamo tutti gli stati eccetto q e lo stato iniziale q_0.

2. Se $q \neq q_0$, allora il risultato è un automa a due stati simile a quello della Figura 3.9. L'espressione regolare per le stringhe accettate si può descrivere in vari modi, uno dei quali è $(R + SU^*T)^*SU^*$, che possiamo spiegare così: possiamo andare dallo stato iniziale verso se stesso per un numero arbitrario di volte seguendo una sequenza di cammini le cui etichette sono in $L(R)$ oppure in $L(SU^*T)$. L'espressione SU^*T rappresenta i cammini che vanno nello stato accettante lungo un cammino in $L(S)$, eventualmente ritornano più volte allo stato accettante attraverso una sequenza di cammini con etichette in $L(U)$, e poi tornano allo stato iniziale con un cammino la cui etichetta è in $L(T)$. A questo punto dobbiamo passare allo stato

accettante senza più tornare allo stato iniziale, seguendo un cammino con un'etichetta in $L(S)$. Una volta nello stato accettante, possiamo ritornarci tutte le volte che vogliamo seguendo un cammino la cui etichetta è in $L(U)$.

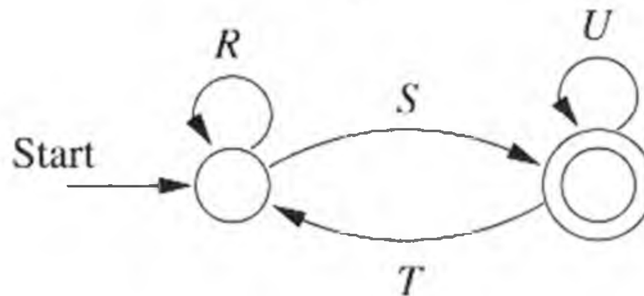


Figura 3.9 Un generico automa a due stati.

3. Se lo stato iniziale è anche accettante, dobbiamo eliminare tutti gli stati dell'automa originale, eccetto quello iniziale. Così facendo otteniamo un automa a uno stato, simile a quello della Figura 3.10. L'espressione regolare che indica le stringhe che tale automa accetta è R^*.

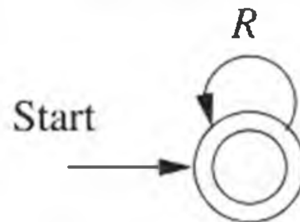


Figura 3.10 Un generico automa a uno stato.

4. L'espressione regolare desiderata è la somma (unione) di tutte le espressioni derivate dagli automi ridotti per ogni stato accettante, in virtù delle regole (2) e (3).

Esempio 3.6 Consideriamo l'NFA della Figura 3.11, che accetta tutte le stringhe di 0 e 1 che hanno 1 in penultima o terzultima posizione. Il primo passo consiste nel convertirlo in un automa con espressioni regolari come etichette. Dato che non è stato eliminato alcuno stato, dobbiamo soltanto sostituire le etichette "0,1" con l'espressione regolare equivalente $\mathbf{0} + \mathbf{1}$. Il risultato è illustrato nella Figura 3.12.

Per prima cosa eliminiamo lo stato B. Poiché tale stato non è lo stato accettante né lo stato iniziale, non si troverà in nessuno degli automi ridotti. Eliminandolo subito, prima di sviluppare i due automi ridotti che corrispondono agli stati accettanti, possiamo perciò risparmiarci del lavoro.

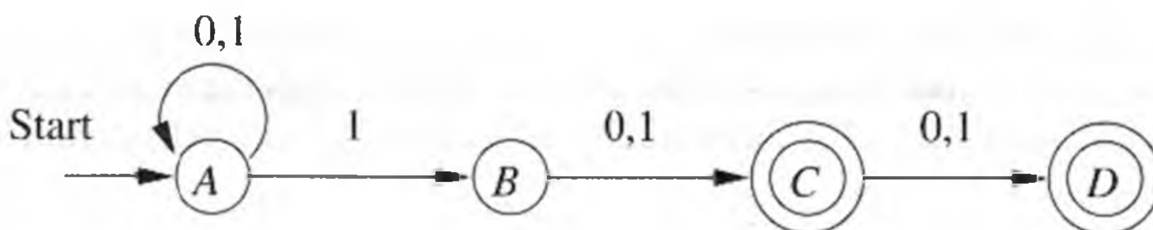


Figura 3.11 Un NFA che accetta stringhe che hanno un 1 in penultima o terzultima posizione.

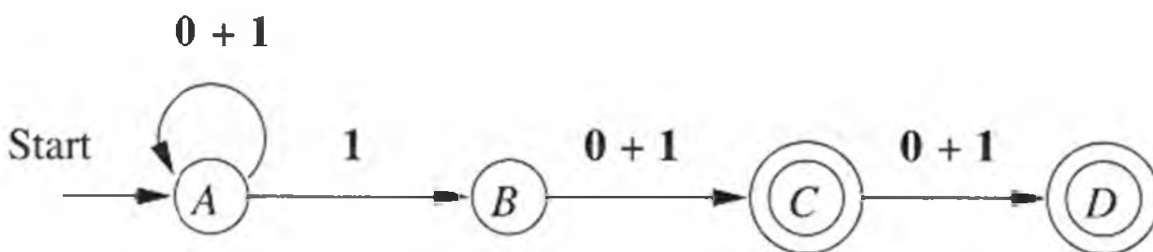


Figura 3.12 L'automa della Figura 3.11 con espressioni regolari come etichette.

Lo stato B ha un predecessore, A, e un successore, C. Nei termini delle espressioni regolari del diagramma nella Figura 3.7: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0} + \mathbf{1}$, $R_{11} = \emptyset$ (dato che l'arco da A a C non esiste) e $S = \emptyset$ (poiché non c'è nessun ciclo sullo stato B). Ne risulta che l'espressione sul nuovo arco da A a C è $\emptyset + \mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1})$.

Per semplificare, eliminiamo prima l'iniziale $\emptyset$, che può essere ignorato in un'unione. L'espressione diventa quindi $\mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1})$. Si noti che l'espressione regolare $\emptyset^*$ è equivalente all'espressione regolare ϵ, dato che

$$L(\emptyset^*) = \{\epsilon\} \cup L(\emptyset) \cup L(\emptyset)L(\emptyset) \cup \cdots$$

Dal momento che tutti i termini eccetto il primo sono vuoti, abbiamo $L(\emptyset^*) = \{\epsilon\}$, il che è lo stesso di $L(\epsilon)$. Perciò $\mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1})$ è equivalente a $\mathbf{1}(\mathbf{0} + \mathbf{1})$, che è l'espressione che usiamo per l'arco $A \to C$ nella Figura 3.13.

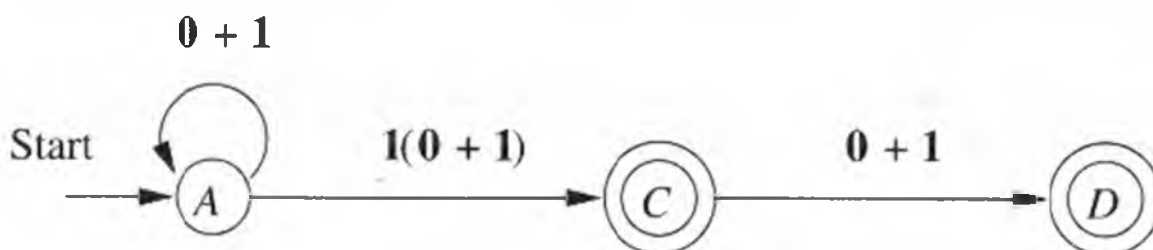


Figura 3.13 Eliminazione dello stato B.

A questo punto dobbiamo prendere due strade distinte, eliminando gli stati C e D in riduzioni separate. I meccanismi per eliminare lo stato C sono simili a quelli impiegati sopra per eliminare lo stato B e l'automa che ne deriva è illustrato nella Figura 3.14.

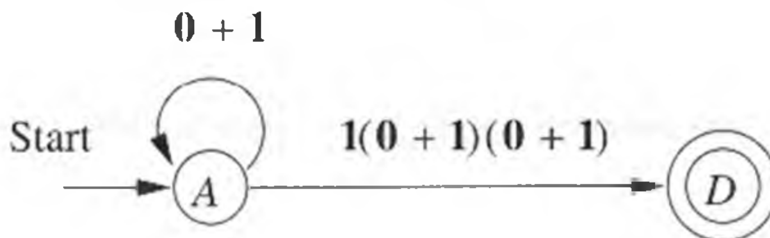


Figura 3.14 Un automa a due stati con gli stati A e D.

Nei termini del generico automa a due stati della Figura 3.9, le espressioni regolari per la Figura 3.14 sono: $R = \mathbf{0} + \mathbf{1}$, $S = \mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$, $T = \emptyset$ e $U = \emptyset$. L'espressione U^* può essere sostituita da ϵ, ossia eliminata, in una concatenazione. La spiegazione è che, come discusso sopra, $\emptyset^* = \epsilon$. Inoltre l'espressione SU^*T è equivalente a $\emptyset$, poiché T, uno dei termini della concatenazione, è $\emptyset$. In questo caso l'espressione generica $(R + SU^*T)^*SU^*$ si semplifica dunque in R^*S, cioè $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$. In termini informali il linguaggio di quest'espressione è una qualsiasi stringa che finisca per 1, seguita da due simboli che sono ciascuno 0 oppure 1. Tale linguaggio è una porzione delle stringhe accettate dall'automa della Figura 3.11: le stringhe la cui terzultima posizione è occupata da un 1.

Dobbiamo ora ricominciare dalla Figura 3.13 ed eliminare lo stato D anziché C. Dato che D non ha successori, l'osservazione della Figura 3.7 rivela che non ci saranno cambiamenti negli archi e che l'arco da C a D viene eliminato insieme allo stato D. L'automa a due stati che ne deriva è rappresentato nella Figura 3.15.

L'automa è molto simile a quello della Figura 3.14; solo l'etichetta sull'arco dallo stato iniziale allo stato accettante è diversa. Possiamo dunque applicare la regola per gli automi a due stati e semplificare l'espressione per ottenere $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$. Questa espressione rappresenta l'altro tipo di stringa accettata dall'automa: le stringhe con un 1 in penultima posizione.

Non rimane altro che compiere la somma delle due espressioni per ricavarne l'espressione relativa all'intero automa della Figura 3.11. Questa è

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

□

L'ordine nell'eliminazione degli stati

Come abbiamo osservato nell'Esempio 3.6, uno stato viene eliminato in tutti gli automi derivati quando non è né lo stato iniziale né uno stato accettante. Perciò uno dei vantaggi del processo di eliminazione di stati, paragonato alla generazione meccanica di espressioni regolari descritta nel Paragrafo 3.2.1, è che possiamo cominciare eliminando definitivamente tutti gli stati che non sono né iniziali né accettanti. Lo sforzo di riduzione è duplicato solo quando si deve eliminare qualche stato accettante.

Anche in questo caso possiamo combinare una parte dello sforzo. Per esempio, se ci sono tre stati accettanti p, q ed r, è possibile eliminare p, e poi eliminare separatamente q ed r, producendo rispettivamente gli automi per gli stati accettanti r e q. Poi ricominciamo con tutti e tre gli stati accettanti ed eliminiamo sia q sia r ottenendo l'automa per p.

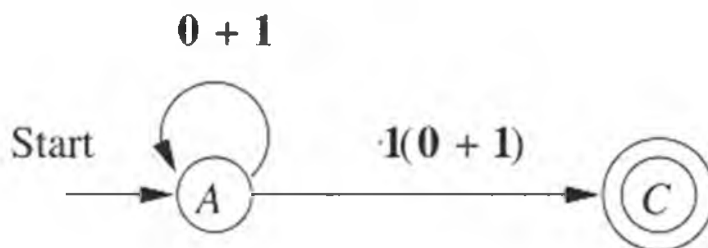


Figura 3.15 Automa a due stati risultante dall'eliminazione di D.

3.2.3 Conversione di espressioni regolari in automi

Completeremo ora il piano della Figura 3.1 mostrando che ogni linguaggio L che sia uguale a $L(R)$ per un'espressione regolare R è anche uguale a $L(E)$ per un ϵ-NFA E. La dimostrazione è un'induzione strutturale sull'espressione R. Cominciamo costruendo gli automi per le espressioni di base: singoli simboli, ϵ e $\emptyset$. Mostriamo poi come combinare questi automi in automi più grandi che accettano l'unione, la concatenazione o la chiusura dei linguaggi accettati da automi più piccoli.

Tutti gli automi che costruiamo sono ϵ-NFA con un unico stato accettante.

Teorema 3.7 Ogni linguaggio definito da un'espressione regolare è definito anche da un automa a stati finiti.

DIMOSTRAZIONE Supponiamo che $L = L(R)$ per un'espressione regolare R. Mostriamo che $L = L(E)$ per un ϵ-NFA E con:

1. esattamente uno stato accettante
2. nessun arco entrante nello stato iniziale
3. nessun arco uscente dallo stato accettante.

La dimostrazione si svolge per induzione strutturale su R, seguendo la definizione ricorsiva delle espressioni regolari data nel Paragrafo 3.1.2.

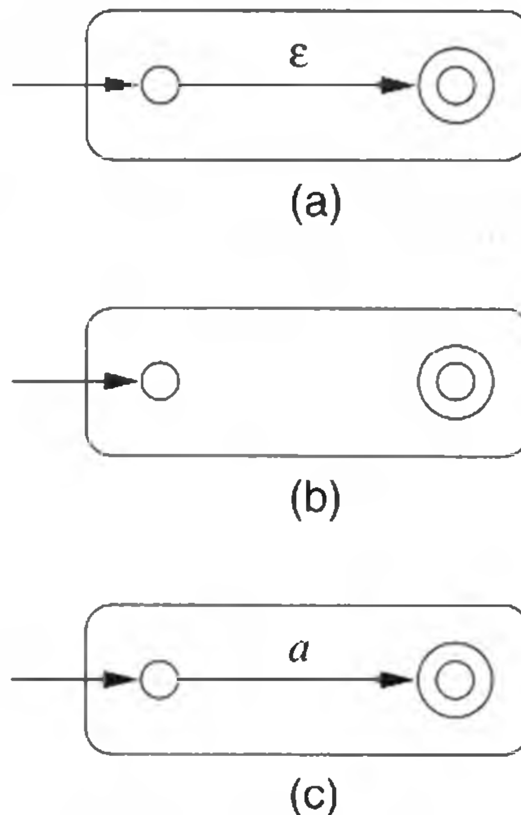


Figura 3.16 La base della costruzione di un automa da un'espressione regolare.

BASE La base consta di tre parti, illustrate dalla Figura 3.16. Nella parte (a) vediamo come trattare l'espressione ϵ. Il linguaggio dell'automa è chiaramente $\{\epsilon\}$, dato che l'unico cammino dallo stato iniziale verso uno stato accettante è etichettato ϵ. La parte (b) mostra la costruzione per $\emptyset$. Chiaramente non esistono cammini dallo stato iniziale allo stato accettante; dunque il linguaggio di quest'automa è $\emptyset$. Infine la parte (c) descrive l'automa per l'espressione regolare **a**. Il linguaggio di quest'automa consiste palesemente nell'unica stringa a che è anche $L(\mathbf{a})$. È facile verificare che tutti questi automi soddisfano le condizioni (1), (2) e (3) dell'ipotesi induttiva.

INDUZIONE Le tre parti dell'induzione sono rappresentate nella Figura 3.17. Supponiamo che l'enunciato del teorema sia vero per le sottoespressioni immediate di una data espressione regolare; vale a dire, i linguaggi di tali sottoespressioni sono anche i linguaggi di ϵ-NFA con un solo stato accettante. Ci sono quattro casi.

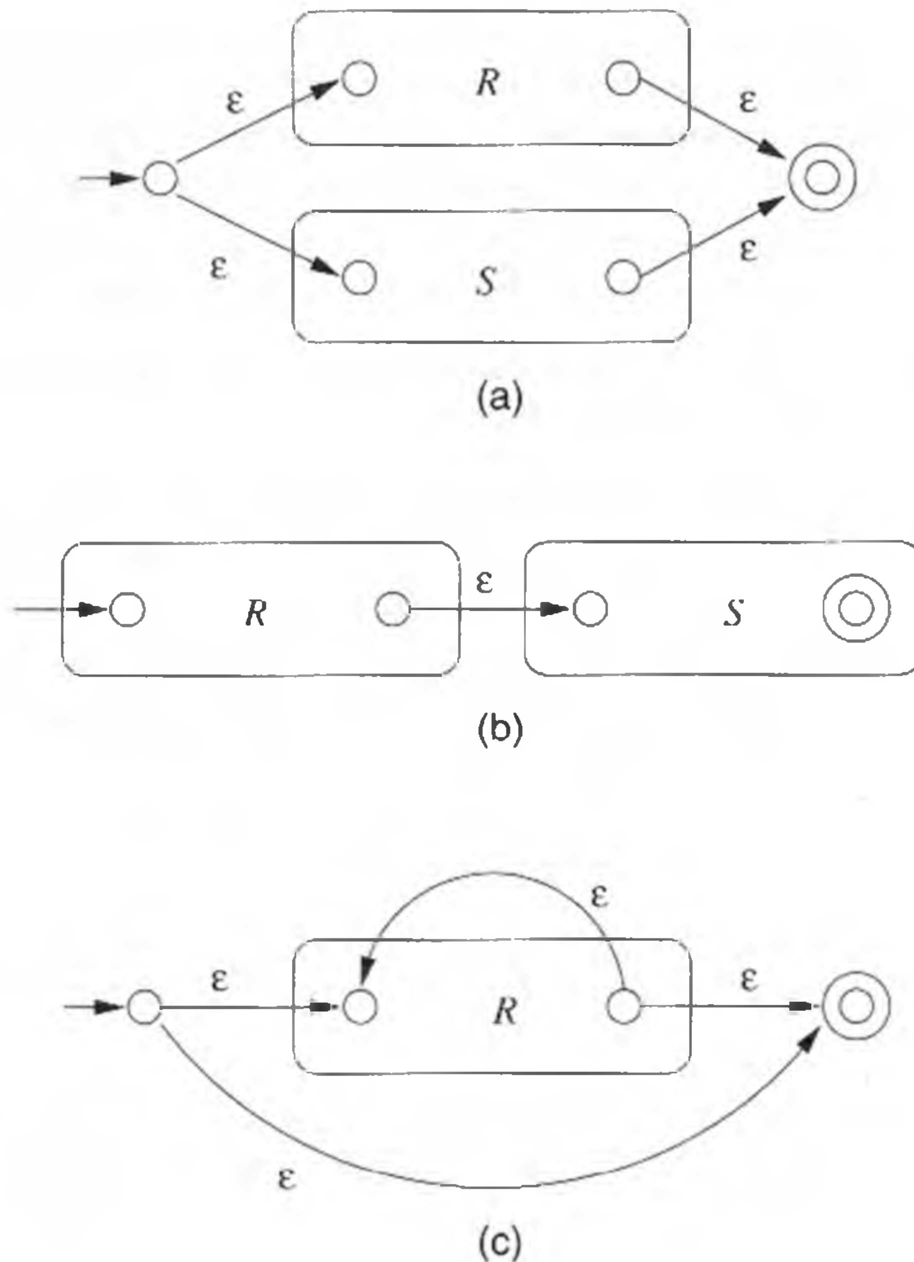


Figura 3.17 Il passo induttivo nella costruzione dall'espressione regolare all'ϵ-NFA.

1. L'espressione è $R + S$ per due espressioni più piccole R ed S. Qui si usa l'automa della Figura 3.17(a). In altre parole, partendo dal nuovo stato iniziale, possiamo andare nello stato iniziale dell'automa per R oppure in quello dell'automa per S. Raggiungiamo poi lo stato accettante di uno di questi automi seguendo un cammino etichettato da una stringa che si trova rispettivamente in $L(R)$ oppure $L(S)$. Una volta raggiunto lo stato accettante dell'automa per R o S, possiamo seguire uno degli archi ϵ verso lo stato accettante del nuovo automa. Il linguaggio dell'automa della Figura 3.17(a) è perciò $L(R) \cup L(S)$.

2. L'espressione è RS per due espressioni più piccole R ed S. L'automa per la concatenazione è rappresentato dalla Figura 3.17(b). Si noti che lo stato iniziale del

primo automa diventa il nuovo stato iniziale, e lo stato accettante del secondo automa diventa lo stato accettante dell'automa complessivo. L'idea è che un cammino dallo stato iniziale a quello accettante deve attraversare prima l'automa per R seguendo un cammino etichettato da una stringa in $L(R)$, e poi quello per S seguendo un cammino etichettato da una stringa in $L(S)$. Quindi i cammini nell'automa della Figura 3.17(b) sono tutti e soli quelli etichettati dalle stringhe in $L(R)L(S)$.

3. L'espressione è R^* per un'espressione più piccola R. Usiamo allora l'automa della Figura 3.17(c), che offre due tipi di percorso.

   (a) Direttamente dallo stato iniziale allo stato accettante lungo un cammino etichettato ϵ. Tale cammino fa accettare ϵ, che si trova in $L(R^*)$ a prescindere da R.

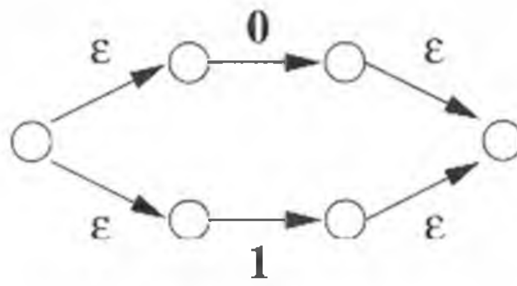
   (b) Verso lo stato iniziale dell'automa per R, attraverso tale automa una o più volte, e poi verso lo stato accettante. Questo insieme di cammini ci permette di accettare stringhe in $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$, e così via, coprendo così tutte le stringhe in $L(R^*)$, eccetto eventualmente ϵ, già coperta dall'arco diretto verso lo stato accettante menzionato in (3a).

4. L'espressione è (R) per un'espressione più piccola R. Dato che le parentesi non cambiano il linguaggio definito dall'espressione, l'automa per R serve anche come automa per (R).

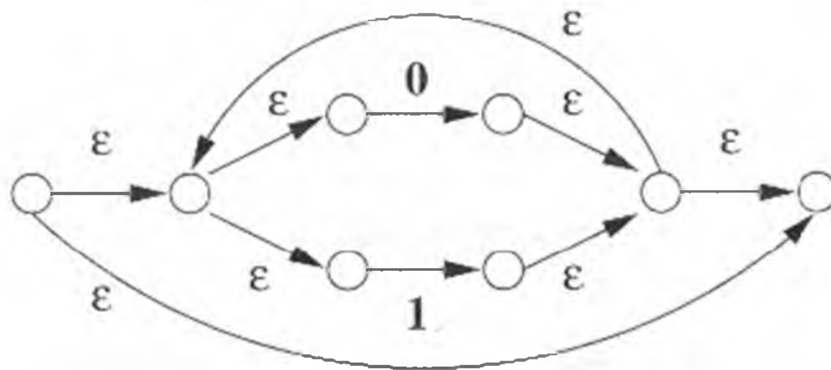
Si osservi infine che gli automi costruiti soddisfano le tre condizioni date nell'ipotesi induttiva: uno stato accettante, nessun arco entrante nello stato iniziale oppure uscente dallo stato accettante. □

Esempio 3.8 Convertiamo l'espressione regolare $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$ in un ϵ-NFA. Il primo passo è costruire un automa per $\mathbf{0}+\mathbf{1}$. Usiamo due automi costruiti secondo la Figura 3.16(c), uno con etichetta **0** sull'arco e uno con etichetta **1**. Questi due automi vengono poi combinati usando la costruzione per l'unione della Figura 3.17(a). Il risultato è rappresentato nella Figura 3.18(a).

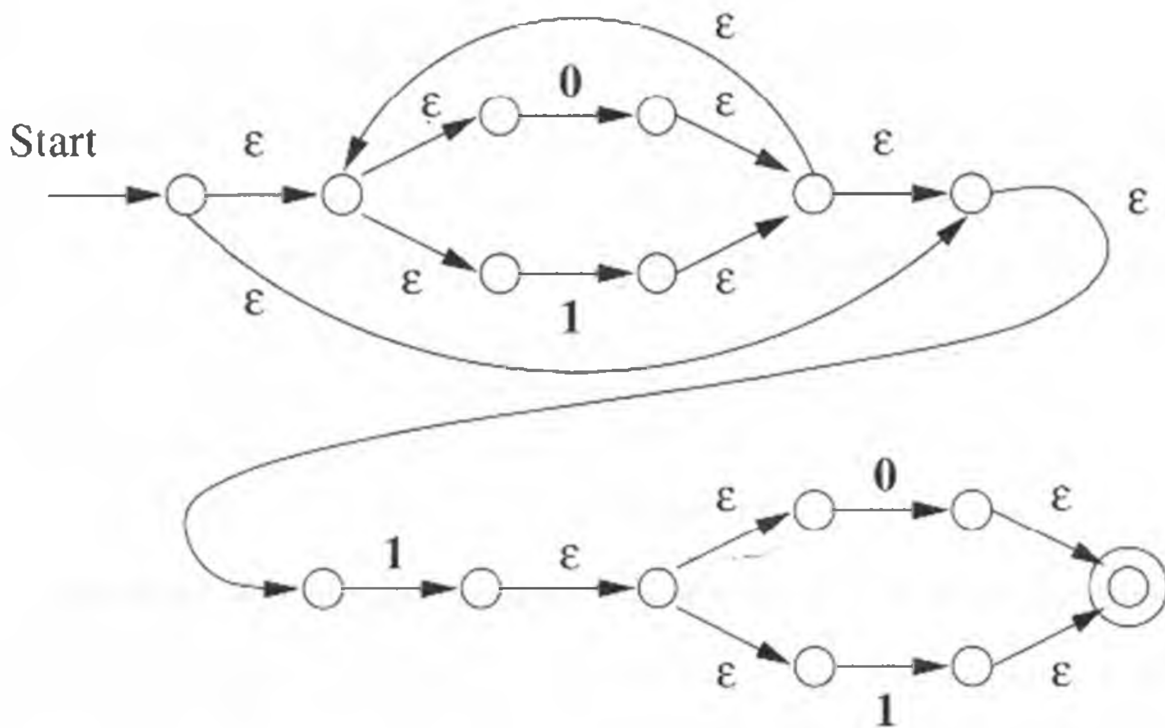
In un secondo passo applichiamo alla Figura 3.18(a) la costruzione per la chiusura della Figura 3.17(c). Il risultato è illustrato dalla Figura 3.18(b). Gli ultimi due passi comportano l'applicazione della costruzione per la concatenazione della Figura 3.17(b). In primo luogo connettiamo l'automa della Figura 3.18(b) all'automa destinato ad accettare solamente la stringa 1. Tale automa è un'altra applicazione della costruzione di base della Figura 3.16(c) con etichetta **1** sull'arco. Si noti che dobbiamo creare un *nuovo* automa per riconoscere 1; non possiamo usare l'automa che riconosce 1, già parte della Figura 3.18(a). Il terzo automa nella concatenazione è di nuovo un automa per $\mathbf{0}+\mathbf{1}$. Ancora una volta dobbiamo creare una copia dell'automa della Figura 3.18(a); non possiamo usare la copia che è diventata parte della Figura 3.18(b). L'automa completo è



(a)



(b)



(c)

Figura 3.18 Gli automi costruiti per l'Esempio 3.8.

rappresentato nella Figura 3.18(c). Osserviamo che questo ϵ-NFA, una volta rimosse le ϵ-transizioni, somiglia all'automa molto più semplice della Figura 3.15, che accetta anche le stringhe con un 1 in penultima posizione. □

3.2.4 Esercizi

Esercizio 3.2.1 Ecco la tabella di transizione per un DFA:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

* a) Scrivete tutte le espressioni regolari $R_{ij}^{(0)}$. *Nota*: si consideri lo stato q_i come se fosse lo stato numero i.

* b) Scrivete tutte le espressioni regolari $R_{ij}^{(1)}$. Semplificate le espressioni il più possibile.

c) Scrivete tutte le espressioni regolari $R_{ij}^{(2)}$. Semplificate le espressioni il più possibile.

d) Scrivete un'espressione regolare per il linguaggio dell'automa.

* e) Costruite il diagramma di transizione per il DFA e scrivete un'espressione regolare per il suo linguaggio eliminando lo stato q_2.

Esercizio 3.2.2 Ripetete l'Esercizio 3.2.1 per il seguente DFA:

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Le soluzioni alle parti (a), (b) ed (e) *non* sono disponibili per quest'esercizio.

Esercizio 3.2.3 Convertite il seguente DFA in un'espressione regolare usando la tecnica di eliminazione di stati del Paragrafo 3.2.2.

	0	1
$\rightarrow *p$	s	p
q	p	s
r	r	q
s	q	r

Esercizio 3.2.4 Convertite le seguenti espressioni regolari in NFA con ϵ-transizioni.

* a) $\mathbf{01}^*$.

b) $(\mathbf{0} + \mathbf{1})\mathbf{01}$.

c) $\mathbf{00}(\mathbf{0} + \mathbf{1})^*$.

Esercizio 3.2.5 Eliminate le ϵ-transizioni dagli NFA dell'Esercizio 3.2.4. Nelle pagine Web del libro potete trovare una soluzione della parte (a).

! **Esercizio 3.2.6** Sia $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ un ϵ-NFA tale che non esiste nessuna transizione entrante in q_0 e nessuna transizione uscente da q_f. Descrivete il linguaggio accettato da ognuna delle seguenti modificazioni di A nei termini di $L = L(A)$.

* a) L'automa costruito da A aggiungendo una ϵ-transizione da q_f a q_0.

* b) L'automa costruito da A aggiungendo una ϵ-transizione da q_0 verso ogni stato raggiungibile da q_0 (lungo un cammino le cui etichette possono includere sia i simboli di Σ sia ϵ).

c) L'automa costruito da A aggiungendo una ϵ-transizione verso q_f da ogni stato che può raggiungere q_f lungo un qualche cammino.

d) L'automa costruito da A svolgendo sia il punto (b) sia il punto (c).

!! **Esercizio 3.2.7** Possiamo semplificare le costruzioni del Teorema 3.7, che trasformano un'espressione regolare in un ϵ-NFA. Presentiamo tre soluzioni.

1. Per l'operatore di unione, anziché creare un nuovo stato iniziale e un nuovo stato accettante, combinate i due stati iniziali in un unico stato con tutte le transizioni di entrambi gli stati iniziali. In modo simile, unite i due stati accettanti e dirigete verso lo stato composto le transizioni dirette a ognuno dei due.

2. Per l'operatore di concatenazione è possibile combinare lo stato accettante del primo automa con lo stato iniziale del secondo.

3. Per l'operatore di chiusura si possono semplicemente aggiungere le ϵ-transizioni dallo stato accettante allo stato iniziale, e viceversa.

Ognuna di queste semplificazioni produce di per sé una costruzione corretta: l'ϵ-NFA risultante per una qualunque espressione regolare accetta il linguaggio dell'espressione. Quali sottoinsiemi di cambiamenti (1), (2) e (3) possono essere svolti insieme nella costruzione, ottenendo comunque un automa corretto per ogni espressione regolare?

*!! **Esercizio 3.2.8** Scrivete un algoritmo che prenda un DFA A e computi il numero delle stringhe di lunghezza n (per un dato n, non correlato con il numero di stati di A) accettate da A. L'algoritmo dovrebbe essere polinomiale sia in n sia nel numero degli stati di A. *Suggerimento*: usate la tecnica seguita nella costruzione del Teorema 3.4.

3.3 Applicazioni delle espressioni regolari

Un'espressione regolare che rappresenta ciò che si vuole riconoscere è lo strumento più adatto per le applicazioni che cercano *pattern* in un testo. Le espressioni regolari vengono poi tradotte, dietro le quinte, in automi deterministici o non deterministici, che vengono simulati per produrre un programma di riconoscimento di *pattern* nel testo. In questo paragrafo tratteremo due importanti classi di applicazioni basate sulle espressioni regolari: gli analizzatori lessicali e la ricerca testuale.

3.3.1 Le espressioni regolari in UNIX

Prima di esaminare le applicazioni, introduciamo la notazione di UNIX per le espressioni regolari estese. Tale notazione offre alcune possibilità supplementari. In realtà le estensioni di UNIX includono caratteristiche, in particolare la possibilità di nominare e fare riferimento a stringhe precedenti conformi a un *pattern*, che permettono il riconoscimento di linguaggi non regolari. Qui non si considerano tali aspetti; ci limitiamo a introdurre alcune scorciatoie utili a esprimere concisamente espressioni regolari complesse.

La prima estensione riguarda il fatto che la maggior parte delle applicazioni impiega l'insieme dei caratteri ASCII. Nei nostri esempi abbiamo usato perlopiù un alfabeto limitato, come $\{0, 1\}$. L'esistenza di due soli simboli ci ha permesso di scrivere espressioni concise come $\mathbf{0} + \mathbf{1}$ per indicare "qualunque carattere". Se ci fossero invece, per esempio, 128 caratteri, la stessa espressione richiederebbe un elenco completo, e sarebbe poco pratica da scrivere. Le espressioni regolari di UNIX permettono dunque di esprimere "classi di caratteri" per la rappresentazione il più possibile concisa di insiemi di caratteri. Le regole per tali classi sono le seguenti.

- Il simbolo `.` (*dot*) sta per "qualunque carattere".

- La sequenza `[`$a_1 a_2 \cdots a_k$`]` sta per l'espressione regolare

$$a_1 + a_2 + \cdots + a_k$$

  Questa notazione fa risparmiare la metà dei caratteri, in quanto non è necessario scrivere i segni +. Per esempio i quattro caratteri usati negli operatori di confronto del C possono essere espressi da `[<>=!]`.

- Possiamo inserire un intervallo della forma x-y tra le parentesi quadre per indicare tutti i caratteri da x a y nella sequenza ASCII. Poiché i codici delle cifre sono progressivi, come quelli delle lettere maiuscole e minuscole, è possibile esprimere molte classi rilevanti di caratteri in poche battute. Per esempio le cifre si possono esprimere con `[0-9]`, le lettere maiuscole con `[A-Z]`, e l'insieme di tutte le lettere e cifre con `[A-Za-z0-9]`. Se vogliamo includere un segno meno nell'elenco dei caratteri possiamo collocarlo al primo o all'ultimo posto, in modo da non confonderne l'uso per la formazione di una serie di caratteri. Per esempio l'insieme delle cifre, più il punto, il più e il meno, utilizzati per formare numeri decimali con segno, si rappresenta con `[-+.0-9]`. Le parentesi quadre o altri caratteri che hanno significati speciali nelle espressioni regolari di UNIX si possono rappresentare come caratteri facendoli precedere da una barra obliqua rovesciata (`\`).

- Esistono notazioni speciali per molte delle più comuni classi di caratteri. Per esempio:

  a) `[:digit:]` è l'insieme delle dieci cifre, esattamente come `[0-9]`[3]

  b) `[:alpha:]` sta per ogni carattere alfabetico, come `[A-Za-z]`

  c) `[:alnum:]` sta per le cifre e le lettere (caratteri alfabetici e numerici), come `[A-Za-z0-9]`.

Ci sono altri operatori, usati nelle espressioni regolari di UNIX, che non abbiamo ancora incontrato. Nessuno di questi operatori estende la classe dei linguaggi esprimibili, ma talvolta consente di descrivere più facilmente ciò che si vuole.

1. L'operatore | si usa al posto di + per indicare l'unione.

2. L'operatore ? significa "zero oppure uno". Perciò in UNIX $R?$ è lo stesso di $\epsilon + R$ nella notazione delle espressioni regolari usata in questo libro.

3. L'operatore + significa "uno o più di uno". Dunque in UNIX $R+$ è l'abbreviazione di RR^* nella nostra notazione.

4. L'operatore $\{n\}$ significa "n copie". Dunque in UNIX $R\{5\}$ è l'abbreviazione di $RRRRR$.

Le espressioni regolari di UNIX consentono l'uso di parentesi per raggruppare sottoespressioni, come per le espressioni regolari descritte nel Paragrafo 3.1.2, e si applicano loro le stesse regole di precedenza degli operatori (con ?, + e $\{n\}$ trattati come * per quanto riguarda la precedenza). L'operatore star * viene usato in UNIX (ma non come apice, per ovvi motivi) con lo stesso significato già visto.

[3] Se si usa un codice diverso dall'ASCII, incluso un codice in cui le cifre non abbiano codici consecutivi, la notazione `[:digit:]` ha il vantaggio di rappresentare ancora `[0123456789]`, mentre `[0-9]` rappresenta qualunque carattere il cui codice sia compreso fra quelli di 0 e di 9, estremi inclusi.

Tutta la verità sulle espressioni regolari di UNIX

Il lettore che desideri l'elenco completo degli operatori e delle abbreviazioni disponibili nella notazione delle espressioni regolari di UNIX può trovarlo nelle pagine di manuale dei vari comandi. Ci sono alcune differenze tra le diverse versioni di UNIX, ma con un comando come `man grep` si ottiene la notazione usata per il comando `grep`, che è sempre presente. Per inciso, *grep* è l'acronimo di "Global (search for) Regular Expression and Print", ossia ricerca globale e stampa di espressioni regolari.

3.3.2 Analisi lessicale

Una delle prime applicazioni delle espressioni regolari riguarda la specifica del componente di un compilatore denominato "analizzatore lessicale". Questa componente scorre il programma sorgente e riconosce tutti i *token*, ossia quelle sottostringhe di caratteri consecutivi che formano una unità logica. Le parole-chiave e gli identificatori sono esempi tipici di *token*, ma ne esistono molti altri.

Il comando `lex` di UNIX e `flex`, la sua versione GNU, accettano come input una lista di espressioni regolari nello stile di UNIX, ciascuna seguita da una sezione di codice tra parentesi che indica che cosa deve fare l'analizzatore lessicale quando trova un *token* di quel tipo. Tale strumento è detto *generatore di analizzatori lessicali*, perché prende come input la descrizione ad alto livello di un analizzatore lessicale e ne trae una funzione che è un analizzatore lessicale funzionante.

Comandi come `lex` e `flex` si sono rivelati utilissimi: la notazione delle espressioni regolari è infatti proprio quella adeguata a descrivere i *token*. Per generare una funzione efficiente che scompone il codice sorgente in *token*, questi comandi sfruttano il processo di conversione da espressione regolare a DFA. In questo modo l'implementazione di un analizzatore lessicale si riduce al lavoro di un pomeriggio. Prima dello sviluppo di strumenti basati sulle espressioni regolari ci volevano mesi per generarlo manualmente. Inoltre, se per una ragione qualsiasi si rende necessario modificare un analizzatore lessicale, spesso basta cambiare una o due espressioni regolari anziché doversi addentrare nei meandri del codice per correggere un errore.

Esempio 3.9 Nella Figura 3.19 è illustrato un esempio di input parziale al comando `lex`, con la descrizione di alcuni *token* presenti nel linguaggio C. La prima riga tratta la parola-chiave `else`; l'azione consiste nel restituire una costante simbolica (`ELSE` in questo caso) al *parser* per l'ulteriore elaborazione. La seconda riga contiene un'espressione regolare che descrive gli identificatori: una lettera seguita da zero o più lettere o cifre. L'azione consiste innanzitutto nell'inserire l'identificatore nella tabella dei simboli, se non

è già presente; `lex` isola il *token* trovato in un buffer, in modo tale che questo segmento di codice sappia esattamente quale identificatore è stato trovato. Infine l'analizzatore lessicale restituisce la costante simbolica `ID`, scelta in quest'esempio per rappresentare gli identificatori.

```
else                        {return(ELSE);}

[A-Za-z][A-Za-z0-9]*        {codice per inserire
                             l'identificatore trovato
                             nella tabella dei simboli;
                             return(ID);
                            }

>=                          {return(GE);}

=                           {return(EQ);}

...
```

Figura 3.19 Un esempio di input per `lex`.

La terza voce della Figura 3.19 corrisponde a `>=`, un operatore di due caratteri. L'ultimo esempio è per il segno `=`, un operatore di un carattere. In pratica comparirebbero le espressioni che descrivono ognuna delle parole-chiave, ognuno dei segni e dei simboli di punteggiatura, come le virgole e le parentesi, e famiglie di costanti, come i numeri e le stringhe. Molte di queste sono semplici: una sequenza di uno o più caratteri specifici. Altre invece somigliano piuttosto agli identificatori e per essere descritte richiedono tutta la potenza della notazione delle espressioni regolari. Gli interi, i numeri a virgola mobile, le stringhe di caratteri e i commenti sono altri esempi di insiemi di stringhe che si giovano della capacità di comandi come `lex` di trattare espressioni regolari. □

La conversione in automa di una famiglia di espressioni come quelle suggerite nella Figura 3.19 procede più o meno nel modo descritto in termini formali nei paragrafi precedenti. Si parte da un automa per l'unione di tutte le espressioni. A rigor di termini, l'automa segnala solo che è stato riconosciuto un *token* non specificato. Tuttavia, se si segue la costruzione del Teorema 3.7 per l'unione delle espressioni, lo stato dell'ϵ-NFA indica esattamente quale *token* è stato riconosciuto.

L'unico problema è che può essere individuato più di un *token* alla volta; per esempio la stringa `else` si accorda non solo con l'espressione regolare **else**, ma anche con quella per gli identificatori. La soluzione consueta è far sì che l'analizzatore lessicale dia

priorità alla prima espressione nella lista. Dunque, se vogliamo che parole-chiave come `else` siano *riservate* (ossia non fruibili come identificatori), è sufficiente elencarle prima dell'espressione per gli identificatori.

3.3.3 Ricerca di pattern in un testo

Nel Paragrafo 2.4.1 abbiamo spiegato come gli automi possano essere usati per cercare efficientemente un insieme di parole in un grande archivio come il Web. Anche se gli strumenti e le tecniche impiegate a questo fine non hanno raggiunto un grado di sviluppo paragonabile a quello degli analizzatori lessicali, la notazione delle espressioni regolari è un mezzo efficace per descrivere la ricerca di *pattern* interessanti. Come per gli analizzatori lessicali, la possibilità di passare dalla notazione, naturale e descrittiva, delle espressioni regolari a un'implementazione efficiente, basata sugli automi, consente un significativo salto concettuale.

Le espressioni regolari si sono rivelate utili alla soluzione del problema generale di descrivere una classe di *pattern* testuali definita in modo approssimativo. L'indeterminatezza della descrizione in pratica garantisce che all'inizio il *pattern* non sarà descritto in termini corretti, e forse non lo sarà mai. Grazie alle espressioni regolari diventa facile descrivere i *pattern* ad alto livello con poca fatica, e modificare rapidamente la descrizione quando le cose vanno male. Per trasformare le espressioni in codice eseguibile, è utile disporre di un "compilatore" di espressioni regolari.

Vediamo ora un dettagliato esempio del tipo di problema che si presenta in molte applicazioni Web. Supponiamo di voler scorrere un numero elevato di pagine Web per individuare indirizzi. Potremmo voler creare una *mailing list* oppure classificare esercizi commerciali secondo la loro posizione, in modo da poter soddisfare richieste come "trova un ristorante a non più di dieci minuti di auto dal punto in cui mi trovo ora".

Ci concentriamo in particolare sul riconoscimento di indirizzi. Che cos'è un indirizzo? Dobbiamo dare una risposta, e se mettendo alla prova il software scopriamo che mancano alcuni casi, dobbiamo modificare l'espressione per recuperarli.[4] Per cominciare, un indirizzo finirà probabilmente con "Street" o con la sua abbreviazione, "St." Tuttavia esistono anche "Avenues" e "Roads", e anche questi potrebbero venire abbreviati. Possiamo dunque usare come parte finale della nostra espressione regolare:

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

Qui abbiamo usato la notazione in stile UNIX, con la barra verticale anziché + come operatore di unione. Notiamo che i punti sono preceduti dalla barra rovesciata, poiché nelle espressioni di UNIX il punto ha il significato speciale di "qualunque carattere", e in questo caso vogliamo il punto vero e proprio per chiudere le tre abbreviazioni; la barra annulla il significato speciale del simbolo che la segue.

[4] Manteniamo l'esempio originale, che fa riferimento a indirizzi nordamericani.

Un'indicazione come `Street` dev'essere preceduta dal nome della via. Di solito il nome è una lettera maiuscola seguita da alcune lettere minuscole. Possiamo descrivere questo schema con l'espressione di UNIX `[A-Z][a-z]*`. Alcune vie, però, hanno un nome composto da più parole, come Rhode Island Avenue a Washington. Dopo esserci resi conto che mancano gli indirizzi così congegnati, possiamo rivedere la descrizione e trasformarla in

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

Questa espressione comincia con un gruppo formato da una lettera maiuscola e zero o più lettere minuscole. Seguono quindi zero o più gruppi formati da uno spazio, un'altra lettera maiuscola e zero o più lettere minuscole. Lo spazio è un carattere normale nelle espressioni di UNIX, ma per evitare che l'espressione possa apparire come due espressioni separate da uno spazio in una riga di comando dobbiamo collocare gli apici prima e dopo. Essi non fanno parte dell'espressione. Ora dobbiamo includere il numero civico come parte dell'indirizzo. La maggior parte dei numeri civici sono formati da una sequenza di cifre. Alcuni però sono seguiti da una lettera, come in "123A Main St.". Dunque l'espressione per i numeri ha una lettera maiuscola opzionale che segue: `[0-9]+[A-Z]?`. Si noti che ricorriamo all'operatore di UNIX + per "una o più" cifre e all'operatore ? per "zero o una" lettera maiuscola. L'intera espressione per gli indirizzi è:

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

Lavorando con questa espressione si ottengono buoni risultati, ma prima o poi si scopre che manca qualcosa:

1. strade designate in modo diverso da street, avenue o road; per esempio "Boulevard", "Place", "Way" e le loro abbreviazioni
2. nomi di strade formati, anche parzialmente, da numeri, come "42nd Street"
3. caselle postali
4. nomi che non terminano per "Street" o simili. Un esempio è El Camino Real nella Silicon Valley. Dato che in spagnolo significa "la strada regale", sarebbe ridondante dire "El Camino Real Road", e dunque bisogna trattare anche indirizzi come "2000 El Camino Real"
5. ogni altra stranezza immaginabile.

Un compilatore di espressioni regolari può facilitare in modo significativo il processo di lenta convergenza verso un riconoscitore completo di indirizzi rispetto alla necessità di ricodificare direttamente ogni mutamento in un linguaggio di programmazione convenzionale.

3.3.4 Esercizi

! **Esercizio 3.3.1** Scrivete un'espressione regolare per descrivere i numeri di telefono in tutte le varie forme che riuscite a concepire. Considerate i numeri internazionali e anche il fatto che i prefissi e i numeri locali sono formati da un numero diverso di cifre in nazioni diverse.

!! **Esercizio 3.3.2** Scrivete un'espressione regolare per rappresentare gli stipendi come compaiono nelle offerte di lavoro. Considerate che gli stipendi possono essere indicati su base oraria, settimanale, mensile oppure annuale, e che possono essere accompagnati dal simbolo del dollaro o da unità di misura come "K". Una o più parole vicine possono indicare uno stipendio. *Suggerimento*: per farvi un'idea di quali *pattern* sono utili, esaminate gli annunci economici in un giornale oppure gli elenchi di offerte di lavoro on-line.

! **Esercizio 3.3.3** Alla fine del Paragrafo 3.3.3 abbiamo fornito alcuni esempi di migliorie per l'espressione regolare che descrive indirizzi. Modificate l'espressione in modo da includere tutte le opzioni menzionate.

3.4 Proprietà algebriche per le espressioni regolari

Per mantenere sotto controllo la dimensione delle espressioni regolari, abbiamo visto nell'Esempio 3.5 che è necessario semplificarle. Abbiamo quindi fornito alcune argomentazioni *ad hoc* per spiegare per quali ragioni un'espressione può essere sostituita da un'altra. In tutti i casi il punto fondamentale in discussione riguardava il fatto che le due espressioni erano *equivalenti*, nel senso che definivano gli stessi linguaggi. In questo paragrafo forniremo una raccolta di proprietà algebriche che spostano la questione dell'equivalenza di due espressioni regolari a un livello più alto. Invece di esaminare espressioni regolari specifiche, prenderemo in considerazione coppie di espressioni regolari con variabili come argomenti. Due espressioni con variabili sono *equivalenti* se generano lo stesso linguaggio per qualsiasi sostituzione delle variabili con linguaggi.

Consideriamo un esempio di questo processo nell'algebra dell'aritmetica. Un conto è dire che $1 + 2 = 2 + 1$: in questo caso siamo di fronte alla proprietà commutativa dell'addizione, che si può verificare facilmente applicando l'operatore di addizione a entrambi i membri, ottenendo $3 = 3$. Ma la *proprietà commutativa dell'addizione* dice di più: afferma infatti che $x + y = y + x$, dove x e y sono variabili che possono essere sostituite da due numeri qualsiasi. In altre parole la somma di due numeri qualsiasi dà lo stesso risultato a prescindere dal loro ordine.

Come per le espressioni aritmetiche, esistono proprietà per le espressioni regolari. Molte di esse sono simili a quelle aritmetiche, se si considera l'unione come somma e la concatenazione come prodotto. Tuttavia in alcuni punti l'analogia non regge, ed esistono anche proprietà valide per le espressioni regolari che non hanno corrispondenza in

aritmetica, soprattutto quando entra in gioco l'operatore di chiusura. I prossimi paragrafi forniscono un catalogo delle proprietà principali. Alla fine si discuterà di come sia possibile verificare se una presunta proprietà per le espressioni regolari sia veramente una proprietà: in altre parole, se è valida per qualunque linguaggio che può essere sostituito alle variabili.

3.4.1 Associatività e commutatività

La *commutatività* è la proprietà di un operatore per cui è possibile scambiare l'ordine dei suoi operandi e ottenere lo stesso risultato. Un esempio per l'aritmetica è stato fornito in precedenza: $x + y = y + x$. L'*associatività* è la proprietà di un operatore che consente di raggruppare gli operandi quando l'operatore viene impiegato due volte. Per esempio la proprietà associativa della moltiplicazione è $(x \times y) \times z = x \times (y \times z)$. Ecco tre proprietà di questo tipo, valide per le espressioni regolari.

- $L + M = M + L$. La *proprietà commutativa dell'unione* afferma che si può effettuare l'unione di due linguaggi in qualsiasi ordine.

- $(L + M) + N = L + (M + N)$. La *proprietà associativa dell'unione* afferma che è possibile effettuare l'unione di tre linguaggi facendo prima o l'unione dei primi due o l'unione degli ultimi due. Si noti che, considerando anche la proprietà commutativa, si conclude che è possibile effettuare l'unione di qualunque famiglia di linguaggi in qualunque ordine e raggruppamento con lo stesso risultato. Intuitivamente una stringa è in $L_1 \cup L_2 \cup \cdots \cup L_k$ se e solo se è in uno o più dei linguaggi L_i.

- $(LM)N = L(MN)$. La *proprietà associativa per la concatenazione* afferma che è possibile concatenare tre linguaggi concatenando inizialmente i primi due oppure gli ultimi due.

In questa lista manca la "proprietà" $LM = ML$. Ciò significherebbe che la concatenazione è commutativa. Questa affermazione è però falsa.

Esempio 3.10 Si considerino le espressioni regolari **01** e **10**. Queste espressioni denotano rispettivamente i linguaggi $\{01\}$ e $\{10\}$. Poiché i linguaggi sono diversi, la proprietà generale $LM = ML$ non è valida. Se lo fosse, si potrebbe sostituire l'espressione regolare **0** con L e **1** con M e si giungerebbe alla falsa conclusione che **01** = **10**. □

3.4.2 Identità e annichilatori

Un'*identità* per un operatore è un valore tale che, quando l'operatore viene applicato all'identità e a un altro valore, il risultato è l'altro valore. Per esempio 0 è l'identità per

l'addizione, in quanto $0 + x = x + 0 = x$, e 1 è l'identità per la moltiplicazione, dato che $1 \times x = x \times 1 = x$. Un *annichilatore* per un operatore è un valore tale che, quando l'operatore viene applicato all'annichilatore e a un altro valore, il risultato è l'annichilatore. Per esempio 0 è un annichilatore per la moltiplicazione, dato che $0 \times x = x \times 0 = 0$. Non ci sono annichilatori per l'addizione.

Questi concetti permeano le tre seguenti proprietà delle espressioni regolari.

- $\emptyset + L = L + \emptyset = L$. Questa proprietà asserisce che $\emptyset$ è l'identità per l'unione.
- $\epsilon L = L\epsilon = L$. Questa proprietà asserisce che ϵ è l'identità per la concatenazione.
- $\emptyset L = L\emptyset = \emptyset$. Questa proprietà asserisce che $\emptyset$ è l'annichilatore per la concatenazione.

Queste proprietà sono utili strumenti di semplificazione. Per esempio, se abbiamo un'unione di svariate espressioni, tra cui alcune sono $\emptyset$ oppure sono state semplificate in $\emptyset$, allora possiamo escluderle dall'unione. Analogamente, se si ha una concatenazione di diverse espressioni, tra cui alcune sono ϵ oppure sono state semplificate in ϵ, nella concatenazione possiamo ometterle. Infine, se si ha una concatenazione di un qualunque numero di espressioni, e se anche una sola di loro è $\emptyset$, l'intera concatenazione può essere sostituita da $\emptyset$.

3.4.3 Proprietà distributive

Una *proprietà distributiva* coinvolge due operatori e afferma che un operatore può essere applicato a ogni argomento dell'altro operatore individualmente. L'esempio più comune nell'aritmetica è la proprietà distributiva della moltiplicazione rispetto all'addizione: $x \times (y + z) = x \times y + x \times z$. Dato che la moltiplicazione è commutativa, non ha importanza se la moltiplicazione è a sinistra o a destra della somma. Esiste una proprietà analoga per le espressioni regolari, che dobbiamo però enunciare in due forme, visto che la concatenazione non è commutativa.

- $L(M + N) = LM + LN$. Questa è la *proprietà distributiva sinistra della concatenazione sull'unione*.
- $(M + N)L = ML + NL$. Questa è la *proprietà distributiva destra della concatenazione sull'unione*.

Dimostriamo la proprietà distributiva destra; per l'altra si procede in modo analogo. La dimostrazione riguarderà esclusivamente linguaggi e non dipenderà dalla presenza di espressioni regolari nei linguaggi.

Teorema 3.11 Se L, M ed N sono linguaggi arbitrari, allora

$$L(M \cup N) = LM \cup LN$$

DIMOSTRAZIONE La dimostrazione è simile a quella di un'altra proprietà distributiva che abbiamo visto nel Teorema 1.10. Dobbiamo dimostrare che una stringa w è in $L(M \cup N)$ se e solo se è in $LM \cup LN$.

(Solo se) Se w è in $L(M \cup N)$, allora $w = xy$, dove x è in L e y è in M oppure in N. Se y è in M, allora xy è in LM, e dunque in $LM \cup LN$. Analogamente, se y è in N, allora xy è in LN, e dunque in $LM \cup LN$.

(Se) Supponiamo che w sia in $LM \cup LN$. Allora w è in LM oppure in LN. In primo luogo supponiamo che w sia in LM. Allora $w = xy$, dove x è in L e y è in M. Dato che y è in M, è anche in $M \cup N$. Dunque xy è in $L(M \cup N)$. Se w non è in LM, allora è sicuramente in LN, e un'argomentazione analoga dimostra che è in $L(M \cup N)$. □

Esempio 3.12 Consideriamo l'espressione regolare $\mathbf{0} + \mathbf{01}^*$. Possiamo raccogliere $\mathbf{0}$ dall'unione, ma prima dobbiamo riconoscere che l'espressione $\mathbf{0}$ stessa è effettivamente la concatenazione di $\mathbf{0}$ con altro, vale a dire ϵ. In altre parole, usiamo la proprietà dell'identità per la concatenazione per sostituire $\mathbf{0}$ con $\mathbf{0}\epsilon$, ottenendo l'espressione $\mathbf{0}\epsilon+\mathbf{01}^*$. A questo punto, applicando la proprietà distributiva sinistra otteniamo l'espressione $\mathbf{0}(\epsilon + \mathbf{1}^*)$. Se inoltre riconosciamo che ϵ è in $L(\mathbf{1}^*)$, allora osserviamo che $\epsilon + \mathbf{1}^* = \mathbf{1}^*$, e possiamo semplificarla in $\mathbf{01}^*$. □

3.4.4 La proprietà di idempotenza

Un operatore si dice *idempotente* se il risultato della sua applicazione a due valori uguali è lo stesso valore. I comuni operatori aritmetici non sono idempotenti; generalmente $x+x \neq x$, così come $x \times x \neq x$ (sebbene esistano *alcuni* valori di x per cui l'uguaglianza è valida, come $0+0 = 0$). L'unione e l'intersezione sono invece esempi comuni di operatori idempotenti. Per le espressioni regolari possiamo perciò enunciare la seguente proprietà.

- $L + L = L$. La *proprietà dell'idempotenza per l'unione* afferma che l'unione di due espressioni identiche equivale a una singola copia dell'espressione.

3.4.5 Proprietà relative alla chiusura

Alcune proprietà riguardano gli operatori di chiusura e le loro varianti in stile UNIX, $^+$ e ?. Ne presentiamo un elenco e spieghiamo per quali motivi sono valide.

- $(L^*)^* = L^*$. Questa proprietà afferma che chiudere un'espressione che è già chiusa non cambia il linguaggio. Il linguaggio di $(L^*)^*$ è composto da tutte le stringhe

create concatenando stringhe nel linguaggio di L^*. Ma tali stringhe sono a loro volta composte da stringhe tratte da L. Perciò anche una stringa in $(L^*)^*$ è una concatenazione di stringhe da L, ed è quindi nel linguaggio di L^*.

- $\emptyset^* = \epsilon$. La chiusura di $\emptyset$ contiene solamente la stringa ϵ, come è stato discusso nell'Esempio 3.6.

- $\epsilon^* = \epsilon$. È facile verificare che l'unica stringa che può essere formata concatenando un qualunque numero di copie della stringa vuota è la stringa vuota stessa.

- $L^+ = LL^* = L^*L$. Ricordiamo che L^+ è definito come $L + LL + LLL + \cdots$. Inoltre $L^* = \epsilon + L + LL + LLL + \cdots$. Dunque

$$LL^* = L\epsilon + LL + LLL + LLLL + \cdots$$

  Se consideriamo che $L\epsilon = L$, deduciamo che le espansioni infinite per LL^* e per L^+ sono le stesse. Ciò dimostra $L^+ = LL^*$. La dimostrazione che $L^+ = L^*L$ è analoga.[5]

- $L^* = L^+ + \epsilon$. La dimostrazione è facile, dato che l'espansione di L^+ include ogni termine nell'espansione di L^* eccetto ϵ. Si osservi che se il linguaggio L contiene la stringa ϵ, allora il termine aggiuntivo "$+\epsilon$" non è necessario; in altre parole, in questo caso particolare $L^+ = L^*$.

- $L? = \epsilon + L$. Questa regola è di fatto la definizione dell'operatore ? .

3.4.6 Alla ricerca di proprietà per le espressioni regolari

Ognuna delle proprietà menzionate sopra è stata dimostrata, in termini formali o informali. D'altra parte si potrebbero congetturare infinite altre proprietà. Esiste un metodo generale che facilita la dimostrazione delle proprietà valide? Scopriamo che in effetti la validità di una proprietà si riduce a una questione di uguaglianza di due linguaggi specifici. È interessante che la tecnica sia strettamente connessa agli operatori delle espressioni regolari e non si possa estendere a espressioni con altri operatori, come l'intersezione.

Per entrare nel vivo di questo metodo, consideriamo una presunta proprietà, come

$$(L + M)^* = (L^*M^*)^*$$

Questa proprietà afferma che se abbiamo due linguaggi qualsiasi L ed M, e chiudiamo la loro unione, otteniamo lo stesso linguaggio che otterremmo se prendessimo il linguaggio

[5] Si noti che di conseguenza qualsiasi linguaggio L commuta (nella concatenazione) con la sua chiusura: $LL^* = L^*L$. Tale regola non contraddice il fatto che generalmente la concatenazione non è commutativa.

L^*M^*, ossia tutte le stringhe composte da zero o più scelte da L seguite da zero o più scelte da M, e lo chiudessimo.

Per dimostrarla, si supponga in primo luogo che la stringa w sia nel linguaggio $(L + M)^*$.[6] Allora possiamo scrivere $w = w_1w_2 \cdots w_k$ per un certo k, dove ogni w_i è in L oppure in M. Ne consegue che ogni w_i è nel linguaggio L^*M^*. Per capirne il motivo, se w_i è in L, si prenda una stringa w_i da L (tale stringa è anche in L^*). Non si prenda invece alcuna stringa da M, o meglio si prenda ϵ da M^*. Se w_i è in M, l'argomentazione è simile. Una volta che si è dimostrato che w_i è in L^*M^*, ne consegue che w è nella chiusura di tale linguaggio.

Per completare la dimostrazione dobbiamo inoltre dimostrare l'inverso, ossia che le stringhe in $(L^*M^*)^*$ sono anche in $(L + M)^*$. Dato che il nostro obiettivo non è dimostrare la proprietà, ma mettere in luce l'importante proprietà delle espressioni regolari di cui parleremo in seguito, tralasciamo questa parte della dimostrazione.

Qualunque espressione con variabili può essere pensata come un'espressione regolare *concreta*, cioè priva di variabili, considerando ogni variabile come se fosse un simbolo distinto. Per esempio nell'espressione $(L + M)^*$ le variabili L ed M possono essere rimpiazzate, rispettivamente, dai simboli a e b. Si ottiene così l'espressione regolare $(\mathbf{a} + \mathbf{b})^*$.

Una volta sostituite le variabili con linguaggi, il linguaggio dell'espressione concreta dà informazioni sulla forma delle stringhe di qualsiasi linguaggio formato dall'espressione originale. Così nella nostra analisi di $(L + M)^*$ abbiamo osservato che qualunque stringa w composta da una sequenza di scelte da L oppure da M si trova nel linguaggio $(L + M)^*$. Arriviamo a questa conclusione esaminando il linguaggio dell'espressione concreta $L\big((\mathbf{a} + \mathbf{b})^*\big)$, che è evidentemente l'insieme di tutte le stringhe di a e b. Potremmo sostituire qualunque occorrenza di a in una di quelle stringhe con una stringa qualsiasi di L, e qualunque occorrenza di b con una stringa qualsiasi di M, scegliendo eventualmente stringhe diverse per diverse occorrenze di a oppure b. Tali sostituzioni, applicate a tutte le stringhe in $(\mathbf{a} + \mathbf{b})^*$, danno tutte le stringhe formate concatenando stringhe di L ed M, in qualsiasi ordine.

L'enunciato qui sopra può apparire ovvio, ma come è sottolineato nel riquadro "Il metodo non si può estendere al di là delle espressioni regolari", la sua validità viene meno quando ai tre operatori delle espressioni regolari se ne aggiungono altri. Nel prossimo teorema dimostriamo il principio generale per le espressioni regolari.

Teorema 3.13 Sia E un'espressione regolare con variabili $L_1, L_2, \ldots, L_m$. Formiamo l'espressione regolare concreta C sostituendo ogni occorrenza di L_i con il simbolo a_i, per $i = 1, 2, \ldots, m$. Allora per ogni sequenza di linguaggi $L_1, L_2, \ldots, L_m$ ogni stringa w in $L(E)$ può essere scritta $w = w_1w_2 \cdots w_k$, dove ogni w_i è in uno dei linguaggi,

[6] Per semplicità identificheremo le espressioni regolari e i loro linguaggi, ed eviteremo di specificare "il linguaggio di" davanti a ogni espressione regolare.

poniamo L_{j_i}, e la stringa $a_{j_1}a_{j_2}\cdots a_{j_k}$ è nel linguaggio $L(C)$. In termini meno formali, possiamo costruire $L(E)$ cominciando da una stringa in $L(C)$, poniamo $a_{j_1}a_{j_2}\cdots a_{j_k}$, e sostituendo a ognuno degli a_{j_i} una stringa del linguaggio L_{j_i} corrispondente.

DIMOSTRAZIONE La dimostrazione è un'induzione strutturale sull'espressione E.

BASE I casi fondamentali si hanno laddove E è ϵ, $\emptyset$ o una variabile L. Nei primi due casi non c'è nulla da dimostrare, in quanto l'espressione concreta C coincide con E. Se E è una variabile L, allora $L(E) = L$. L'espressione concreta C è $\mathbf{a}$, dove a è il simbolo corrispondente a L. Dunque $L(C) = \{a\}$. Se in quest'unica stringa sostituiamo il simbolo a con tutte le stringhe di L otteniamo il linguaggio L, che è anche $L(E)$.

INDUZIONE A seconda dell'ultimo operatore di E possiamo avere tre casi. In primo luogo supponiamo che $E = F + G$, ossia che l'ultimo operatore sia l'unione. Siano C e D le espressioni concrete formate, rispettivamente, da F e G, sostituendo le variabili con simboli concreti. Si noti che lo stesso simbolo deve sostituire tutte le occorrenze della stessa variabile, sia in F sia in G. Allora l'espressione concreta che si ottiene da E è $C + D$, e $L(C + D) = L(C) + L(D)$.

Supponiamo che w sia una stringa in $L(E)$, con le variabili di E rimpiazzate da linguaggi specifici. Allora w è in $L(F)$ oppure in $L(G)$. Per l'ipotesi induttiva, w si ottiene a partire da una stringa concreta in $L(C)$ o in $L(D)$ sostituendo i simboli con stringhe dei linguaggi corrispondenti. Di conseguenza, in entrambi i casi, la stringa w può essere costruita a partire da una stringa concreta in $L(C + D)$ e compiendo le stesse sostituzioni di simboli con stringhe.

Dobbiamo inoltre considerare i casi in cui E è FG oppure F^*. Le dimostrazioni sono analoghe a quelle viste sopra nel caso di unione. Le lasciamo al lettore. □

3.4.7 Verifica di proprietà algebriche sulle espressioni regolari

Possiamo ora enunciare e dimostrare il metodo per verificare la validità di una proprietà relativa alle espressioni regolari. Per verificare se $E = F$ è vero, dove E ed F sono due espressioni regolari con lo stesso insieme di variabili, si procede in questo modo.

1. Si convertono E ed F nelle espressioni regolari concrete C e D, sostituendo ogni variabile con un simbolo concreto.

2. Si verifica se $L(C) = L(D)$. Se è così, allora $E = F$ è una proprietà valida; se invece non è così, allora la "proprietà" è falsa. Si tenga conto che fino al Paragrafo 4.4 non tratteremo la procedura per verificare se due espressioni regolari denotano lo stesso linguaggio. Tuttavia, per stabilire l'uguaglianza di due particolari linguaggi è possibile usare strumenti *ad hoc*. Ricordiamo che se i linguaggi *non* sono identici è sufficiente fornire un controesempio: una stringa che sia in un linguaggio ma non nell'altro.

Teorema 3.14 Il metodo descritto sopra individua correttamente le proprietà valide per le espressioni regolari.

DIMOSTRAZIONE Mostreremo che $L(E) = L(F)$ per ogni sostituzione di linguaggi alle variabili di E ed F se e solo se $L(C) = L(D)$.

(Solo se) Supponiamo che $L(E) = L(F)$ per tutte le scelte di linguaggi al posto delle variabili. In particolare scegliamo per ogni variabile L il simbolo concreto a che rimpiazza L nelle espressioni C e D. Allora per questa scelta $L(C) = L(E)$ e $L(D) = L(F)$. Poiché $L(E) = L(F)$ è dato, ne consegue che $L(C) = L(D)$.

(Se) Supponiamo che $L(C) = L(D)$. Per il Teorema 3.13 $L(E)$ e $L(F)$ sono entrambi costruiti rimpiazzando i simboli concreti delle stringhe in $L(C)$ e $L(D)$ con stringhe nei linguaggi che corrispondono a quei simboli. Se le stringhe di $L(C)$ e $L(D)$ sono le stesse, allora anche i due linguaggi costruiti in questo modo saranno gli stessi; in altre parole, $L(E) = L(F)$. □

Esempio 3.15 Consideriamo una proprietà plausibile: $(L + M)^* = (L^*M^*)^*$. Se rimpiazziamo le variabili L ed M con i simboli concreti a e b, otteniamo le espressioni regolari $(\mathbf{a} + \mathbf{b})^*$ e $(\mathbf{a}^*\mathbf{b}^*)^*$. È facile verificare che entrambe denotano il linguaggio formato da tutte le stringhe di a e b. Di conseguenza le due espressioni concrete denotano lo stesso linguaggio e la proprietà è valida.

Come ulteriore esempio consideriamo $L^* = L^*L^*$. I linguaggi concreti sono rispettivamente $\mathbf{a}^*$ e $\mathbf{a}^*\mathbf{a}^*$, e ognuno di questi è l'insieme di tutte le stringhe di a. Anche in questo caso la proprietà risulta valida; in altre parole la concatenazione di un linguaggio chiuso con se stesso produce il linguaggio stesso.

Consideriamo infine la presunta proprietà $L + ML = (L + M)L$. Se scegliamo i simboli a e b per le variabili L ed M, otteniamo le due espressioni regolari concrete $\mathbf{a} + \mathbf{ba}$ e $(\mathbf{a} + \mathbf{b})\mathbf{a}$. Ebbene, i linguaggi di queste espressioni non sono gli stessi. Per esempio la stringa aa è nel secondo, ma non nel primo. La presunta proprietà risulta perciò falsa. □

3.4.8 Esercizi

Esercizio 3.4.1 Verificate le seguenti identità fra espressioni regolari.

* a) $R + S = S + R$.

b) $(R + S) + T = R + (S + T)$.

c) $(RS)T = R(ST)$.

d) $R(S + T) = RS + RT$.

Il metodo non si può estendere al di là delle espressioni regolari

Consideriamo un'algebra estesa delle espressioni regolari che includa l'operatore di intersezione. È interessante notare che, come vedremo nel Teorema 4.8, se si aggiunge $\cap$ ai tre operatori delle espressioni regolari, l'insieme dei linguaggi che possiamo descrivere non si allarga. D'altra parte il metodo di verifica delle proprietà algebriche non è più valido.

Prendiamo in esame la "proprietà" $L \cap M \cap N = L \cap M$, cioè l'intersezione di tre linguaggi qualsiasi coincide con l'intersezione dei primi due. Questa "proprietà" è palesemente falsa. Per esempio siano $L = M = \{a\}$ e $N = \emptyset$. Il metodo basato sulla sostituzione delle variabili non coglie la differenza. In altre parole, sostituendo L, M ed N con i simboli a, b e c, dovremmo verificare se $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Poiché entrambi i membri sono l'insieme vuoto, i linguaggi coincidono e dovremmo concludere che la "proprietà" è vera.

e) $(R + S)T = RT + ST$.

* f) $(R^*)^* = R^*$.

g) $(\epsilon + R)^* = R^*$.

h) $(R^*S^*)^* = (R + S)^*$.

! **Esercizio 3.4.2** Dimostrate o confutate i seguenti enunciati su espressioni regolari.

* a) $(R + S)^* = R^* + S^*$.

b) $(RS + R)^*R = R(SR + R)^*$.

* c) $(RS + R)^*RS = (RR^*S)^*$.

d) $(R + S)^*S = (R^*S)^*$.

e) $S(RS + S)^*R = RR^*S(RR^*S)^*$.

Esercizio 3.4.3 Nell'Esempio 3.6 abbiamo sviluppato l'espressione regolare

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

Usate le proprietà distributive per sviluppare due espressioni equivalenti, diverse e più semplici.

Esercizio 3.4.4 All'inizio del Paragrafo 3.4.6 abbiamo svolto parte di una dimostrazione che $(L^*M^*)^* = (L + M)^*$. Completate la dimostrazione mostrando che le stringhe in $(L^*M^*)^*$ sono anche in $(L + M)^*$.

! **Esercizio 3.4.5** Completate la dimostrazione del Teorema 3.13 trattando i casi in cui l'espressione regolare E è della forma FG oppure della forma F^*.

3.5 Riepilogo

- *Espressioni regolari*: notazione algebrica che descrive esattamente gli stessi linguaggi definiti dagli automi a stati finiti: i linguaggi regolari. Gli operatori delle espressioni regolari sono unione, concatenazione (o "punto") e chiusura (o "star").

- *Le espressioni regolari nella pratica*: molti comandi di sistemi come UNIX usano un linguaggio delle espressioni regolari esteso dotato di scorciatoie per molte espressioni comuni. Le classi di caratteri permettono di esprimere agevolmente insiemi di simboli, mentre gli operatori come uno-o-più e al-massimo-uno accrescono gli usuali operatori delle espressioni regolari.

- *Equivalenza di espressioni regolari e automi a stati finiti*: possiamo convertire un DFA in un'espressione regolare per mezzo di una costruzione induttiva in cui si creano espressioni per le etichette di cammini che possono passare attraverso insiemi di stati sempre più grandi. In alternativa possiamo usare una procedura di eliminazione di stati per costruire l'espressione regolare per un DFA. Nell'altra direzione possiamo costruire ricorsivamente un ϵ-NFA da un'espressione regolare e poi eventualmente convertirlo in un DFA.

- *L'algebra delle espressioni regolari*: sebbene esistano differenze, le espressioni regolari obbediscono a molte delle leggi algebriche dell'aritmetica. Unione e concatenazione sono associative, ma solo l'unione è commutativa. La concatenazione è distributiva rispetto all'unione. L'unione è idempotente.

- *Verificare identità algebriche*: possiamo stabilire se un'equivalenza di espressioni regolari con variabili come argomenti è vera rimpiazzando le variabili con costanti distinte e verificando se i linguaggi che ne risultano sono gli stessi.

3.6 Bibliografia

L'idea di espressione regolare e la dimostrazione dell'equivalenza con gli automi a stati finiti sono di S. C. Kleene [3]. La costruzione di un ϵ-NFA da un'espressione regolare

che abbiamo presentato qui si deve invece a McNaughton e a Yamada ([4]). Il metodo per verificare identità relative alle espressioni regolari trattando le variabili come costanti è stato pubblicato da J. Gischer [2]. In questo lavoro si dimostra anche una proprietà ritenuta patrimonio comune: che l'aggiunta di altre operazioni, come l'intersezione o lo *shuffle* (vedi Esercizio 7.3.4), invalida il metodo, anche se la classe dei linguaggi rappresentabili non si allarga.

Ancora prima di sviluppare UNIX, Ken Thompson aveva studiato l'impiego delle espressioni regolari in comandi come `grep`; il suo algoritmo per il trattamento di questi comandi compare in [5]. Già dalle prime fasi di sviluppo di UNIX, diversi altri comandi, come `lex`, ideato da M. Lesk, sfruttavano la notazione delle espressioni regolari estese. Una descrizione di `lex` e di altre tecniche fondate sulle espressioni regolari si trova in [1].

1. A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

2. J. L. Gischer, STAN-CS-TR-84-1033 (1984).

3. S. C. Kleene, "Representation of events in nerve nets and finite automata," in C. E. Shannon, J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 3–42.

4. R. McNaughton, H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. Electronic Computers* **9**:1 (Jan., 1960), pp. 39–47.

5. K. Thompson, "Regular expression search algorithm," *Comm. ACM* **11**:6 (Giugno, 1968), pp. 419–422.

Capitolo 4

Proprietà dei linguaggi regolari

Questo capitolo prende in esame le proprietà dei linguaggi regolari. Il primo strumento di indagine è una tecnica per dimostrare che certi linguaggi non sono regolari espressa in forma di teorema, il *pumping lemma*, introdotto nel Paragrafo 4.1.

Nei linguaggi regolari ha grande rilevanza una classe di proprietà dette di "chiusura". Queste proprietà permettono di costruire riconoscitori di linguaggi formati a partire da altri linguaggi con opportune operazioni. Per esempio l'intersezione di due linguaggi regolari è regolare; dati gli automi che riconoscono due linguaggi regolari, possiamo costruire meccanicamente un automa che riconosce la loro intersezione. Poiché l'automa per l'intersezione può avere molti più stati dei due automi dati, la proprietà di chiusura si rivela uno strumento utile per costruire automi complessi. Nel Paragrafo 2.1 questa costruzione è stata sfruttata in modo determinante.

Un'altra importante classe di fatti relativi ai linguaggi regolari comprende le "proprietà di decisione". Il loro studio fornisce algoritmi per risolvere questioni rilevanti. Un esempio basilare concerne un algoritmo per decidere se due automi definiscono lo stesso linguaggio. Sfruttando la capacità di risolvere tale questione, siamo in grado di "minimizzare" un automa, cioè di trovare un automa equivalente con il minor numero possibile di stati. Dato che l'area di un chip occupata da un circuito (cioè il suo costo) tende a diminuire quando diminuisce il numero di stati dell'automa realizzato da quello, da decenni questo problema riveste particolare importanza nella progettazione di circuiti digitali.

4.1 Dimostrare che un linguaggio non è regolare

Abbiamo stabilito che la classe dei linguaggi detti regolari ammette almeno quattro descrizioni diverse: i linguaggi accettati dai DFA, dagli NFA e dagli ϵ-NFA, e inoltre i linguaggi definiti dalle espressioni regolari.

Non tutti i linguaggi sono però regolari. Per dimostrare che certi linguaggi non lo sono, introduciamo in questo paragrafo una tecnica efficace, nota come *pumping lemma*. Forniremo poi alcuni esempi di linguaggi non regolari. Nel Paragrafo 4.2 vedremo come usare il *pumping lemma* insieme alle proprietà di chiusura dei linguaggi regolari per dimostrare che altri linguaggi non lo sono.

4.1.1 Il pumping lemma per i linguaggi regolari

Consideriamo il linguaggio $L_{01} = \{0^n1^n \mid n \geq 1\}$. Esso contiene le stringhe 01, 0011, 000111, e così via, cioè le stringhe formate da uno o più 0 seguiti dallo stesso numero di 1. Affermiamo che L_{01} non è regolare. Il ragionamento intuitivo è che se L_{01} fosse regolare sarebbe il linguaggio di un DFA A. Questo automa avrebbe un certo numero di stati, poniamo k. Immaginiamo che l'automa riceva k 0 in input. Dopo aver letto i $k+1$ prefissi di tale stringa: $\epsilon, 0, 00, \ldots, 0^k$, l'automa si trova in un certo stato. Poiché ci sono solo k stati, per il principio della piccionaia ci devono essere due prefissi, poniamo 0^i e 0^j, letti i quali A si trova nello stesso stato, diciamo q.

Supponiamo ora che dopo aver letto i o j simboli 0 l'automa cominci a ricevere degli 1 in input. Dopo aver ricevuto i 1, l'automa deve accettare la stringa se ha ricevuto in precedenza i 0, ma non se ne ha ricevuti j. Trovandosi in q all'inizio degli 1, l'automa non può "ricordare" quale dei due casi si verifica; possiamo perciò "ingannare" A e indurlo a prendere la decisione sbagliata: accettare anche se non dovrebbe oppure, nel caso opposto, rifiutare.

Questo ragionamento informale può essere precisato. La stessa conclusione, che il linguaggio L_{01} non è regolare, si può comunque raggiungere per mezzo del seguente risultato generale.

Teorema 4.1 (Il *pumping lemma per i linguaggi regolari*) Sia L un linguaggio regolare. Esiste allora una costante n (dipendente da L) tale che, per ogni stringa w in L per la quale $|w| \geq n$, possiamo scomporre w in tre stringhe $w = xyz$ in modo che:

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. per ogni $k \geq 0$ anche la stringa xy^kz è in L.

In altre parole possiamo sempre trovare un stringa non vuota y, non troppo distante dall'inizio di w, da "replicare", cioè da ripetere quante volte vogliamo o anche cancellare (è il caso di $k = 0$) senza uscire dal linguaggio L.

DIMOSTRAZIONE Supponiamo che L sia regolare. Allora $L = L(A)$ per un certo DFA A. Supponiamo che A abbia n stati. Consideriamo una stringa w di lunghezza n o più:

$w = a_1a_2 \cdots a_m$, dove $m \geq n$ e ogni a_i è un simbolo di input. Per $i = 0, 1, \ldots, n$ definiamo lo stato p_i come $\hat{\delta}(q_0, a_1a_2 \cdots a_i)$, dove δ è la funzione di transizione di A e q_0 è lo stato iniziale. Dunque p_i è lo stato in cui si trova A dopo aver letto i primi i simboli di w. Si noti che $p_0 = q_0$.

Per il principio della piccionaia, dato che ci sono solo n stati, gli $n + 1$ stati p_i per $i = 0, 1, \ldots, n$ non possono essere tutti distinti. Ci sono perciò due interi distinti, i e j, con $0 \leq i < j \leq n$, tali che $p_i = p_j$. Possiamo ora scomporre $w = xyz$:

1. $x = a_1a_2 \cdots a_i$
2. $y = a_{i+1}a_{i+2} \cdots a_j$
3. $z = a_{j+1}a_{j+2} \cdots a_m$.

In altre parole x porta a p_i la prima volta; y riporta da p_i a p_i (dato che p_i e p_j coincidono), mentre z conclude w. Le relazioni fra stringhe e stati sono indicate nella Figura 4.1. Si noti che x può essere vuota (quando $i = 0$), così come z (quando $j = n = m$). Viceversa, y non può essere vuota perché i è strettamente minore di j.

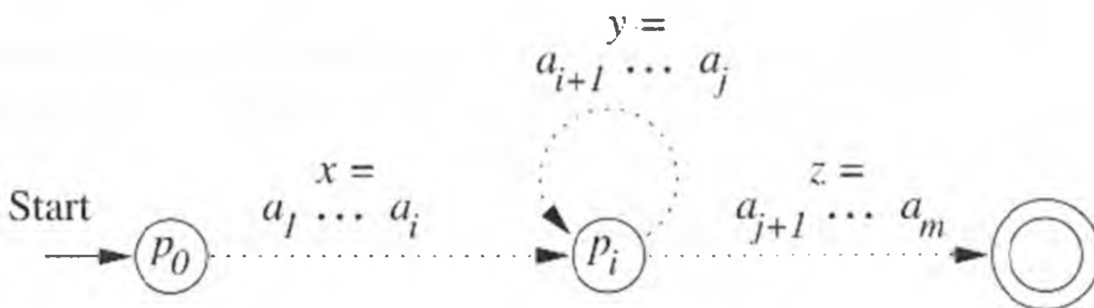


Figura 4.1 Ogni stringa di lunghezza superiore al numero di stati obbliga a ripetere uno stato.

Consideriamo ora che cosa accade se l'automa A riceve l'input xy^kz per un $k \geq 0$. Se $k = 0$, l'automa passa dallo stato iniziale q_0 (che è anche p_0) a p_i sull'input x. Poiché p_i coincide con p_j, A deve passare da p_i allo stato accettante della Figura 4.1 a fronte dell'input z. Perciò A accetta xz.

Se $k > 0$, A va da q_0 a p_i su input x, cicla su p_i per k volte a fronte dell'input y^k, e passa allo stato accettante a fronte di z. Perciò, per ogni $k \geq 0$, anche xy^kz è accettato da A; quindi xy^kz appartiene a L. □

4.1.2 Applicazioni del pumping lemma

Vediamo alcuni esempi relativi all'uso del *pumping lemma*. In ciascun caso proponiamo un linguaggio e per mezzo del lemma dimostriamo che non è regolare.

Il pumping lemma come gioco a due

Nel Paragrafo 1.2.3 abbiamo visto che un teorema il cui enunciato comprende diverse alternative di quantificatori "per ogni" ed "esiste" può essere considerato come una partita fra due giocatori. Il *pumping lemma* è un esempio rilevante di questo tipo di teorema perché comporta quattro quantificatori distinti: "**per ogni** linguaggio regolare L, **esiste** n tale che, **per ogni** w in L con $|w| \geq n$, **esiste** xyz uguale a w tale che $\cdots$." Possiamo interpretare l'applicazione del lemma come un gioco in cui:

1. il giocatore 1 sceglie il linguaggio L, che si suppone non regolare
2. il giocatore 2 sceglie n senza rivelarlo a 1; 1 deve ideare una strategia per qualsiasi n
3. il giocatore 1 sceglie w, che può dipendere da n e dev'essere di lunghezza almeno pari a n
4. il giocatore 2 scompone w in x, y e z, rispettando i vincoli imposti dal lemma: $y \neq \epsilon$ e $|xy| \leq n$; anche in questo caso 2 non deve rivelare x, y e z a 1, ma deve soltanto rispettare i vincoli
5. il giocatore 1 "vince" scegliendo k, che può dipendere da n, x, y e z, in modo che xy^kz non appartenga a L.

Esempio 4.2 Dimostriamo che il linguaggio L_{eq} formato da tutte le stringhe con lo stesso numero di 0 e di 1 (senza vincoli sull'ordine) non è regolare. Rispetto al gioco descritto nel riquadro "Il *pumping lemma* come gioco a due", faremo la parte del giocatore 1: dobbiamo rispondere a ogni mossa del giocatore 2. Sia n la costante citata nel *pumping lemma*, nel caso che L_{eq} fosse regolare. Il giocatore 2 sceglie n. Noi scegliamo allora $w = 0^n1^n$, cioè n simboli 0 seguiti da n simboli 1, una stringa che appartiene senz'altro a L_{eq}.

Ora il giocatore 2 scompone w in xyz. Noi sappiamo solo che $y \neq \epsilon$ e $|xy| \leq n$. Ma questa informazione è utile e ci basta per "vincere". Poiché $|xy| \leq n$ e xy si trova all'inizio di w, sappiamo che x e y sono formate da soli 0. Il *pumping lemma* dice che, se L_{eq} fosse regolare, anche xz gli apparterrebbe. È il caso $k = 0$ del lemma.[1] D'altra parte xz ha n 1 perché tutti gli 1 di w sono in z. Ma xz ha anche meno di n 0 perché

[1] Si noti che avremmo vinto anche scegliendo $k = 2$ o, in generale, qualsiasi valore diverso da 1.

mancano quelli di y. Poiché $y \neq \epsilon$, non ci possono essere più di $n - 1$ simboli 0 in x e z insieme. Così, dopo aver supposto che L_{eq} fosse regolare, abbiamo dimostrato un fatto che sappiamo essere falso, cioè che xz appartiene a L_{eq}. Abbiamo dunque dimostrato per assurdo che L_{eq} non è regolare. □

Esempio 4.3 Dimostriamo che il linguaggio L_{pr}, formato da tutte le stringhe di 1 la cui lunghezza è un numero primo, non è regolare. Supponiamo che lo sia. Allora esiste una costante n che soddisfa le condizioni del *pumping lemma*. Consideriamo un numero primo $p \geq n + 2$; dal momento che i numeri primi sono infiniti, un numero simile esiste. Sia $w = 1^p$.

Per il *pumping lemma* possiamo scomporre $w = xyz$, con $y \neq \epsilon$ e $|xy| \leq n$. Sia $|y| = m$. Ne segue $|xz| = p - m$. Consideriamo ora la stringa $xy^{p-m}z$, che in base al lemma dovrebbe appartenere a L_{pr} se questo fosse regolare. Abbiamo però

$$|xy^{p-m}z| = |xz| + (p - m)|y| = p - m + (p - m)m = (m + 1)(p - m)$$

Poiché ha due fattori, $m + 1$ e $p - m$, il numero $|xy^{p-m}z|$ non sembra essere primo. Dobbiamo però verificare che nessun fattore sia 1 perché in tal caso il numero potrebbe essere primo. Poiché $y \neq \epsilon$, vale $m \geq 1$, da cui $m + 1 > 1$. Anche $p - m > 1$, perché abbiamo scelto $p \geq n + 2$, e $m \leq n$, dato che

$$m = |y| \leq |xy| \leq n$$

Perciò $p - m \geq 2$.

Anche in questo caso abbiamo anzitutto supposto che un linguaggio fosse regolare, e ne abbiamo derivato una contraddizione mostrando che una stringa non appartenente al linguaggio dovrebbe invece appartenergli, secondo il *pumping lemma*. Deduciamo così che L_{pr} non è un linguaggio regolare. □

4.1.3 Esercizi

Esercizio 4.1.1 Dimostrate che i seguenti linguaggi non sono regolari.

a) $\{0^n1^n \mid n \geq 1\}$. Questo linguaggio, i cui elementi sono formati da una stringa di 0 seguita da una stringa di uguale lunghezza di 1, è proprio L_{01}, il linguaggio esaminato informalmente all'inizio del paragrafo. Nella dimostrazione applicate il *pumping lemma*.

b) L'insieme delle stringhe di parentesi bilanciate. Si tratta delle stringhe di "(" e ")" che compaiono nelle espressioni aritmetiche ben formate.

* c) $\{0^n10^n \mid n \geq 1\}$.

d) $\{0^n 1^m 2^n \mid n \text{ ed } m \text{ sono interi arbitrari}\}$.

e) $\{0^n 1^m \mid n \leq m\}$.

f) $\{0^n 1^{2n} \mid n \geq 1\}$.

! **Esercizio 4.1.2** Dimostrate che i seguenti linguaggi non sono regolari.

* a) $\{0^n \mid n \text{ è un quadrato perfetto}\}$.

b) $\{0^n \mid n \text{ è un cubo perfetto}\}$.

c) $\{0^n \mid n \text{ è una potenza di } 2\}$.

d) L'insieme delle stringhe di 0 e 1 la cui lunghezza è un quadrato perfetto.

e) L'insieme delle stringhe di 0 e 1 della forma ww, formate cioè da una stringa ripetuta.

f) L'insieme delle stringhe di 0 e 1 della forma ww^R, cioè formate da una stringa seguita dalla stessa, invertita (per una definizione formale dell'inversione di una stringa cfr. il Paragrafo 4.2.2).

g) L'insieme delle stringhe di 0 e 1 della forma $w\overline{w}$, dove $\overline{w}$ è formata da w sostituendo gli 0 con 1, e viceversa; per esempio $\overline{011} = 100$, quindi 011100 appartiene al linguaggio.

h) L'insieme delle stringhe della forma $w1^n$, dove w è una stringa di 0 e 1 di lunghezza n.

!! **Esercizio 4.1.3** Dimostrate che i seguenti linguaggi non sono regolari.

a) L'insieme delle stringhe di 0 e 1 che cominciano per 1 e che, interpretate come numero intero, danno un numero primo.

b) L'insieme delle stringhe della forma $0^i 1^j$ tali che il massimo comun divisore di i e j è 1.

! **Esercizio 4.1.4** Quando proviamo ad applicare il *pumping lemma* a un linguaggio regolare "vince l'avversario", e non siamo in grado di portare a termine la dimostrazione. Spiegate dove fallisce la prova per i seguenti linguaggi:

* a) l'insieme vuoto

* b) $\{00, 11\}$

* c) $(\mathbf{00} + \mathbf{11})^*$

d) $\mathbf{01^*0^*1}$.

4.2 Proprietà di chiusura dei linguaggi regolari

In questo paragrafo dimostreremo diversi teoremi della forma "se determinati linguaggi sono regolari e da essi si forma un linguaggio L per mezzo di certe operazioni (per esempio L è l'unione di due linguaggi regolari), allora anche L è regolare". Spesso tali teoremi sono detti *proprietà di chiusura* perché mostrano che la classe dei linguaggi regolari è chiusa rispetto all'operazione citata. Le proprietà di chiusura esprimono l'idea che quando uno o più linguaggi sono regolari, lo sono anche altri linguaggi correlati. Sono inoltre un esempio interessante di come le rappresentazioni equivalenti dei linguaggi regolari (automi ed espressioni regolari) si rinforzino a vicenda nella nostra comprensione della classe dei linguaggi. Quando si tratta di dimostrare una proprietà di chiusura, infatti, una rappresentazione è spesso di gran lunga migliore delle altre. Ecco un riassunto delle principali proprietà di chiusura dei linguaggi regolari:

1. l'unione di due linguaggi regolari è regolare
2. l'intersezione di due linguaggi regolari è regolare
3. il complemento di un linguaggio regolare è regolare
4. la differenza di due linguaggi regolari è regolare
5. l'inversione di un linguaggio regolare è regolare
6. la chiusura (star) di un linguaggio regolare è regolare
7. la concatenazione di linguaggi regolari è regolare
8. un omomorfismo (sostituzione di simboli con stringhe) di un linguaggio regolare è regolare
9. l'omomorfismo inverso di un linguaggio regolare è regolare.

4.2.1 Chiusura dei linguaggi regolari rispetto a operazioni booleane

Le prime proprietà di chiusura riguardano le tre operazioni booleane: unione, intersezione e complementazione.

1. Siano L ed M linguaggi sull'alfabeto Σ. Allora $L \cup M$ è il linguaggio che contiene tutte le stringhe che si trovano in L o in M oppure in entrambi.
2. Siano L ed M linguaggi sull'alfabeto Σ. Allora $L \cap M$ è il linguaggio che contiene tutte le stringhe che si trovano sia in L sia in M.

Che cosa succede se i linguaggi hanno alfabeti differenti

Due linguaggi, L_1 ed L_2, di cui facciamo l'unione o l'intersezione, potrebbero avere alfabeti differenti. Per esempio è possibile che $L_1 \subseteq \{a, b\}^*$, mentre $L_2 \subseteq \{b, c, d\}^*$. Se però un linguaggio L consiste di stringhe con simboli in Σ, allora L può essere pensato come un linguaggio su qualunque alfabeto finito che contenga Σ. Dunque possiamo considerare, per esempio, i due linguaggi citati sopra, L_1 ed L_2, come linguaggi sull'alfabeto $\{a, b, c, d\}$. Il fatto che nessuna delle stringhe di L_1 contenga i simboli c o d è irrilevante, come lo è il fatto che le stringhe di L_2 non contengano a.

Analogamente, se prendiamo il complemento di un linguaggio L che è un sottoinsieme di Σ_1^* per un alfabeto Σ_1, possiamo scegliere di prendere il complemento *rispetto a* un alfabeto Σ_2 che contiene Σ_1. In tal caso il complemento di L sarà $\Sigma_2^* - L$; in altre parole il complemento di L rispetto a Σ_2 include (tra le altre stringhe) tutte quelle stringhe in Σ_2^* che hanno almeno un simbolo che si trova in Σ_2 ma non in Σ_1. Se avessimo preso il complemento di L rispetto a Σ_1, allora nessuna stringa con simboli in $\Sigma_2 - \Sigma_1$ sarebbe stata presente in $\overline{L}$. Quindi, a rigor di termini, bisognerebbe sempre dichiarare l'alfabeto rispetto al quale si complementa. Spesso comunque risulta chiaro quale alfabeto si intende: per esempio se L è definito da un automa, allora la specifica di questo include l'alfabeto. Si parlerà quindi frequentemente del "complemento" senza specificare l'alfabeto.

3. Sia L un linguaggio sull'alfabeto Σ. Allora $\overline{L}$, il *complemento* di L, è l'insieme delle stringhe in Σ^* che non sono in L.

I linguaggi regolari risultano essere chiusi rispetto a tutte e tre le operazioni booleane. Come si vedrà, le dimostrazioni seguono però vie molto diverse.

Chiusura rispetto all'unione

Teorema 4.4 Se L ed M sono linguaggi regolari, allora è regolare anche $L \cup M$.

DIMOSTRAZIONE La dimostrazione è semplice. Poiché L ed M sono regolari, essi sono descritti da espressioni regolari, poniamo $L = L(R)$ e $M = L(S)$. Allora $L \cup M = L(R + S)$ per la definizione dell'operatore $+$ delle espressioni regolari. $\square$

Chiusura rispetto alle operazioni regolari

Dimostrare che i linguaggi regolari sono chiusi rispetto all'unione è stato estremamente facile perché l'unione è una delle tre operazioni che definiscono le espressioni regolari. L'idea del Teorema 4.4 si applica anche alla concatenazione e alla chiusura:

- se L ed M sono linguaggi regolari, anche LM è regolare
- se L è un linguaggio regolare, allora L^* è regolare.

Chiusura rispetto alla complementazione

Il teorema per l'unione è stato facilitato dall'uso delle espressioni regolari per rappresentare linguaggi. Consideriamo ora la complementazione. Di primo acchito è difficile immaginare come si possa trasformare un'espressione regolare in una che definisce il linguaggio complemento; eppure è possibile perché, come vedremo nel Teorema 4.5, possiamo facilmente partire da un DFA per costruire un DFA che accetta il complemento. Partendo da un'espressione regolare possiamo quindi trovarne una per il suo complemento seguendo questa procedura:

1. convertiamo l'espressione regolare in un ϵ-NFA
2. convertiamo l'ϵ-NFA in un DFA per mezzo della costruzione per sottoinsiemi
3. complementiamo gli stati accettanti del DFA
4. ritrasformiamo il DFA per il complemento in un'espressione regolare servendoci della costruzione descritta nei Paragrafi 3.2.1 o 3.2.2.

Teorema 4.5 Se L è un linguaggio regolare sull'alfabeto Σ, allora anche $\overline{L} = \Sigma^* - L$ è regolare.

DIMOSTRAZIONE Sia $L = L(A)$ per un DFA $A = (Q, \Sigma, \delta, q_0, F)$. Allora $\overline{L} = L(B)$, dove B è il DFA $(Q, \Sigma, \delta, q_0, Q - F)$. In altre parole B è esattamente come A, ma gli stati accettanti di A sono diventati stati non accettanti di B, e viceversa. Allora w è in $L(B)$ se e solo se $\hat{\delta}(q_0, w)$ è in $Q - F$, il che succede se e solo se w non è in $L(A)$. □

Si noti che è importante per la dimostrazione svolta che $\hat{\delta}(q_0, w)$ sia sempre definito; vale a dire che non ci siano transizioni mancanti in A. Se ce ne fossero, allora alcune

stringhe potrebbero non condurre ad alcuno stato di A, né accettante né non accettante, e le stringhe verrebbero a mancare sia in $L(A)$ sia in $L(B)$. Fortunatamente, per la definizione data, un DFA ha in ogni stato una transizione su ciascun simbolo di Σ: ogni stringa conduce così a uno stato in F oppure a uno stato in $Q - F$.

Esempio 4.6 Sia A l'automa della Figura 2.14. Ricordiamo che A accetta tutte e sole le stringhe di 0 e 1 che finiscono per 01; nei termini delle espressioni regolari, $L(A) = (\mathbf{0} + \mathbf{1})^*\mathbf{01}$. Il complemento di $L(A)$ è perciò formato da tutte le stringhe di 0 e 1 che *non* finiscono per 01. La Figura 4.2 mostra l'automa per $\{0, 1\}^* - L(A)$. Esso è uguale a quello della Figura 2.14, ma con lo stato accettante reso non accettante e i due stati non accettanti resi accettanti. □

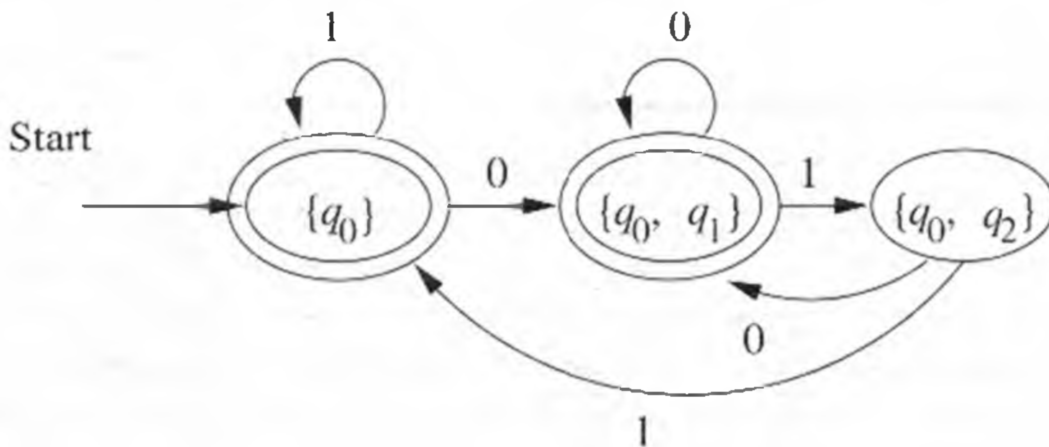


Figura 4.2 DFA che accetta il complemento del linguaggio $(\mathbf{0} + \mathbf{1})^*\mathbf{01}$.

Esempio 4.7 In questo esempio applicheremo il Teorema 4.5 per dimostrare che un certo linguaggio non è regolare. Nell'Esempio 4.2 abbiamo mostrato che il linguaggio L_{eq}, costituito dalle stringhe con un numero uguale di 0 e 1, non è regolare. La dimostrazione consisteva in una semplice applicazione del *pumping lemma*. Ora consideriamo il linguaggio M formato dalle stringhe di 0 e 1 che hanno un numero disuguale di 0 e 1.

Sarebbe difficile usare il *pumping lemma* per mostrare che M non è regolare. Intuitivamente, se partiamo da una stringa w in M, la scomponiamo in $w = xyz$ e "replichiamo" y, potremmo trovare che y stessa è una stringa come 01, con un numero uguale di 0 e 1. Se è così, non esisterà un k tale che xy^kz abbia un numero uguale di 0 e 1, dato che xyz ha un numero disuguale di 0 e 1, e i numeri di 0 e 1 cambiano uniformemente mentre "replichiamo" y. Non possiamo perciò usare il lemma per contraddire l'assunto che M è regolare.

Eppure M non è regolare, e la ragione è che $M = \overline{L}$. Poiché il complemento del complemento è l'insieme da cui siamo partiti, ne consegue anche che $L = \overline{M}$. Se M

fosse regolare, per il Teorema 4.5 anche L sarebbe regolare. Sappiamo però che L *non* lo è; abbiamo dunque una dimostrazione per assurdo che M non è regolare. □

Chiusura rispetto all'intersezione

Consideriamo ora l'intersezione di due linguaggi regolari. Data l'interdipendenza delle tre operazioni booleane, non dobbiamo faticare molto. Avendo modo di svolgere la complementazione e l'unione, possiamo conseguire l'intersezione dei linguaggi L ed M attraverso l'identità

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

In generale l'intersezione di due insiemi è l'insieme degli elementi che non sono nel complemento di uno dei due insiemi. L'osservazione espressa dall'Equazione (4.1) è una delle *leggi di DeMorgan.* L'altra legge è uguale, a patto di scambiare unione e intersezione, ossia: $L \cup M = \overline{\overline{L} \cap \overline{M}}$.

Possiamo d'altra parte compiere una costruzione diretta di un DFA per l'intersezione di due linguaggi regolari. La costruzione, che in sostanza esegue due DFA parallelamente, è utile di per sé. Per esempio è stata usata per costruire l'automa della Fig. 2.3, che rappresentava il "prodotto" delle azioni dei due partecipanti, la banca e il negozio. Il prossimo teorema formalizzerà la *costruzione per prodotto.*

Teorema 4.8 Se L ed M sono linguaggi regolari, allora anche $L \cap M$ è regolare.

DIMOSTRAZIONE Siano L ed M i linguaggi degli automi $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ e $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Osserviamo che il nostro assunto è che gli alfabeti di entrambi gli automi siano uguali; in altre parole, se i due alfabeti sono diversi, Σ è l'unione degli alfabeti di L ed M. Effettivamente, la costruzione per prodotto funziona tanto per gli NFA quanto per i DFA, ma per semplificare al massimo il ragionamento assumiamo che A_L e A_M siano DFA.

Per $L \cap M$ costruiremo un automa A che simuli sia A_L sia A_M. Gli stati di A sono coppie di stati, il primo da A_L e il secondo da A_M. Per stabilire le transizioni di A, supponiamo che A si trovi nello stato (p, q), dove p è lo stato di A_L e q è lo stato di A_M. Se a è il simbolo di input, vediamo come si comporta A_L a fronte dell'input a; poniamo che vada nello stato s. Vediamo anche come si comporta A_M a fronte dell'input a; poniamo che compia una transizione nello stato t. Allora il prossimo stato di A sarà (s, t). In questo modo A ha simulato l'effetto sia di A_L sia di A_M. L'idea è illustrata nella Fig. 4.3.

I dettagli che restano sono semplici. Lo stato iniziale di A è la coppia di stati iniziali di A_L e A_M. Poiché vogliamo che l'automa accetti se e solo se entrambi gli automi accettano, selezioniamo come stati accettanti tutte quelle coppie (p, q) tali che p sia uno

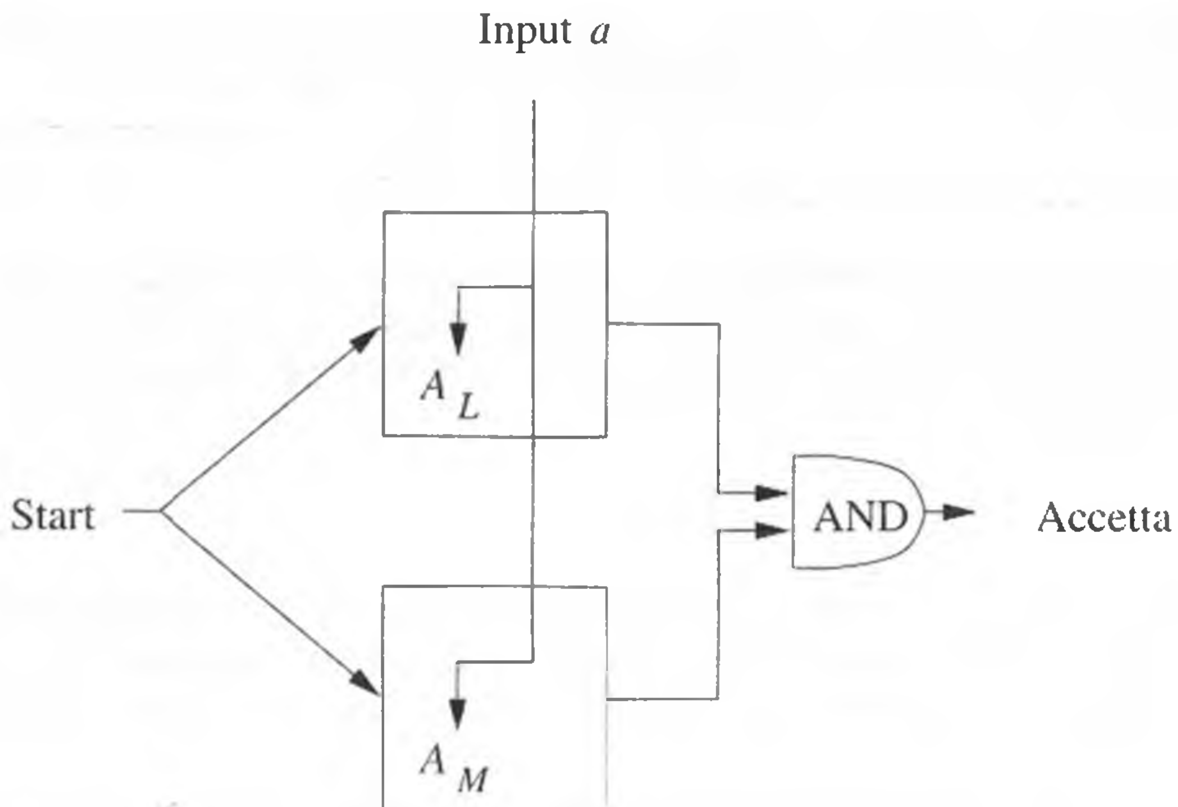


Figura 4.3 Un automa che simula altri due automi e che accetta se e solo se entrambi gli automi accettano.

stato accettante di A_L e q sia uno stato accettante di A_M. In termini formali definiamo:

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M)$$

dove $\delta\big((p, q), a\big) = \big(\delta_L(p, a), \delta_M(q, a)\big)$.

Per capire perché $L(A) = L(A_L) \cap L(A_M)$, osserviamo in primo luogo che grazie a una facile induzione su $|w|$ si dimostra che $\hat{\delta}\big((q_L, q_M), w\big) = \big(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\big)$. Ma A accetta w se e solo se $\hat{\delta}\big((q_L, q_M), w\big)$ è una coppia di stati accettanti. Quindi $\hat{\delta}_L(q_L, w)$ dev'essere in F_L e $\hat{\delta}_M(q_M, w)$ dev'essere in F_M. In altre parole A accetta w se e solo se sia A_L sia A_M accettano w. Di conseguenza A accetta l'intersezione di L ed M. $\square$

Esempio 4.9 Nella Figura 4.4 sono rappresentati due DFA. L'automa della Figura 4.4(a) accetta tutte le stringhe che hanno uno 0, mentre quello della Fig. 4.4(b) accetta tutte le stringhe che hanno un 1. Nella Fig. 4.4(c) vediamo il prodotto dei due automi i cui stati sono etichettati dalla coppia di stati degli automi in (a) e (b).

Si può facilmente dimostrare che tale automa accetta l'intersezione dei primi due linguaggi: le stringhe che hanno sia uno 0 sia un 1. Lo stato pr rappresenta solo la condizione iniziale, in cui non sono stati visti né 0 né 1. Lo stato qr significa che sono stati visti solo

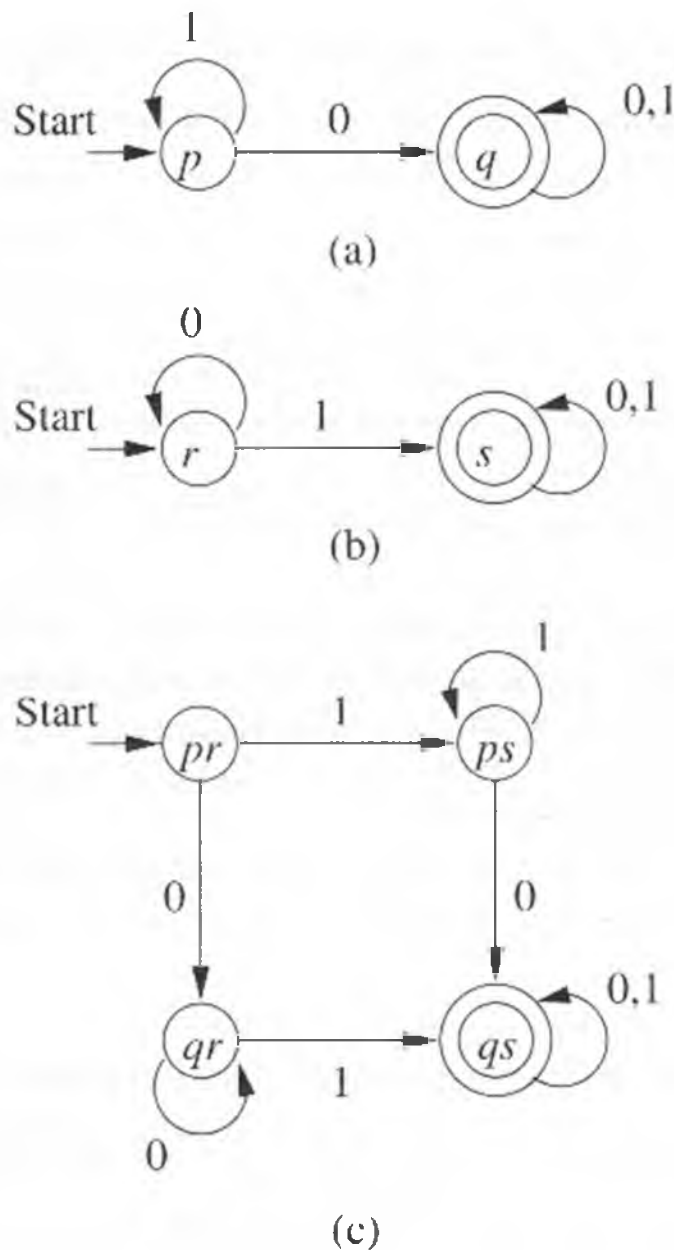


Figura 4.4 La costruzione per prodotto.

0, mentre lo stato ps rappresenta la condizione in cui sono stati visti solo 1. Lo stato accettante qs raffigura la condizione in cui sono stati visti sia 0 sia 1. □

Chiusura rispetto alla differenza

Una terza operazione viene spesso applicata agli insiemi e correlata alle operazioni booleane: la differenza. In termini di linguaggio, $L - M$, la *differenza* di L ed M, è l'insieme delle stringhe che si trovano in L ma non in M. I linguaggi regolari sono chiusi anche rispetto a quest'operazione, e la dimostrazione si desume facilmente dai teoremi appena visti.

Teorema 4.10 Se L ed M sono linguaggi regolari, allora anche $L - M$ è regolare.

DIMOSTRAZIONE Osserviamo che $L - M = L \cap \overline{M}$. Per il Teorema 4.5 $\overline{M}$ è regolare, e per il Teorema 4.8 $L \cap \overline{M}$ è regolare. Di conseguenza $L - M$ è regolare. □

4.2.2 Inversione

L'*inversione* di una stringa $a_1 a_2 \cdots a_n$ è la stringa scritta al contrario, ossia $a_n a_{n-1} \cdots a_1$. Con w^R denotiamo l'inversione della stringa w. Perciò 0010^R è 0100 e $\epsilon^R = \epsilon$.

L'inversione di un linguaggio L, denotata da L^R, è il linguaggio formato dalle inversioni di tutte le sue stringhe. Per esempio, se $L = \{001, 10, 111\}$, allora $L^R = \{100, 01, 111\}$.

Anche l'inversione preserva i linguaggi regolari: se L è un linguaggio regolare, lo è anche L^R. Ci sono due semplici dimostrazioni, una basata sugli automi e una sulle espressioni regolari. Per prima diamo una dimostrazione informale basata sugli automi, lasciando i dettagli al lettore. In seguito dimostreremo il teorema in termini formali servendoci delle espressioni regolari.

Dato un linguaggio L che sia $L(A)$ per un certo automa a stati finiti A, eventualmente non deterministico e con ϵ-transizioni, possiamo costruire un automa per L^R seguendo questa procedura:

1. invertiamo tutti gli archi nel diagramma di transizione di A

2. rendiamo lo stato iniziale di A unico stato accettante per il nuovo automa

3. creiamo un nuovo stato iniziale p_0 con ϵ-transizioni verso tutti gli stati accettanti di A.

Il risultato è un automa che simula A "al contrario", e accetta dunque una stringa w se e solo se A accetta w^R. Dimostriamo ora il teorema dell'inversione in modo formale.

Teorema 4.11 Se L è un linguaggio regolare, lo è anche L^R.

DIMOSTRAZIONE Assumiamo che L sia definito dall'espressione regolare E. La dimostrazione è un'induzione strutturale sulla dimensione di E. Dimostriamo che esiste un'altra espressione regolare E^R tale che $L(E^R) = \big(L(E)\big)^R$; vale a dire che il linguaggio di E^R è l'inversione del linguaggio di E.

BASE Se E è ϵ, $\emptyset$ oppure $\mathbf{a}$, per un simbolo a, allora E^R è uguale a E. In altri termini sappiamo che $\{\epsilon\}^R = \{\epsilon\}$, $\emptyset^R = \emptyset$, e $\{a\}^R = \{a\}$.

INDUZIONE Esistono tre casi, a seconda della forma di E.

1. $E = E_1 + E_2$. Allora $E^R = E_1^R + E_2^R$. La formula si spiega così: l'inversione dell'unione di due linguaggi si ottiene calcolando l'inversione di ciascuno e facendone l'unione.

2. $E = E_1 E_2$. Allora $E^R = E_2^R E_1^R$. Si noti che invertiamo l'ordine dei due linguaggi, oltre ai linguaggi stessi. Per esempio, se $L(E_1) = \{01, 111\}$ e $L(E_2) = \{00, 10\}$, allora $L(E_1 E_2) = \{0100, 0110, 11100, 11110\}$. L'inversione di quest'ultimo linguaggio è
$$\{0010, 0110, 00111, 01111\}$$
Se concateniamo le inversioni di $L(E_2)$ e $L(E_1)$ in quest'ordine, otteniamo
$$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$
cioè lo stesso linguaggio di $\big(L(E_1 E_2)\big)^R$. In generale, se una parola w in $L(E)$ è la concatenazione di w_1 da $L(E_1)$ e w_2 da $L(E_2)$, allora $w^R = w_2^R w_1^R$.

3. $E = E_1^*$. Allora $E^R = (E_1^R)^*$. La giustificazione è che qualunque stringa w in $L(E)$ può essere scritta come $w_1 w_2 \cdots w_n$, dove ogni w_i è in $L(E)$. Ma
$$w^R = w_n^R w_{n-1}^R \cdots w_1^R$$
e ogni w_i^R è in $L(E^R)$, così w^R è in $L\big((E_1^R)^*\big)$. Viceversa, qualunque stringa in $L\big((E_1^R)^*\big)$ è della forma $w_1 w_2 \cdots w_n$, dove ogni w_i è l'inversione di una stringa in $L(E_1)$. L'inversione di questa stringa, $w_n^R w_{n-1}^R \cdots w_1^R$, è dunque una stringa in $L(E_1^*)$, che è $L(E)$. Abbiamo così dimostrato che una stringa è in $L(E)$ se e solo se la sua inversione è in $L\big((E_1^R)^*\big)$.

□

Esempio 4.12 Sia L definito dall'espressione regolare $(\mathbf{0} + \mathbf{1})\mathbf{0}^*$. Allora, per la regola della concatenazione, L^R è il linguaggio di $(\mathbf{0}^*)^R(\mathbf{0}+\mathbf{1})^R$. Se applichiamo la regola della chiusura e dell'unione alle due parti, e poi applichiamo la regola di base secondo la quale le inversioni di $\mathbf{0}$ e $\mathbf{1}$ rimangono immutate, abbiamo che L^R ha l'espressione regolare $\mathbf{0}^*(\mathbf{0} + \mathbf{1})$. □

4.2.3 Omomorfismi

Un *omomorfismo* di stringhe è una funzione che sostituisce a ogni simbolo una particolare stringa.

Esempio 4.13 La funzione h definita da $h(0) = ab$ e $h(1) = \epsilon$ è un omomorfismo. Data una qualunque stringa di 0 e 1, la funzione sostituisce tutti gli 0 con la stringa ab e tutti gli 1 con la stringa vuota. Per esempio h applicato alla stringa 0011 è $abab$. □

In termini formali, se h è un omomorfismo sull'alfabeto Σ e $w = a_1a_2 \cdots a_n$ è una stringa di simboli in Σ, allora $h(w) = h(a_1)h(a_2) \cdots h(a_n)$. Applichiamo cioè h a ogni simbolo di w e concateniamo i risultati in successione. Consideriamo l'omomorfismo h dell'Esempio 4.13. Se $w = 0011$, allora $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\epsilon)(\epsilon) = abab$, come affermato nell'esempio.

È inoltre possibile applicare un omomorfismo a un linguaggio applicandolo a ognuna delle stringhe in esso contenute. In altre parole, se L è un linguaggio sull'alfabeto Σ e h è un omomorfismo su Σ, allora $h(L) = \{h(w) \mid w \text{ è in } L\}$. Per esempio, se L è il linguaggio dell'espressione regolare $\mathbf{10^*1}$, cioè un numero arbitrario di 0 preceduto e seguito da un 1, $h(L)$ è il linguaggio $(\mathbf{ab})^*$. La ragione è che l'omomorfismo h dell'Esempio 4.13 cancella di fatto gli 1, sostituendoli con ϵ, e trasforma ogni 0 in ab. L'idea di applicare un omomorfismo direttamente a un'espressione regolare può essere usata per dimostrare che i linguaggi regolari sono chiusi rispetto a omomorfismi.

Teorema 4.14 Se L è un linguaggio regolare sull'alfabeto Σ e h è un omomorfismo su Σ, allora anche $h(L)$ è regolare.

DIMOSTRAZIONE Sia $L = L(R)$ per un'espressione regolare R. In generale, se E è un'espressione regolare con simboli in Σ, sia $h(E)$ l'espressione che si ottiene sostituendo ogni simbolo a di Σ in E con $h(a)$. Sosteniamo che $h(R)$ definisce il linguaggio $h(L)$.

La dimostrazione è una semplice induzione strutturale: se applichiamo h a ogni sottoespressione E di R, ottenendo $h(E)$, il linguaggio di $h(E)$ è lo stesso che si ottiene se si applica h al linguaggio $L(E)$. In termini formali $L\big(h(E)\big) = h\big(L(E)\big)$.

BASE Se E è ϵ oppure $\emptyset$, allora $h(E)$ è uguale a E, dato che h non ha effetto sulla stringa ϵ o sul linguaggio $\emptyset$. Di conseguenza $L\big(h(E)\big) = L(E)$. Se d'altra parte E è $\emptyset$ o ϵ, allora $L(E)$ o non contiene stringhe o contiene una stringa priva di simboli. Perciò si ha $h\big(L(E)\big) = L(E)$ in ambedue i casi. Possiamo concludere che $L\big(h(E)\big) = L(E) = h\big(L(E)\big)$.

L'unico altro caso di base possibile si ha se $E = \mathbf{a}$ per un simbolo a in Σ. In questo caso $L(E) = \{a\}$, dunque $h\big(L(E)\big) = \{h(a)\}$. Inoltre $h(E)$ è l'espressione regolare formata dalla stringa di simboli $h(a)$. Quindi anche $L\big(h(E)\big)$ è $\{h(a)\}$, e concludiamo che $L\big(h(E)\big) = h\big(L(E)\big)$.

INDUZIONE Ci sono tre casi, tutti semplici. Prendiamo in esame solo il caso dell'unione, con $E = F + G$. Il modo in cui applichiamo gli omomorfismi alle espressioni regolari garantisce che $h(E) = h(F+G) = h(F)+h(G)$. Sappiamo inoltre che $L(E) = L(F) \cup L(G)$ e

$$L\big(h(E)\big) = L\big(h(F) + h(G)\big) = L\big(h(F)\big) \cup L\big(h(G)\big) \tag{4.2}$$

per la definizione di "+" nelle espressioni regolari. Infine

$$h(L(E)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G)) \tag{4.3}$$

poiché h viene applicato a ogni singola stringa del linguaggio. Ora possiamo invocare l'ipotesi induttiva per asserire che $L(h(F)) = h(L(F))$ e $L(h(G)) = h(L(G))$. In questo modo le espressioni finali in (4.2) e (4.3) sono equivalenti, e dunque lo sono anche i loro rispettivi primi termini. Quindi $L(h(E)) = h(L(E))$.

Non dimostreremo i casi in cui l'espressione E è una concatenazione o una chiusura, in quanto le idee da applicare sono analoghe a quanto visto sopra. La conclusione è che $L(h(R))$ coincide con $h(L(R))$; cioè l'applicazione dell'omomorfismo h all'espressione regolare per il linguaggio L risulta in un'espressione regolare che definisce il linguaggio $h(L)$. □

4.2.4 Omomorfismi inversi

Gli omomorfismi si possono applicare anche in senso inverso, e anche così preservano i linguaggi regolari. In altre parole supponiamo che h sia un omomorfismo da un alfabeto Σ verso stringhe di un altro (eventualmente lo stesso) alfabeto T.[2] Sia L un linguaggio sull'alfabeto T. Allora $h^{-1}(L)$, detta la *controimmagine* di L rispetto ad h, è l'insieme delle stringhe w in Σ^* tali che $h(w)$ sia in L. La Figura 4.5 mostra l'effetto di un omomorfismo su un linguaggio L nella parte (a) e l'effetto di un omomorfismo inverso nella parte (b).

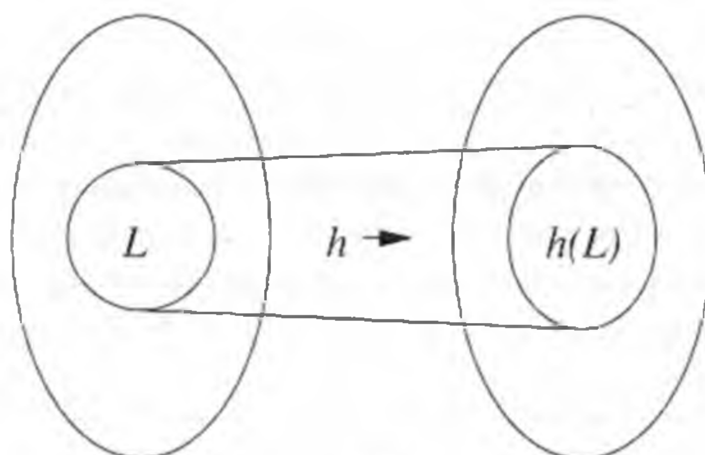
Esempio 4.15 Sia L il linguaggio dell'espressione regolare $(\mathbf{00} + \mathbf{1})^*$; L consiste di tutte le stringhe di 0 e 1 tali che tutti gli 0 occorrono in coppie adiacenti. Dunque 0010011 e 10000111 sono in L, ma non 000 e 10100.

Sia h l'omomorfismo definito da $h(a) = 01$ e $h(b) = 10$. Sosteniamo che $h^{-1}(L)$ è il linguaggio dell'espressione regolare $(\mathbf{ba})^*$, ossia le stringhe formate da ripetizioni di ba. Dimostreremo che $h(w)$ è in L se e solo se w è della forma $baba \cdots ba$.

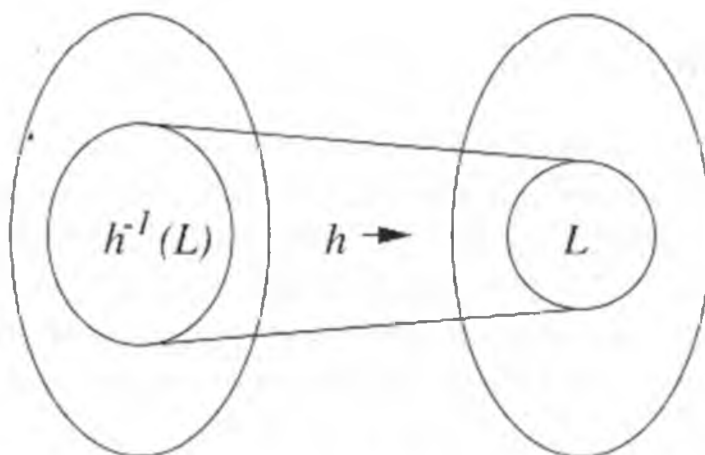
(Se) Supponiamo che w sia formato da n ripetizioni di ba per $n \geq 0$. Notiamo che $h(ba) = 1001$, così $h(w)$ è formato da n ripetizioni di 1001. Dato che 1001 è composto da due 1 e da una coppia di 0, sappiamo che 1001 è in L. Qualunque ripetizione di 1001 è perciò formata da segmenti 1 e 00, e si trova in L. Di conseguenza $h(w)$ è in L.

(Solo-se) Dobbiamo ora assumere che $h(w)$ sia in L e mostrare che w è della forma $baba \cdots ba$. Esistono quattro condizioni per le quali una stringa *non* è di tale forma, e mostreremo che se una di esse è valida allora $h(w)$ non è in L. In altre parole dimostriamo il contronominale dell'enunciato che ci proponevamo di dimostrare.

[2] T va letta come la maiuscola del tau greco, la lettera successiva a sigma.



(a)



(b)

Figura 4.5 Un omomorfismo applicato in avanti e all'indietro.

1. Se w comincia per a, allora $h(w)$ comincia per 01. Ciò significa che ha uno 0 isolato e non si trova in L.
2. Se w finisce per b, allora $h(w)$ finisce per 10, e ancora una volta c'è uno 0 isolato in $h(w)$.
3. Se w ha due a consecutivi, allora $h(w)$ ha una sottostringa 0101. Anche in questo caso troviamo uno 0 isolato in w.
4. Analogamente, se w ha due b consecutivi, allora $h(w)$ ha una sottostringa 1010 e quindi uno 0 isolato.

Perciò, quando risulta valido uno dei casi enunciati sopra, $h(w)$ non è in L. Se escludiamo i casi in cui almeno uno dei punti da (1) a (4) è valido, ne consegue che w è della forma

$baba \cdots ba$. Per capirne la ragione assumiamo che nessuno dei punti da (1) a (4) sia valido. Allora (1) indica che w deve iniziare per b e (2) indica che w finisce per a. Gli enunciati (3) e (4) indicano che a e b devono alternarsi in w. Quindi l'OR logico dei punti da (1) a (4) equivale all'enunciato "w non è della forma $baba \cdots ba$". Abbiamo dimostrato che l'OR dei punti da (1) a (4) implica che $h(w)$ non è in L, e questo è proprio il contronominale dell'enunciato che volevamo: "se $h(w)$ è in L, allora w è della forma $baba \cdots ba$". □

Ora dimostreremo che anche l'omomorfismo inverso di un linguaggio regolare è regolare; successivamente dimostreremo come usare tale teorema.

Teorema 4.16 Se h è un omomorfismo dall'alfabeto Σ verso l'alfabeto T, ed L è un linguaggio regolare su T, allora anche $h^{-1}(L)$ è un linguaggio regolare.

DIMOSTRAZIONE La dimostrazione parte da un DFA A per L. Da A e h costruiamo un DFA per $h^{-1}(L)$ usando lo schema suggerito dalla Fig. 4.6. Il DFA usa gli stati di A, ma traduce i simboli di input in conformità ad h prima di prendere decisioni sullo stato successivo.

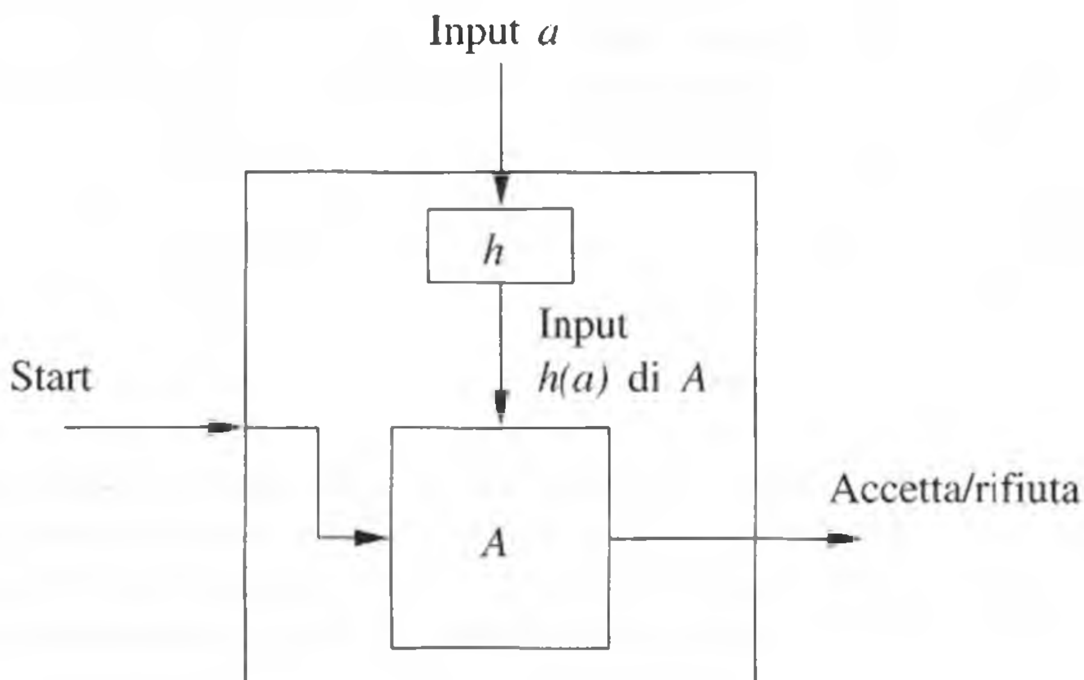


Figura 4.6 Il DFA per $h^{-1}(L)$ applica h al suo input e poi simula il DFA per L.

In termini formali poniamo che L sia $L(A)$ per un DFA $A = (Q, T, \delta, q_0, F)$. Definiamo un DFA

$$B = (Q, \Sigma, \gamma, q_0, F)$$

dove la funzione di transizione γ è definita dalla regola $\gamma(q, a) = \hat{\delta}(q, h(a))$. In altre parole la transizione che B compie su input a è il risultato della sequenza di transizioni

che A compie sulla stringa di simboli $h(a)$. Ricordiamo che $h(a)$ può essere ϵ, un solo simbolo oppure molti, ma $\hat{\delta}$ è definita in modo da trattare correttamente tutti questi casi.

Con una semplice induzione su $|w|$ mostriamo che $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$. Poiché gli stati accettanti di A e B sono uguali, B accetta w se e solo se A accetta $h(w)$. Detto in altro modo, B accetta esattamente le stringhe w che si trovano in $h^{-1}(L)$. □

Esempio 4.17 In quest'esempio useremo l'omomorfismo inverso e alcune altre proprietà di chiusura degli insiemi regolari per dimostrare una strana proprietà degli automi a stati finiti. Supponiamo di imporre a un DFA di visitare ogni stato almeno una volta per accettare un input. Più precisamente supponiamo che $A = (Q, \Sigma, \delta, q_0, F)$ sia un DFA e che siamo interessati al linguaggio L di tutte le stringhe w in Σ^* tali che $\hat{\delta}(q_0, w)$ è in F, con la proprietà aggiuntiva che per ogni q in Q esiste un prefisso x_q di w tale che $\hat{\delta}(q_0, x_q) = q$. L è regolare? Possiamo dimostrare che lo è, ma la costruzione è complessa.

In primo luogo partiamo dal linguaggio M che è $L(A)$, ossia le stringhe che A accetta nella maniera già descritta, senza considerare quali stati vengono visitati durante l'elaborazione dell'input. Dal momento che la definizione di L pone un'ulteriore condizione sulle stringhe di $L(A)$, osserviamo che $L \subseteq M$. La dimostrazione che L è regolare comincia facendo uso di un omomorfismo inverso per inserire gli stati di A nei simboli di input. Più esattamente definiamo un nuovo alfabeto T consistente di simboli che possiamo pensare come terne $[paq]$, dove:

1. p e q sono stati in Q
2. a è un simbolo in Σ
3. $\delta(p, a) = q$.

In altre parole possiamo considerare i simboli in T come rappresentazioni delle transizioni dell'automa A. Sottolineiamo che la notazione $[paq]$ non è la concatenazione di tre simboli, ma il nostro modo di esprimerne uno solo. Avremmo potuto attribuirgli come nome una lettera singola, ma sarebbe stato difficile descrivere la sua relazione con p, q e a.

Definiamo a questo punto l'omomorfismo $h([paq]) = a$ per ogni p, a e q. In altre parole h rimuove i componenti di stato da ognuno dei simboli di T lasciando soltanto il simbolo di Σ. Il primo passo per mostrare che L è regolare è la costruzione del linguaggio $L_1 = h^{-1}(M)$. Dato che M è regolare, lo è anche L_1 per il Teorema 4.16. Le stringhe di L_1 sono proprio le stringhe di M, in cui a ogni simbolo è associata una coppia di stati che rappresenta una transizione.

A titolo esemplificativo consideriamo ora l'automa a due stati della Figura 4.4(a). L'alfabeto Σ è $\{0, 1\}$ e l'alfabeto T consiste dei quattro simboli $[p0q]$, $[q0q]$, $[p1p]$ e $[q1q]$. Il simbolo $[p0q]$, per esempio, si spiega con la transizione dallo stato p allo stato q

su input 0. Poiché 101 è una stringa accettata dall'automa, se vi applichiamo h^{-1} avremo $2^3 = 8$ stringhe, di cui sono due esempi $[p1p][p0q][q1q]$ e $[q1q][q0q][p1p]$.

Costruiamo ora L da L_1 ricorrendo a una serie di ulteriori operazioni che preservano i linguaggi regolari. Il primo obiettivo è eliminare tutte le stringhe di L_1 che non trattano correttamente gli stati. Possiamo cioè interpretare il simbolo $[paq]$ dicendo che l'automa si trovava nello stato p, ha letto l'input a ed è quindi entrato nello stato q. Per essere considerata come una computazione accettante di A, la sequenza di simboli deve soddisfare tre condizioni:

1. il primo stato nel primo simbolo deve essere q_0, lo stato iniziale di A

2. ogni transizione deve riprendere dal punto in cui termina la precedente, ossia il primo stato in un simbolo deve coincidere con il secondo stato del simbolo che lo precede

3. il secondo stato dell'ultimo simbolo deve essere in F; poiché sappiamo che ogni stringa in L_1 proviene da una stringa accettata da A, questa condizione sarà garantita una volta che avremo applicato i punti (1) e (2).

Lo schema della costruzione di L è illustrato nella Figura 4.7.

Per garantire la condizione (1) facciamo l'intersezione di L_1 con l'insieme delle stringhe che cominciano con un simbolo della forma $[q_0aq]$ per un simbolo a e uno stato q. Sia dunque E_1 l'espressione $[q_0a_1q_1] + [q_0a_2q_2] + \cdots$, dove le coppie a_iq_i percorrono tutte le coppie in $\Sigma \times Q$ tali che $\delta(q_0, a_i) = q_i$. Sia poi $L_2 = L_1 \cap L(E_1T^*)$. Dato che E_1T^* è un'espressione regolare che denota tutte le stringhe in T^* che cominciano con lo stato iniziale (possiamo considerare T nell'espressione regolare come la somma dei suoi simboli), L_2 è l'insieme delle stringhe formate applicando h^{-1} al linguaggio M e che hanno lo stato iniziale come primo componente del suo primo simbolo. Esse soddisfano quindi la condizione (1).

Per quanto riguarda il punto (2) la via più facile è sottrarre da L_2 (usando l'operazione di differenza di insiemi) tutte le stringhe che la violano. Sia E_2 l'espressione regolare consistente della somma (unione) della concatenazione di tutte le coppie di simboli che non riescono a combinarsi, ossia le coppie della forma $[paq][rbs]$ dove $q \neq r$. Allora $T^*E_2T^*$ è un'espressione regolare che denota tutte le stringhe che non soddisfano la condizione (2).

Possiamo ora definire $L_3 = L_2 - L(T^*E_2T^*)$. Le stringhe di L_3 soddisfano la condizione (1) perché le stringhe in L_2 devono cominciare con il simbolo iniziale. Soddisfano inoltre la condizione (2), in quanto la sottrazione di $L(T^*E_2T^*)$ rimuove qualunque stringa che la violi. Soddisfano infine la condizione (3), ossia che l'ultimo stato sia accettante. Siamo infatti partiti solo con le stringhe in M, che conducono tutte all'accettazione da parte di A. L'effetto è che L_3 consiste di tutte le stringhe in M in cui gli stati di una

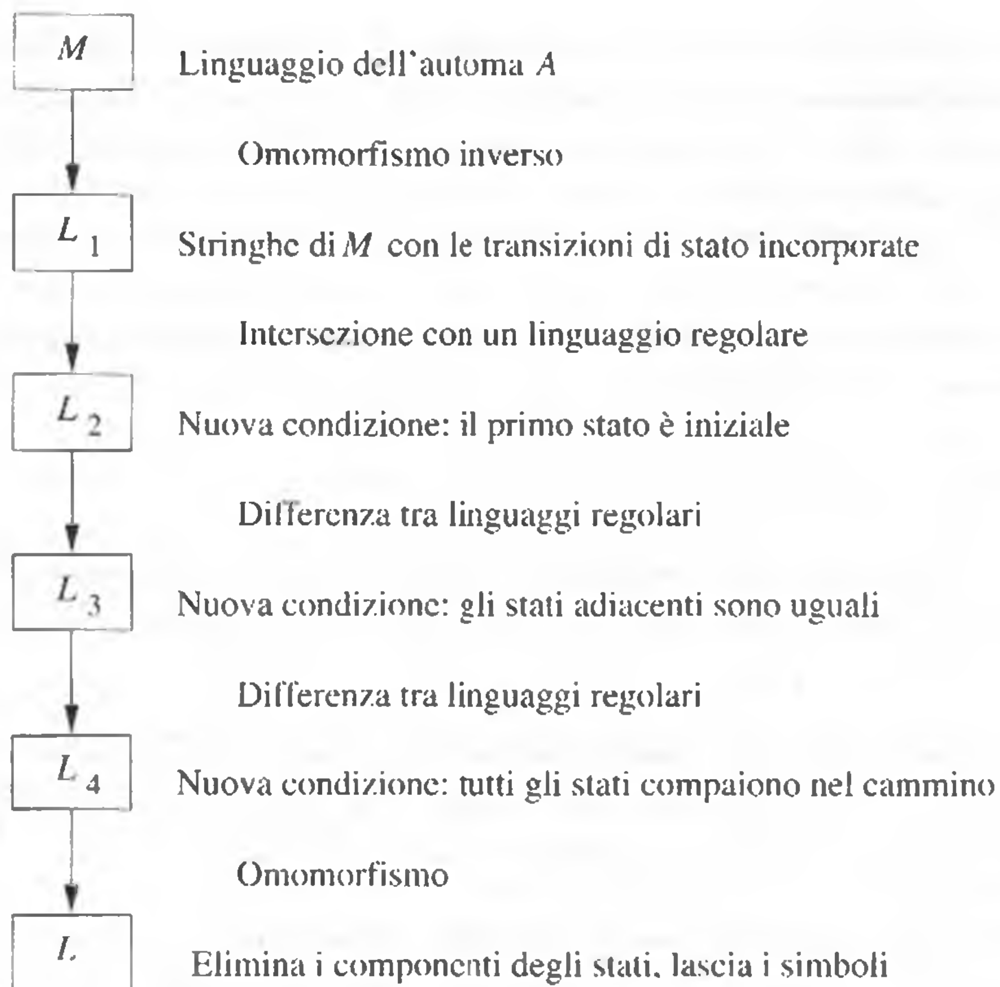


Figura 4.7 La costruzione del linguaggio L dal linguaggio M applicando operazioni che preservano la regolarità dei linguaggi.

computazione accettante sono incorporati come parte dei simboli. Notiamo che L_3 è regolare perché ottenuto a partire dal linguaggio regolare M con l'applicazione di operazioni – omomorfismo inverso, intersezione e differenza di insiemi – che applicate a insiemi regolari danno come risultato insiemi regolari.

Ricordiamo che il nostro obiettivo era accettare solo le stringhe in M che visitano ogni stato nella computazione accettante. Possiamo imporre tale condizione per mezzo di ulteriori applicazioni dell'operatore di differenza di insiemi. In altre parole, per ogni stato q, sia E_q l'espressione regolare che somma tutti i simboli in T tali che q non compaia né nella prima né nell'ultima posizione. Se sottraiamo tutti i linguaggi $L(E_q^*)$ da L_3, otteniamo le stringhe che sono una computazione accettante di A e che visitano lo stato q almeno una volta. Se sottraiamo da L_3 tutti i linguaggi $L(E_q^*)$ per q in Q, otteniamo le computazioni accettanti di A che visitano tutti gli stati. Per il Teorema 4.10 questo linguaggio, che chiameremo L_4, è regolare.

Nell'ultimo passo costruiamo L da L_4 sbarazzandoci dei componenti di stato. In altre

parole $L = h(L_4)$. L è l'insieme delle stringhe in Σ^* che sono accettate da A e che visitano ogni stato di A almeno una volta durante la loro accettazione. Poiché i linguaggi regolari sono chiusi per omomorfismo, concludiamo che L è regolare. □

4.2.5 Esercizi

Esercizio 4.2.1 Supponiamo che h sia l'omomorfismo dall'alfabeto $\{0, 1, 2\}$ verso l'alfabeto $\{a, b\}$ definito da: $h(0) = a$, $h(1) = ab$, e $h(2) = ba$.

* a) Che cos'è $h(0120)$?

b) Che cos'è $h(21120)$?

* c) Se L è il linguaggio $L(\mathbf{01^*2})$, che cos'è $h(L)$?

d) Se L è il linguaggio $L(\mathbf{0} + \mathbf{12})$, che cos'è $h(L)$?

* e) Supponiamo che L sia il linguaggio $\{ababa\}$, ossia il linguaggio formato dalla sola stringa $ababa$. Che cos'è $h^{-1}(L)$?

! f) Se L è il linguaggio $L\big(\mathbf{a}(\mathbf{ba})^*\big)$, che cos'è $h^{-1}(L)$?

*! **Esercizio 4.2.2** Se L è un linguaggio e a un simbolo, allora L/a, il *quoziente* di L e a, è l'insieme delle stringhe w tali che wa è in L. Per esempio, se $L = \{a, aab, baa\}$, allora $L/a = \{\epsilon, ba\}$. Dimostrate che, se L è regolare, lo è anche L/a. *Suggerimento*: partite da un DFA per L e considerate l'insieme di stati accettanti.

! **Esercizio 4.2.3** Se L è un linguaggio e a un simbolo, allora $a\backslash L$ è l'insieme delle stringhe w tali che aw è in L. Per esempio, se $L = \{a, aab, baa\}$, allora $a\backslash L = \{\epsilon, ab\}$. Dimostrate che se L è regolare, lo è anche $a\backslash L$. *Suggerimento*: ricordate che i linguaggi regolari sono chiusi rispetto all'inversione e all'operazione di quoziente dell'Esercizio 4.2.2.

! **Esercizio 4.2.4** Quale delle seguenti identità è vera?

a) $(L/a)a = L$ (il membro sinistro rappresenta la concatenazione dei linguaggi L/a e $\{a\}$).

b) $a(a\backslash L) = L$ (anche qui si intende la concatenazione con $\{a\}$, questa volta a sinistra).

c) $(La)/a = L$.

d) $a\backslash(aL) = L$.

Esercizio 4.2.5 L'operazione dell'Esercizio 4.2.3 s'intende a volte come una "derivata", e $a\backslash L$ è scritta $\frac{dL}{da}$. Tali derivate si applicano alle espressioni regolari in modo simile a quello in cui le normali derivate si applicano alle espressioni aritmetiche. Perciò, se R è un'espressione regolare, useremo $\frac{dR}{da}$ con lo stesso significato di $\frac{dL}{da}$, se $L = L(R)$.

a) Mostrate che $\frac{d(R+S)}{da} = \frac{dR}{da} + \frac{dS}{da}$.

*! b) Scrivete la regola per la "derivata" di RS. *Suggerimento*: bisogna considerare due casi: se $L(R)$ contiene o no ϵ. Questa regola non è esattamente uguale alla regola del prodotto per le normali derivate, ma è simile.

! c) Scrivete la regola per la derivata di una chiusura, ossia $\frac{d(R^*)}{da}$.

d) Usate le regole dei punti (a)–(c) per trovare le derivate dell'espressione regolare $(\mathbf{0} + \mathbf{1})^*\mathbf{011}$ rispetto a 0 e 1.

* e) Caratterizzate i linguaggi L per cui $\frac{dL}{d0} = \emptyset$.

*! f) Caratterizzate i linguaggi L per cui $\frac{dL}{d0} = L$.

! **Esercizio 4.2.6** Mostrate che i linguaggi regolari sono chiusi rispetto alle seguenti operazioni.

a) $\mathit{min}(L) = \{w \mid w$ è in L, ma nessun prefisso proprio di w è in $L\}$.

b) $\mathit{max}(L) = \{w \mid w$ è in L e per nessun x diverso da ϵ, wx è in $L\}$.

c) $\mathit{init}(L) = \{w \mid$ per qualche x, wx è in $L\}$.

Suggerimento: come nell'Esercizio 4.2.2, la strada più facile è partire da un DFA e compiere una costruzione per ottenere il linguaggio desiderato.

! **Esercizio 4.2.7** Se $w = a_1a_2\cdots a_n$ e $x = b_1b_2\cdots b_n$ sono stringhe della stessa lunghezza, definite $\mathit{alt}(w,x)$ come la stringa in cui si alternano i simboli di w e x, a partire da w, ossia $a_1b_1a_2b_2\cdots a_nb_n$. Se L ed M sono linguaggi, definite $\mathit{alt}(L,M)$ come l'insieme delle stringhe della forma $\mathit{alt}(w,x)$, dove w è una stringa qualsiasi in L e x una stringa qualsiasi della stessa lunghezza in M. Dimostrate che se L ed M sono regolari lo è anche $\mathit{alt}(L,M)$.

*!! **Esercizio 4.2.8** Sia L un linguaggio. Definite *half*(L) come l'insieme delle prime metà delle stringhe in L, ossia $\{w \mid$ per un x tale che $|x| = |w|$, abbiamo wx in $L\}$. Per esempio, se $L = \{\epsilon, 0010, 011, 010110\}$, allora *half*$(L) = \{\epsilon, 00, 010\}$. Notiamo che le stringhe di lunghezza dispari non contribuiscono a *half*(L). Dimostrate che se L è un linguaggio regolare lo è anche *half*(L).

!! **Esercizio 4.2.9** Possiamo estendere l'Esercizio 4.2.8 ad alcune funzioni che stabiliscono quanta parte della stringa prendiamo. Se f è una funzione su interi, definite il linguaggio $f(L) = \{w \mid$ per un x con $|x| = f(|w|)$, abbiamo wx in $L\}$. Per esempio l'operazione *half* corrisponde alla funzione identità $f(n) = n$, dato che *half*(L) è definito da $|x| = |w|$. Mostrate che se L è un linguaggio regolare allora lo è anche $f(L)$, se f è una delle seguenti funzioni:

a) $f(n) = 2n$ (il primo terzo di ogni stringa)

b) $f(n) = n^2$ (cioè la parte presa è di lunghezza uguale alla radice quadrata di quanto si omette)

c) $f(n) = 2^n$ (cioè quanto si prende è di lunghezza uguale al logaritmo di quanto si omette).

!! **Esercizio 4.2.10** Supponiamo che L sia un qualunque linguaggio, non necessariamente regolare, il cui alfabeto è $\{0\}$; ossia le stringhe di L consistono solo di 0. Dimostrate che L^* è regolare. *Suggerimento*: a prima vista il teorema sembra assurdo, eppure un esempio aiuterà a capirlo. Consideriamo il linguaggio $L = \{0^i \mid i$ è primo$\}$, che sappiamo non essere regolare dall'Esempio 4.3. Poiché sia 2 sia 3 sono numeri primi, le stringhe 00 e 000 sono in L. Di conseguenza, se $j \geq 2$ possiamo mostrare che 0^j è in L^*. Se j è pari, usate $j/2$ copie di 00; se j è dispari, usate una copia di 000 e $(j-3)/2$ copie di 00. Dunque $L^* = \mathbf{000}^*$.

!! **Esercizio 4.2.11** Mostrate che i linguaggi regolari sono chiusi rispetto alla seguente operazione: *cycle*$(L) = \{w \mid$ possiamo scrivere w come $w = xy$ tale che yx è in $L\}$. Per esempio, se $L = \{01, 011\}$, allora *cycle*$(L) = \{01, 10, 011, 110, 101\}$. *Suggerimento*: partite da un DFA per L e costruite un ϵ-NFA per *cycle*(L).

!! **Esercizio 4.2.12** Sia $w_1 = a_0a_0a_1$ e $w_i = w_{i-1}w_{i-1}a_i$ per ogni $i > 1$. Per esempio $w_3 = a_0a_0a_1a_0a_0a_1a_2a_0a_0a_1a_0a_0a_1a_2a_3$. L'espressione regolare più breve per il linguaggio $L_n = \{w_n\}$, vale a dire il linguaggio consistente dell'unica stringa w_n, è la stringa w_n stessa, e la lunghezza di quest'espressione è $2^{n+1} - 1$. Se però ammettiamo l'operatore di intersezione possiamo scrivere un'espressione per L_n la cui lunghezza è $O(n^2)$. Trovate l'espressione. *Suggerimento*: trovate n linguaggi, ciascuno con espressioni regolari di lunghezza $O(n)$, la cui intersezione sia L_n.

! **Esercizio 4.2.13** Possiamo usare le proprietà di chiusura come ausilio per dimostrare che certi linguaggi non sono regolari. Partite dalla constatazione che il linguaggio

$$L_{0^n1^n} = \{0^n1^n \mid n \geq 0\}$$

non è un insieme regolare. Dimostrate che i seguenti linguaggi non sono regolari trasformandoli, per mezzo di operazioni che sappiamo mantenere la regolarità, in $L_{0^n1^n}$:

* a) $\{0^i 1^j \mid i \neq j\}$

b) $\{0^n 1^m 2^{n-m} \mid n \geq m \geq 0\}$.

Esercizio 4.2.14 Nel Teorema 4.8 abbiamo descritto la "costruzione per prodotto": da due DFA si costruisce un unico DFA il cui linguaggio è l'intersezione dei linguaggi dei primi due.

a) Mostrate come compiere la costruzione per prodotto sugli NFA (senza ϵ-transizioni).

! b) Mostrate come compiere la costruzione per prodotto sugli ϵ-NFA.

* c) Mostrate come modificare la costruzione per prodotto così che il DFA risultante accetti la differenza dei linguaggi dei due DFA dati.

d) Mostrate come modificare la costruzione per prodotto così che il DFA risultante accetti l'unione dei linguaggi dei due DFA dati.

Esercizio 4.2.15 Nella dimostrazione del Teorema 4.8 abbiamo affermato che poteva essere dimostrato per induzione sulla lunghezza di w che

$$\hat{\delta}\big((q_L, q_M), w\big) = \big(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\big)$$

Scrivete tale dimostrazione.

Esercizio 4.2.16 Completate la dimostrazione del Teorema 4.14 considerando i casi in cui l'espressione E è una concatenazione delle due sottoespressioni e in cui E è la chiusura di un'espressione.

Esercizio 4.2.17 Nel Teorema 4.16 abbiamo omesso una dimostrazione per induzione sulla lunghezza di w che $\hat{\gamma}(q_0, w) = \hat{\delta}\big(q_0, h(w)\big)$. Dimostrate l'enunciato.

4.3 Problemi di decisione per i linguaggi regolari

In questo paragrafo cerchiamo di rispondere ad alcune questioni importanti relative ai linguaggi regolari. Dobbiamo anzitutto stabilire che cos'è una questione inerente a un linguaggio. Poiché di norma un linguaggio è infinito, non possiamo porre questioni che richiedano l'esame dell'insieme di stringhe che lo compongono. Dobbiamo invece presentare il linguaggio per mezzo di una delle rappresentazioni finite che abbiamo sviluppato: DFA, NFA, ϵ-NFA ed espressioni regolari.

Ovviamente un linguaggio così descritto è regolare; di fatto non c'è modo di rappresentare linguaggi del tutto arbitrari. Nei capitoli successivi studieremo rappresentazioni finite di linguaggi non regolari e potremo quindi trattare questioni sui linguaggi in classi

più generali. D'altra parte, per molte delle questioni che porremo, disponiamo di algoritmi solo per la classe dei linguaggi regolari. Se formulate rispetto a notazioni più espressive di quelle introdotte per i linguaggi regolari, cioè notazioni atte a esprimere classi più estese di linguaggi, le stesse questioni diventano "indecidibili" (non esistono algoritmi per risolverle).

Lo studio degli algoritmi per questioni sui linguaggi regolari prende le mosse da un riepilogo dei modi per tradurre una rappresentazione in un'altra dello stesso linguaggio. In particolare ci interessa la complessità in tempo degli algoritmi di conversione. Tratteremo poi alcune questioni fondamentali relative ai linguaggi.

1. Il linguaggio descritto è vuoto?

2. Una certa stringa w appartiene al linguaggio descritto?

3. Due descrizioni date specificano lo stesso linguaggio? Questo problema è conosciuto come il problema dell'*equivalenza* di linguaggi.

4.3.1 Conversioni

Sappiamo già che è possibile convertire ognuna delle quattro rappresentazioni dei linguaggi regolari in una delle altre. Nella Figura 3.1 sono illustrati i passaggi da una rappresentazione all'altra. Per ogni conversione esistono algoritmi; a volte però siamo interessati non solo alla possibilità di convertire, ma anche alla quantità di tempo necessaria. In particolare è importante distinguere gli algoritmi che richiedono tempo esponenziale (in funzione delle dimensioni del problema), e che quindi si possono eseguire solo su istanze relativamente "piccole", da quelli che richiedono tempo lineare, quadratico o polinomiale con grado piccolo rispetto alla dimensione. Questi ultimi algoritmi sono "praticabili", nel senso che ci si può aspettare di eseguirli anche su istanze di dimensione "grande" di un problema. Esamineremo la complessità in tempo di tutte le conversioni discusse.

Conversione da NFA a DFA

Quando convertiamo un NFA o un ϵ-NFA in un DFA, il tempo può essere esponenziale nel numero di stati dell'NFA. Per cominciare, il calcolo dell'ϵ-chiusura di n stati richiede un tempo $O(n^3)$. Da ognuno degli n stati dobbiamo esaminare tutti gli archi etichettati ϵ. Se ci sono n stati, non ci possono essere più di n^2 archi. Una strategia attenta e strutture dati appropriate consentono di esaminare ogni stato in tempo $O(n^2)$. Si può usare un algoritmo per la chiusura transitiva, come quello di Warshall, per calcolare in una volta sola l'intera ϵ-chiusura.[3]

[3]Per una trattazione degli algoritmi per la chiusura transitiva, cfr. A. V. Aho, J. E. Hopcroft, J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1984.

Dopo aver calcolato l'ϵ-chiusura, possiamo costruire il DFA equivalente tramite la costruzione per sottoinsiemi. A priori, il costo dominante è dato dal numero di stati del DFA, che può essere 2^n. Per ogni stato possiamo calcolare le transizioni in tempo $O(n^3)$ consultando le informazioni dell'ϵ-chiusura e la tabella di transizione dell'NFA per ogni simbolo di input. Supponiamo di dover calcolare $\delta(\{q_1, q_2, \ldots, q_k\}, a)$ per il DFA. Ci possono essere fino a n stati raggiungibili da ogni q_i lungo cammini etichettati ϵ, e ogni stato può avere fino a n archi etichettati a. Usando un array con gli stati come indici, calcoliamo l'unione di (fino a) n insiemi di (fino a) n stati in un tempo proporzionale a n^2.

Possiamo così calcolare, per ogni q_i, l'insieme degli stati raggiungibili da q_i lungo cammini etichettati a, eventualmente in presenza di ϵ. Poiché $k \leq n$, dobbiamo trattare al massimo n stati. Per ciascuno calcoliamo gli stati raggiungibili in tempo $O(n^2)$. Il tempo totale speso nel calcolo degli stati raggiungibili è quindi $O(n^3)$. L'unione degli insiemi di stati raggiungibili richiede solo un tempo aggiuntivo $O(n^2)$; possiamo concludere che l'elaborazione di una transizione del DFA richiede un tempo $O(n^3)$.

Si noti che assumiamo costante, cioè indipendente da n, il numero di simboli di input. Per questo motivo, in questa e in altre stime del tempo di esecuzione, non consideriamo tra i fattori il numero di simboli di input. La dimensione dell'alfabeto di input influisce sul fattore costante nascosto nella notazione "O grande", ma niente di più.

Concludiamo che il tempo di esecuzione della conversione da NFA a DFA, incluso il caso di NFA con ϵ-transizioni, è $O(n^3 2^n)$. In pratica il numero di stati generati è di solito molto minore di 2^n, spesso pari a n. Si può dire che il tempo di esecuzione è $O(n^3 s)$, dove s è il numero di stati del DFA.

Conversione da DFA a NFA

Questa conversione è semplice e richiede tempo $O(n)$ per un DFA di n stati. Dobbiamo soltanto modificare la tabella di transizione del DFA ponendo tra parentesi graffe gli stati e, qualora si costruisca un ϵ-NFA, aggiungendo una colonna per ϵ. Poiché assumiamo costante il numero di simboli di input (cioè la larghezza della tabella di transizione), la copia e l'elaborazione della tabella richiedono un tempo $O(n)$.

Conversione da automa a espressione regolare

Esaminando la costruzione del Paragrafo 3.2.1, si nota che a ognuno degli n cicli, dove n è il numero di stati del DFA, la lunghezza delle espressioni regolari generate può quadruplicare. Ciascuna è infatti costruita da quattro espressioni del ciclo precedente. La sola scrittura delle n^3 espressioni può quindi richiedere un tempo $O(n^3 4^n)$. La variante migliorata del Paragrafo 3.2.2 riduce il fattore costante, ma non muta il carattere esponenziale nel caso peggiore.

Anche se non l'abbiamo dimostrato, la stessa costruzione opera con il medesimo tempo di esecuzione se l'input è un NFA o un ϵ-NFA. È importante usare proprio questa costruzione per gli NFA: convertire un NFA in un DFA, e poi convertire questo in espressione regolare, potrebbe richiedere un tempo $O(n^3 4^{n^3 2^n})$, che è doppiamente esponenziale.

Conversione da espressione regolare ad automa

Convertire un'espressione regolare in un ϵ-NFA richiede tempo lineare. Bisogna leggere l'espressione in modo efficiente, con una tecnica che impieghi un tempo $O(n)$ per un'espressione regolare di lunghezza n.[4] Si ottiene un albero sintattico con un nodo per ogni simbolo dell'espressione (le parentesi hanno il solo scopo di guidare il *parser* e possono essere omesse).

Una volta ricavato l'albero dell'espressione, dobbiamo risalirlo costruendo un ϵ-NFA per ogni nodo. Le regole costruttive per la conversione di un'espressione regolare date nel Paragrafo 3.2.3 non aggiungono mai più di due stati e quattro archi per ogni nodo dell'albero. Perciò il numero di stati e di archi dell'ϵ-NFA risultante sono entrambi $O(n)$. Inoltre il lavoro svolto per ogni nodo dell'albero sintattico nel creare tali elementi è costante, purché la funzione che elabora ciascun sottoalbero restituisca puntatori allo stato iniziale e agli stati accettanti del proprio automa.

Possiamo concludere che la costruzione di un ϵ-NFA da un'espressione regolare richiede tempo lineare nella lunghezza dell'espressione. Possiamo eliminare le ϵ-transizioni da un ϵ-NFA di n stati per farne un NFA normale, in tempo $O(n^3)$, senza aumentare il numero di stati. La trasformazione in DFA può comunque richiedere tempo esponenziale.

4.3.2 Verificare se un linguaggio regolare è vuoto

A prima vista la risposta alla domanda "il linguaggio regolare L è vuoto?" sembra ovvia: $\emptyset$ è vuoto, ogni altro linguaggio regolare no. Ma come si è detto all'inizio del Paragrafo 4.3, il problema non viene enunciato con l'elenco esplicito delle stringhe in L; abbiamo invece una rappresentazione di L e dobbiamo decidere se essa denota il linguaggio $\emptyset$.

Se la rappresentazione è un automa finito, il problema equivale a stabilire se esiste un cammino dallo stato iniziale a uno stato accettante. In questo caso il linguaggio non è vuoto, mentre se gli stati accettanti sono separati dallo stato iniziale, il linguaggio è vuoto. Decidere se si può raggiungere uno stato accettante a partire dallo stato iniziale è un caso semplice del problema di raggiungibilità nei grafi, simile in sostanza al calcolo della ϵ-chiusura trattato nel Paragrafo 2.5.3. L'algoritmo si può riassumere nella seguente procedura ricorsiva.

---

[4] I metodi di *parsing* in grado di svolgere l'operazione in tempo $O(n)$ sono trattati in A. V. Aho, R. Sethi, J. D. Ullman, *Compiler Design: Principles, Tools, and Techniques*, Addison-Wesley, 1986.

BASE Lo stato iniziale è sicuramente raggiungibile dallo stato iniziale.

INDUZIONE Se lo stato q è raggiungibile dallo stato iniziale e c'è un arco da q a p con etichetta arbitraria (un simbolo di input o ϵ se l'automa è un ϵ-NFA), anche p è raggiungibile.

In questo modo determiniamo l'insieme degli stati raggiungibili. Se fra loro c'è uno stato accettante, la risposta è "no" (il linguaggio dell'automa *non* è vuoto), altrimenti è "sì". Si noti che il calcolo non richiede tempo superiore a $O(n^2)$ se l'automa ha n stati; di fatto è al massimo proporzionale al numero di archi nel diagramma di transizione dell'automa, che può essere inferiore a n^2 e non può superare $O(n^2)$.

Se partiamo da un'espressione regolare che rappresenta un linguaggio L, anziché da un automa, possiamo trasformare l'espressione in un ϵ-NFA e continuare come sopra. Poiché l'automa ricavato da un'espressione regolare di lunghezza n ha al massimo $O(n)$ stati e transizioni, l'algoritmo richiede un tempo $O(n)$.

D'altra parte possiamo anche stabilire se il linguaggio è vuoto esaminando direttamente l'espressione regolare. Osserviamo anzitutto che, se nell'espressione non compare $\emptyset$, il linguaggio non può essere vuoto. Se invece compare, il linguaggio può esserlo oppure no. Le regole ricorsive che seguono rivelano se un'espressione regolare denota il linguaggio vuoto.

BASE $\emptyset$ denota il linguaggio vuoto; ϵ e **a** no, per qualsiasi simbolo di input a.

INDUZIONE Sia R un'espressione regolare. Si devono considerare quattro casi, che corrispondono ai modi in cui si può costruire R.

1. $R = R_1 + R_2$. In questo caso $L(R)$ è vuoto se e solo se sia $L(R_1)$ sia $L(R_2)$ sono vuoti.

2. $R = R_1 R_2$. In questo caso $L(R)$ è vuoto se e solo se o $L(R_1)$ o $L(R_2)$ è vuoto.

3. $R = R_1^*$. In questo caso $L(R)$ non è vuoto: contiene sempre almeno ϵ.

4. $R = (R_1)$. In questo caso $L(R)$ è vuoto se e solo se $L(R_1)$ è vuoto, dato che sono lo stesso linguaggio.

4.3.3 Appartenenza a un linguaggio regolare

La prossima questione rilevante è se, dati una stringa w e un linguaggio regolare L, w appartenga a L. Mentre w è data esplicitamente, L è rappresentato da un automa o da un'espressione regolare.

Se L è rappresentato da un DFA, l'algoritmo è semplice: si simula l'elaborazione della stringa w da parte del DFA a partire dallo stato iniziale; se il DFA finisce in uno stato accettante, la risposta è "sì", altrimenti è "no". Questo algoritmo è molto veloce.

Se $|w| = n$ e il DFA è rappresentato da una struttura dati appropriata, come un array bidimensionale corrispondente alla tabella di transizione, ogni transizione richiede tempo costante, e il tempo totale è $O(n)$.

Se la rappresentazione di L è di altro tipo, possiamo convertirla in un DFA e procedere come sopra. Questa soluzione può richiedere tempo esponenziale nella dimensione della rappresentazione, pur essendo lineare in $|w|$. Se però la rappresentazione è un NFA o un ϵ-NFA, è più semplice ed efficiente simularlo direttamente, cioè elaborare i simboli di w, uno per volta, tenendo traccia dell'insieme degli stati in cui può trovarsi l'NFA seguendo ogni cammino etichettato da un certo prefisso di w. L'idea è stata presentata nella Figura 2.10.

Se w è lunga n e l'NFA ha s stati, il tempo di esecuzione dell'algoritmo è $O(ns^2)$. Ogni simbolo di input viene elaborato esaminando i successori di ogni elemento dell'insieme precedente di stati, che sono al massimo s. Si deve fare l'unione di non più di s insiemi, ciascuno contenente al massimo s stati, il che richiede tempo $O(s^2)$.

Se l'NFA comprende ϵ-transizioni, prima di avviare la simulazione dobbiamo calcolare l'ϵ-chiusura. Quindi l'elaborazione di ogni simbolo a di input comprende due fasi, ognuna delle quali richiede tempo $O(s^2)$. Prima di tutto troviamo i successori, rispetto al simbolo di input a, degli stati nell'insieme precedente. Poi calcoliamo l'ϵ-chiusura del nuovo insieme. L'insieme da cui avviare la simulazione è l'ϵ-chiusura dello stato iniziale dell'NFA.

Infine, se L è rappresentato da un'espressione regolare di dimensione s, possiamo convertirla in un ϵ-NFA di non più di $2s$ stati in tempo $O(s)$. Si compie poi la simulazione come sopra in tempo $O(ns^2)$ su un input w di lunghezza n.

4.3.4 Esercizi

* **Esercizio 4.3.1** Ideate un algoritmo che decida se un linguaggio regolare L è infinito. *Suggerimento*: per mezzo del *pumping lemma* mostrate che se il linguaggio contiene una stringa di lunghezza maggiore di una certa costante allora deve essere infinito.

Esercizio 4.3.2 Ideate un algoritmo che decida se un linguaggio regolare L contiene almeno cento stringhe.

Esercizio 4.3.3 Sia L un linguaggio regolare sull'alfabeto Σ. Ideate un algoritmo che decida se $L = \Sigma^*$, cioè se contiene tutte le stringhe su quell'alfabeto.

Esercizio 4.3.4 Ideate un algoritmo che decida se due linguaggi regolari L_1 ed L_2 hanno almeno una stringa in comune.

Esercizio 4.3.5 Ideate un algoritmo che decida se, dati due linguaggi regolari L_1 ed L_2 sullo stesso alfabeto Σ, esiste una stringa di Σ^* che non appartiene né a L_1 né a L_2.

4.4 Equivalenza e minimizzazione di automi

A differenza dei problemi trattati in precedenza (se un linguaggio sia vuoto e se una stringa appartenga a un linguaggio), i cui algoritmi sono relativamente semplici, la questione se due descrizioni di linguaggi regolari denotino lo stesso linguaggio comporta un considerevole sforzo intellettuale. In questo paragrafo esaminiamo come stabilire se due descrittori di linguaggi regolari sono *equivalenti*, nel senso che definiscono lo stesso linguaggio. Un risultato importante di questa ricerca è l'esistenza di un modo per minimizzare un DFA, cioè per trovare un DFA equivalente con il numero minimo di stati. Questo DFA si rivela essere in sostanza unico: dati due DFA minimi equivalenti, possiamo sempre rinominare gli stati in modo che i due DFA coincidano.

4.4.1 Verifica dell'equivalenza di stati

Cominciamo da una questione relativa agli stati di un DFA. Lo scopo è capire quando due stati distinti, p e q, possono essere sostituiti da un solo stato che si comporta come p e q. Diciamo che due stati p e q sono *equivalenti* se:

- per ogni stringa di input w, $\hat{\delta}(p, w)$ è uno stato accettante se e solo se $\hat{\delta}(q, w)$ è uno stato accettante.

È quindi impossibile distinguere due stati equivalenti p e q partendo da uno di loro e osservando se una data stringa di input porta a uno stato accettante. Si noti che *non* si richiede che $\hat{\delta}(p, w)$ e $\hat{\delta}(q, w)$ siano lo *stesso* stato, ma solo che siano entrambi accettanti o non accettanti.

Se due stati non sono equivalenti, diciamo che sono *distinguibili*. In altre parole lo stato p è distinguibile da q se esiste almeno una stringa w tale che uno fra $\hat{\delta}(p, w)$ e $\hat{\delta}(q, w)$ è accettante e l'altro no.

Esempio 4.18 Consideriamo il DFA della Figura 4.8, la cui funzione di transizione sarà indicata da δ. Alcune coppie di stati sono evidentemente non equivalenti. Per esempio C e G non sono equivalenti perché uno è accettante e l'altro no. Dunque la stringa vuota li distingue perché $\hat{\delta}(C, \epsilon)$ è accettante e $\hat{\delta}(G, \epsilon)$ no.

Consideriamo gli stati A e G. La stringa ϵ non li distingue perché sono entrambi non accettanti. La stringa 0 non li distingue perché porta rispettivamente in B e in G, entrambi non accettanti. Similmente la stringa 1 non distingue A da G perché porta rispettivamente a F e a E, entrambi non accettanti. D'altra parte 01 distingue A da G perché $\hat{\delta}(A, 01) = C$, $\hat{\delta}(G, 01) = E$ e C è accettante, al contrario di E. Qualsiasi stringa porti da A e da G in stati di cui uno solo è accettante basta per dimostrare che A e G non sono equivalenti.

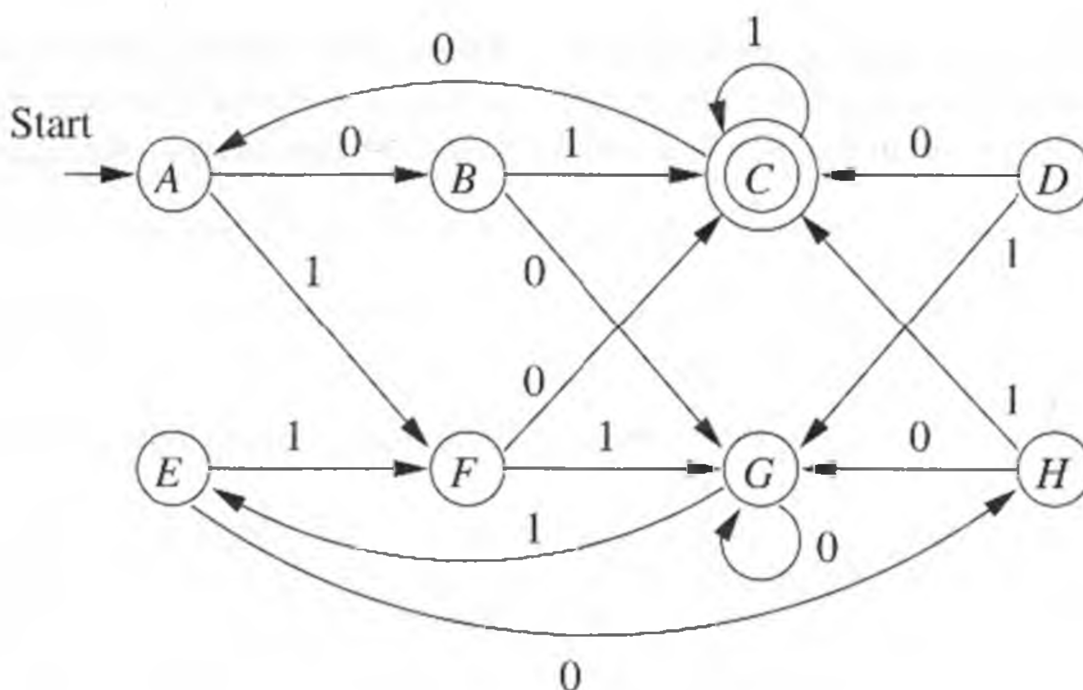


Figura 4.8 Un automa con stati equivalenti.

Consideriamo invece A ed E: non sono accettanti, quindi ϵ non li distingue. A fronte di input 1 entrambi vanno in F. Nessuna stringa che cominci per 1 può quindi distinguerli perché, per ogni stringa x, $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$.

Esaminiamo ora il comportamento di A e di E su input che cominciano per 0: A va in B, E in H. Nessuno dei due nuovi stati è accettante, perciò la stringa 0 da sola non distingue A da E. Anche B e H non sono d'aiuto: su input 1 entrambi vanno in C e su input 0 entrambi vanno in G. Quindi nessun input che cominci per 0 può distinguerli. Concludiamo che nessuna stringa di input distingue A da E: i due stati sono equivalenti. □

Se vogliamo scoprire gli stati equivalenti, dobbiamo fare del nostro meglio per trovare coppie di stati distinguibili. Può sorprendere, ma seguendo l'algoritmo descritto più avanti due stati risultano equivalenti se non siamo riusciti a distinguerli. L'algoritmo, che chiameremo *algoritmo riempi-tabella*, scopre ricorsivamente le coppie di stati distinguibili di un DFA $A = (Q, \Sigma, \delta, q_0, F)$.

BASE Se p è uno stato accettante e q è non accettante, la coppia $\{p, q\}$ è distinguibile.

INDUZIONE Siano p e q due stati tali che, per un simbolo di input a, $r = \delta(p, a)$ e $s = \delta(q, a)$ sono stati che sappiamo essere distinguibili. Allora $\{p, q\}$ è una coppia di stati distinguibili. La regola vale perché dev'esserci una stringa w che distingue r da s; in altre parole uno solo tra $\hat{\delta}(r, w)$ e $\hat{\delta}(s, w)$ è accettante. Ma allora la stringa aw distingue p da q perché $\hat{\delta}(p, aw)$ e $\hat{\delta}(q, aw)$ coincidono rispettivamente con $\hat{\delta}(r, w)$ e $\hat{\delta}(s, w)$.

Esempio 4.19 Eseguiamo l'algoritmo riempi-tabella sul DFA della Figura 4.8. La tabella finale è riportata nella Figura 4.9, in cui un x indica due stati distinguibili e un quadrato vuoto indica le coppie di cui non si è ancora provata l'equivalenza. All'inizio non c'è nessun x.

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	B	C	D	E	F	G

Figura 4.9 Tabella di non equivalenza fra stati.

Per la base, poiché C è l'unico stato accettante, scriviamo x in ogni casella relativa a C. Ora conosciamo qualche coppia distinguibile e possiamo scoprirne altre. Per esempio, poiché $\{C, H\}$ è distinguibile e gli stati E ed F vanno rispettivamente in H e C su input 0, deduciamo che anche $\{E, F\}$ è una coppia distinguibile. Tutti gli x nella Figura 4.9, a eccezione della coppia $\{A, G\}$, si scoprono semplicemente esaminando le transizioni da una coppia di stati su 0 o su 1 e osservando che, per uno dei due simboli, uno stato va in C mentre l'altro no. Possiamo dimostrare che $\{A, G\}$ è distinguibile nella fase successiva: su input 1 i due stati vanno, rispettivamente, in F ed E, e abbiamo già dimostrato che la coppia $\{E, F\}$ è distinguibile.

A questo punto non possiamo più scoprire altre coppie distinguibili. Le tre coppie rimanenti, che sono quindi equivalenti, sono $\{A, E\}$, $\{B, H\}$ e $\{D, F\}$. Esaminiamo ad esempio per quale ragione non possiamo desumere che $\{A, E\}$ è una coppia distinguibile. Su input 0, A ed E vanno rispettivamente in B e H, e $\{B, H\}$ non è stata ancora dichiarata distinguibile. Su input 1 sia A sia E vanno in F: per questa via non c'è modo di distinguerli. Le altre due coppie, $\{B, H\}$ e $\{D, F\}$, non saranno mai distinte perché hanno transizioni identiche sia su 0 sia su 1. L'algoritmo riempi-tabella si ferma quindi alla tabella illustrata nella Figura 4.9, che determina correttamente gli stati equivalenti e quelli distinguibili. □

Teorema 4.20 Se due stati non sono distinti dall'algoritmo riempi-tabella, allora sono equivalenti.

DIMOSTRAZIONE Fissiamo di nuovo un DFA $A = (Q, \Sigma, \delta, q_0, F)$. Supponiamo che l'enunciato sia falso, cioè che esista almeno una coppia di stati $\{p, q\}$ tale che

1. gli stati p e q sono distinguibili, nel senso che esiste una stringa w tale che uno solo tra $\hat{\delta}(p, w)$ e $\hat{\delta}(q, w)$ è accettante, ma

2. l'algoritmo riempi-tabella non distingue p da q.

Una coppia di stati di questo tipo sarà denominata *cattiva*.

Se ci sono coppie cattive, ce n'è una distinta dalla stringa più corta fra tutte quelle che distinguono coppie cattive. Sia $\{p, q\}$ una tale coppia e sia $w = a_1 a_2 \cdots a_n$ una stringa di lunghezza minima fra quelle che distinguono p da q. Allora uno solo fra $\hat{\delta}(p, w)$ e $\hat{\delta}(q, w)$ è accettante.

Innanzitutto osserviamo che w non può essere ϵ perché, se ϵ distingue una coppia di stati, quella coppia è marcata nella base dell'algoritmo riempi-tabella. Perciò $n \geq 1$.

Consideriamo gli stati $r = \delta(p, a_1)$ e $s = \delta(q, a_1)$; r ed s sono distinti dalla stringa $a_2 a_3 \cdots a_n$ perché questa stringa li porta rispettivamente in $\hat{\delta}(p, w)$ e $\hat{\delta}(q, w)$. Ma la stringa che distingue r da s è più corta di ogni stringa che distingue una coppia cattiva: quindi $\{r, s\}$ non può essere una coppia cattiva: l'algoritmo riempi-tabella deve avere scoperto che sono distinguibili.

Ma poiché scopre che $\delta(p, a_1) = r$ è distinguibile da $\delta(q, a_1) = s$, la parte induttiva dell'algoritmo non si arresta prima di aver dedotto che anche p e q sono distinguibili. Abbiamo così contraddetto l'ipotesi che esistano coppie cattive; ma se non ci sono coppie cattive, ogni coppia di stati distinguibili è distinta dall'algoritmo, come volevamo dimostrare. □

4.4.2 Equivalenza di linguaggi regolari

L'algoritmo riempi-tabella fornisce un modo semplice per stabilire se due linguaggi regolari coincidono. Siano L ed M due linguaggi rappresentati, per esempio, uno da un'espressione regolare e l'altro da un NFA. Convertiamo le due rappresentazioni in DFA. Ora consideriamo un DFA i cui stati siano l'unione degli stati dei DFA di L e di M. Questo DFA avrebbe tecnicamente due stati iniziali, ma per quanto concerne la verifica di equivalenza lo stato iniziale è irrilevante. Possiamo quindi sceglierne uno arbitrario.

A questo punto verifichiamo se gli stati iniziali dei due DFA originali sono equivalenti usando l'algoritmo riempi-tabella. Se sono equivalenti, $L = M$, altrimenti $L \neq M$.

Esempio 4.21 Analizziamo i due DFA della Figura 4.10. Entrambi accettano la stringa vuota e tutte le stringhe che terminano per 0, cioè il linguaggio dell'espressione regolare

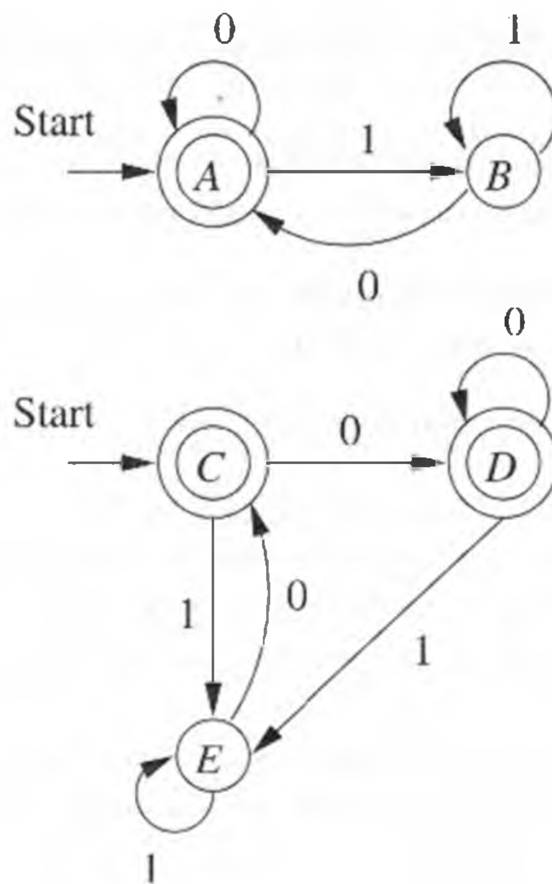


Figura 4.10 Due DFA equivalenti.

$\epsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0}$. Possiamo immaginare che la figura rappresenti un solo DFA con cinque stati, da A a E. Applicando l'algoritmo riempi-tabella a questo automa si ottiene la tabella della Figura 4.11.

Per vedere come si riempie la tabella, cominciamo ponendo degli x nelle coppie di stati di cui uno solo è accettante. Scopriamo che non c'è altro da fare. Le quattro coppie residue $\{A, C\}$, $\{A, D\}$, $\{C, D\}$ e $\{B, E\}$ sono equivalenti. Il lettore può verificare che nella parte induttiva dell'algoritmo non si scoprono altre coppie distinguibili. Per esempio, con la tabella della Figura 4.11 non possiamo distinguere la coppia $\{A, D\}$ perché A e D su 0 tornano in se stessi e su 1 vanno nella coppia $\{B, E\}$, che non è ancora stata distinta. Dal momento che l'algoritmo stabilisce l'equivalenza di A e C, che sono gli stati iniziali dei due automi originari, concludiamo che questi accettano lo stesso linguaggio. □

Il tempo per riempire la tabella, e quindi per decidere se due stati sono equivalenti, è polinomiale nel numero di stati. Se ci sono n stati, le coppie di stati sono $\binom{n}{2}$, cioè $n(n-1)/2$. In una iterazione si considerano tutte le coppie di stati per verificare se una delle coppie di successori è stata dichiarata distinguibile; una iterazione richiede perciò un tempo non superiore a $O(n^2)$. Se poi in una iterazione non si aggiungono nuovi x,

B	x			
C		x		
D		x		
E	x		x	x
	A	B	C	D

Figura 4.11 Tabella di distinguibilità per la Figura 4.10.

l'algoritmo termina. Quindi non ci possono essere più di $O(n^2)$ iterazioni, e $O(n^4)$ è senz'altro un limite superiore per il tempo di esecuzione dell'algoritmo.

Esiste tuttavia un algoritmo migliore, che riempie la tabella in tempo $O(n^2)$. L'idea è di predisporre, per ogni coppia di stati $\{r, s\}$, un elenco delle coppie $\{p, q\}$ che dipendono da $\{r, s\}$, tali cioè che, se $\{r, s\}$ è distinguibile, lo è anche $\{p, q\}$. Inizialmente gli elenchi si creano esaminando ogni coppia di stati $\{p, q\}$ e ponendola, per ogni simbolo a, nell'elenco associato alla coppia $\{\delta(p, a), \delta(q, a)\}$, cioè i successori di p e q rispetto ad a.

Se si stabilisce che $\{r, s\}$ è distinguibile, si percorre l'elenco a essa associato e si dichiara distinguibile ogni coppia dell'elenco che non lo era ancora; tali coppie vengono aggiunte a una coda di coppie per le quali si deve esaminare l'elenco nello stesso modo.

Il lavoro complessivo svolto dall'algoritmo è proporzionale alla somma delle lunghezze degli elenchi perché a ogni passo o si aggiunge qualcosa a un elenco (fase di preparazione) o si esamina, per la prima e ultima volta, un elemento dell'elenco (scansione dell'elenco di una coppia dichiarata distinguibile). Poiché si assume costante la dimensione dell'alfabeto di input, ogni coppia sta in $O(1)$ elenchi. Dato che ci sono $O(n^2)$ coppie, il lavoro totale è $O(n^2)$.

4.4.3 Minimizzazione di DFA

La soluzione al problema dell'equivalenza di stati ha un'altra conseguenza importante: è possibile "minimizzare" un DFA, cioè trovare un DFA equivalente con il numero minimo di stati fra i DFA che accettano lo stesso linguaggio. Inoltre il DFA minimo rispetto agli stati è unico, a meno di rinominare gli stati. Vediamo l'algoritmo.

1. Per prima cosa si eliminano gli stati irraggiungibili dallo stato iniziale.

2. Gli stati rimanenti vengono poi ripartiti in blocchi in modo che gli stati in uno stesso blocco siano tutti equivalenti e due stati in blocchi diversi non lo siano mai. Il Teorema 4.24 mostra che possiamo sempre eseguire questa partizione.

Esempio 4.22 Consideriamo la tabella della Figura 4.9, che stabilisce equivalenze e distinguibilità per gli stati della Figura 4.8. La partizione degli stati in blocchi di equivalenza è data da $(\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\})$. Si osservi che le tre coppie di stati equivalenti si trovano ciascuna in un blocco, mentre gli stati distinguibili da ogni altro stato se ne stanno da soli, ciascuno in un blocco.

Per l'automa della Figura 4.10 la partizione è $(\{A, C, D\}, \{B, E\})$. Questo esempio mostra che un blocco può contenere più di due stati. Può sembrare casuale che A, C e D siano nello stesso blocco perché sono equivalenti a coppie e nessuno di loro è equivalente a un altro stato, ma, come vedremo nel prossimo teorema, ciò è garantito dalla definizione di equivalenza fra stati. □

Teorema 4.23 L'equivalenza di stati è transitiva. In altre parole, se due stati p e q di un DFA $A = (Q, \Sigma, \delta, q_0, F)$ sono equivalenti, e scopriamo che anche q ed r lo sono, saranno equivalenti anche p ed r.

DIMOSTRAZIONE Notiamo che la transitività è una proprietà insita in ogni relazione di "equivalenza", ma non basta dire che qualcosa è un'equivalenza per renderla transitiva, bisogna dimostrare che quella denominazione è giustificata.

Supponiamo che le coppie $\{p, q\}$ e $\{q, r\}$ siano equivalenti, ma che $\{p, r\}$ sia distinguibile. Allora esiste una stringa di input w tale che uno solo fra $\hat{\delta}(p, w)$ e $\hat{\delta}(r, w)$ è uno stato accettante. Per simmetria supponiamo che si tratti di $\hat{\delta}(p, w)$.

Ci chiediamo se $\hat{\delta}(q, w)$ sia o no accettante. Se lo fosse, $\{q, r\}$ sarebbe distinguibile perché $\hat{\delta}(r, w)$ non è accettante. Se $\hat{\delta}(q, w)$ non fosse accettante, $\{p, q\}$ sarebbe distinguibile per una ragione analoga. Abbiamo dimostrato per assurdo che $\{p, r\}$ non è distinguibile, quindi è una coppia equivalente. □

Possiamo usare il Teorema 4.23 a sostegno dell'ovvio algoritmo di partizionamento degli stati: per ogni stato q costruiamo un blocco formato da q e da tutti gli stati a esso equivalenti. Dobbiamo dimostrare che i blocchi così formati sono una partizione, cioè che nessuno stato si trova in due blocchi distinti.

Osserviamo per prima cosa che tutti gli stati in un blocco sono fra loro equivalenti. In altre parole, se p ed r si trovano nel blocco degli stati equivalenti a q, per il Teorema 4.23 p ed r sono fra loro equivalenti.

Supponiamo che ci siano due blocchi con un'intersezione non nulla, ma non identici; sia allora B un blocco contenente gli stati p e q, e sia C un altro blocco contenente p ma non q. Poiché p e q si trovano in uno stesso blocco, sono equivalenti. Vediamo come è stato formato il blocco C. Se fosse il blocco generato da p, anche q, essendo equivalente a p, sarebbe in C. Deve quindi esistere un terzo stato s che ha generato C, ovvero C è formato dagli stati equivalenti a s.

Sappiamo che p, essendo nel blocco C, è equivalente a s; sappiamo anche che p è equivalente a q perché entrambi appartengono al blocco B. Per transitività (Teorema 4.23) q è

equivalente a s. Ma allora q appartiene a C, e quindi si ha una contraddizione. Concludiamo che l'equivalenza fra stati è una partizione degli stessi: o due stati hanno il medesimo insieme di stati equivalenti (compresi essi stessi) o i loro insiemi di stati equivalenti sono disgiunti. Possiamo tirare le somme dell'analisi condotta fin qui.

Teorema 4.24 Se per ogni stato q di un DFA si crea un *blocco* formato da q e da tutti gli stati equivalenti a q, allora l'insieme dei blocchi distinti costituisce una *partizione* dell'insieme degli stati.[5] Quindi ogni stato appartiene a un solo blocco. Tutti gli elementi di un blocco sono equivalenti, mentre due stati scelti da blocchi diversi non lo sono. □

Possiamo ora riassumere l'algoritmo che minimizza un DFA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Con l'algoritmo riempi-tabella determiniamo le coppie di stati equivalenti.

2. Con il metodo sopra descritto partizioniamo l'insieme Q degli stati in blocchi di stati a due a due equivalenti.

3. Costruiamo il DFA minimo equivalente B, prendendo i blocchi come stati. Sia γ la funzione di transizione di B. Sia S un insieme di stati equivalenti di A, e a un simbolo di input. Deve esistere allora un blocco T tale che, per ogni stato q in S, $\delta(q, a)$ è un elemento di T. In caso contrario il simbolo a porterebbe due stati di S, p e q, in blocchi diversi e, per il Teorema 4.24, quegli stati sarebbero distinguibili. Concludiamo perciò che p e q non sono equivalenti e non appartengono entrambi a S. Di conseguenza possiamo porre $\gamma(S, a) = T$. Inoltre vale quanto segue.

   (a) Lo stato iniziale di B è il blocco contenente lo stato iniziale di A.

   (b) L'insieme degli stati accettanti di B è l'insieme dei blocchi che contengono stati accettanti di A. Notiamo che, se uno stato è accettante, tutti gli stati nel suo blocco devono essere accettanti. Infatti uno stato accettante è distinguibile da ogni stato non accettante; quindi un blocco di stati equivalenti non può contenere stati accettanti e stati non accettanti.

Esempio 4.25 Minimizziamo il DFA della Figura 4.8. Abbiamo calcolato i blocchi della partizione degli stati nell'Esempio 4.22. La Figura 4.12 mostra l'automa minimo. I suoi cinque stati corrispondono ai cinque blocchi di stati equivalenti per l'automa della Figura 4.8.

Poiché A è lo stato iniziale della Figura 4.8, lo stato iniziale è $\{A, E\}$. L'unico stato accettante è $\{C\}$ perché C è l'unico stato accettante nella Figura 4.8. Si osservi che le

[5] Precisiamo che lo stesso blocco può essere costruito più volte, a partire da stati diversi. La partizione è fatta da blocchi *distinti*, cosicché ogni blocco vi compare una volta sola.

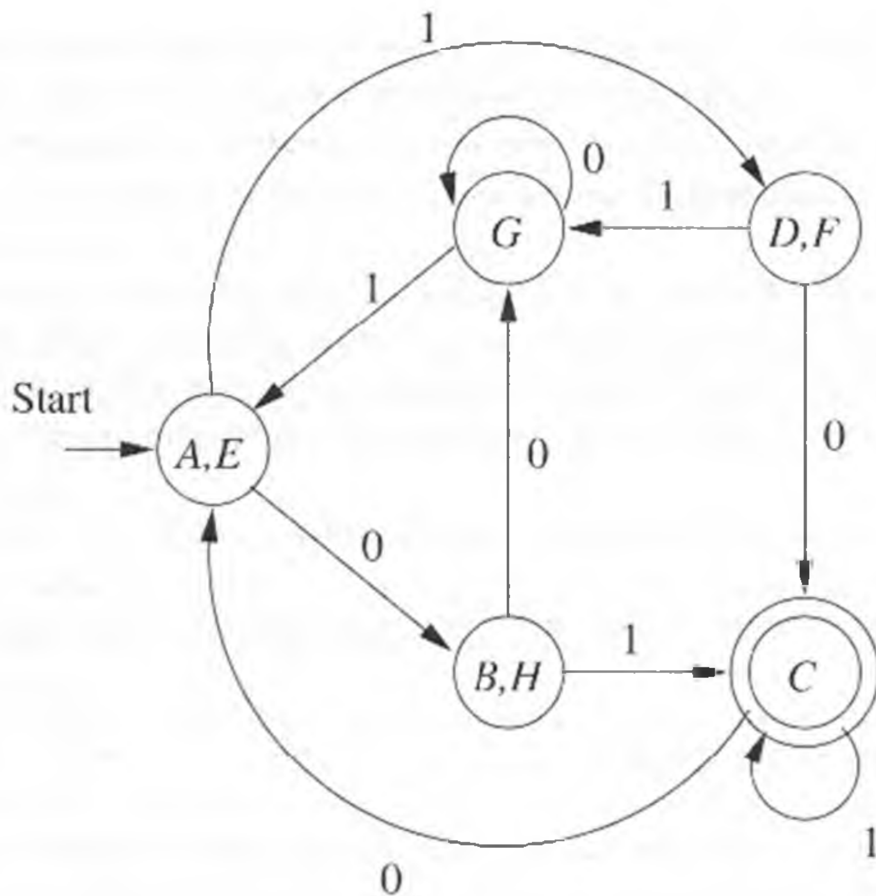


Figura 4.12 DFA minimo equivalente alla Figura 4.8.

transizioni della Figura 4.12 riflettono correttamente quelle della Figura 4.8. Per esempio nella Figura 4.12 c'è una transizione da $\{A, E\}$ a $\{B, H\}$ su input 0 perché nella Figura 4.8, su input 0, A va in B ed E va in H. Analogamente, su input 1 $\{A, E\}$ va in $\{D, F\}$. Esaminando la Figura 4.8 vediamo che sia A sia E vanno in F su input 1; quindi anche la scelta del successore di $\{A, E\}$ su input 1 è corretta. Notiamo che né A né E vanno in D su input 1, ma ciò è irrilevante. Il lettore può verificare la correttezza delle altre transizioni. □

4.4.4 Perché il DFA minimo non può essere migliorato

Supponiamo di minimizzare un DFA A con il metodo di partizionamento del Teorema 4.24, e di ottenere un DFA M. Il teorema dimostra che non possiamo raggruppare gli stati di A in un numero più piccolo di blocchi mantenendo l'equivalenza dei DFA. Può esistere un altro DFA N, senza alcuna relazione con A, che accetta lo stesso linguaggio di A e di M, ma con meno stati di M? Dimostreremo per assurdo che N non esiste.

Cominciamo eseguendo la procedura per la distinguibilità descritta nel Paragrafo 4.4.1 sugli stati di M ed N congiuntamente, come se si trattasse di un solo DFA. Possiamo

Minimizzare gli stati di un DFA

Si potrebbe pensare che la tecnica di partizione degli stati che minimizza gli stati di un DFA si applichi anche alla ricerca di un NFA minimo equivalente a un DFA o a un NFA assegnato. Invece, anche se è possibile trovare, per enumerazione completa, un NFA con numero minimo di stati che riconosce un linguaggio regolare dato, non è sufficiente ripartire gli stati di un NFA per quel linguaggio.

Ne abbiamo un esempio nella Figura 4.13. Non ci sono coppie di stati equivalenti; lo stato B è senz'altro distinguibile dagli stati non accettanti A e C; A e C sono distinguibili dall'input 0; A è l'unico successore di C e non è accettante, mentre l'insieme dei successori di A è $\{A, B\}$, che contiene uno stato accettante. Perciò non si può ridurre il numero di stati raggruppando quelli equivalenti.

Eppure si può trovare un NFA più piccolo per lo stesso linguaggio eliminando lo stato C. A e B da soli accettano infatti tutte le stringhe che terminano per 0, e l'aggiunta di C non permette di accettare altre stringhe.

supporre che i nomi degli stati di M siano distinti da quelli di N; in questo modo la funzione di transizione dell'automa composto è l'unione delle due funzioni, senza interferenze. Uno stato è accettante nel DFA composto se e solo se è accettante nel suo DFA di provenienza.

Gli stati iniziali di M ed N sono indistinguibili perché $L(M) = L(N)$. Inoltre, se p e q sono indistinguibili, lo sono anche i loro successori su ogni simbolo di input; infatti, se potessimo distinguere i successori, sapremmo distinguere anche p da q.

Né M né N hanno stati inaccessibili. In caso contrario potremmo eliminarli e ottenere un DFA più piccolo per lo stesso linguaggio. Perciò ogni stato di M è indistinguibile da almeno uno stato di N. Per convincersene, sia p uno stato di M. Deve esistere una stringa $a_1a_2 \cdots a_k$ che porta dallo stato iniziale di M a p. La stessa stringa porta anche dallo stato iniziale di N a uno stato q. Sappiamo che gli stati iniziali sono indistinguibili, quindi anche i loro successori rispetto ad a_1 sono indistinguibili. Lo stesso vale per i successori di questi stati rispetto ad a_2, e così via, fino a concludere che p e q sono indistinguibili.

Poiché N ha meno stati di M, ci sono due stati di M indistinguibili dallo stesso stato di N, e quindi indistinguibili fra loro. Ma M è stato costruito in modo tale che tutti i suoi stati siano distinguibili a due a due. Abbiamo una contraddizione, da cui segue che N non può esistere ed M ha un numero di stati non superiore a quello di un qualsiasi DFA equivalente ad A. In termini formali abbiamo dimostrato quanto segue.

Teorema 4.26 Sia A un DFA ed M il DFA costruito a partire da A dall'algoritmo de-

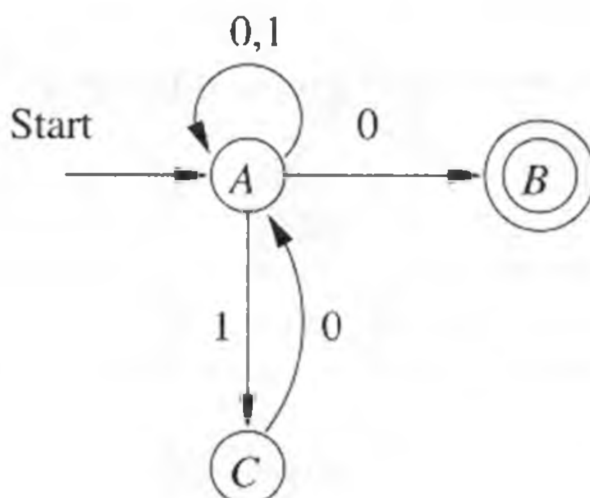


Figura 4.13 Un NFA che non si può minimizzare per equivalenza sugli stati.

scritto nell'enunciato del Teorema 4.24. Allora il numero di stati di M non è superiore a quello di un qualsiasi DFA equivalente ad A. □

Possiamo in realtà affermare qualcosa di più del Teorema 4.26: esiste una corrispondenza uno-a-uno fra gli stati di qualsiasi altro N minimale e quelli di M. Abbiamo infatti dimostrato che ogni stato di M dev'essere equivalente a uno stato di N, e nessuno stato di M può essere equivalente a due stati di N. In modo simile possiamo provare che nessuno stato di N può essere equivalente a due stati di M, e che ogni stato di N dev'essere equivalente a uno stato di M. In conclusione, il DFA minimo equivalente ad A è unico, a meno di rinominare gli stati.

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

Figura 4.14 Un DFA da minimizzare.

4.4.5 Esercizi

* **Esercizio 4.4.1** La Figura 4.14 riporta la tabella di transizione di un DFA.

a) Tracciate la tabella di distinguibilità dell'automa.

b) Costruite il DFA minimo equivalente.

Esercizio 4.4.2 Svolgete l'Esercizio 4.4.1 per il DFA della Figura 4.15.

	0	1
$\rightarrow A$	B	E
B	C	F
$*C$	D	H
D	E	H
E	F	I
$*F$	G	B
G	H	B
H	I	C
$*I$	A	E

Figura 4.15 Un altro DFA da minimizzare.

!! **Esercizio 4.4.3** Siano p e q due stati distinguibili di un DFA A con n stati. Qual è il più stretto limite superiore, in funzione di n, della lunghezza della stringa più corta che distingue p da q?

4.5 Riepilogo

✦ *Il pumping lemma*: se un linguaggio è regolare, allora ogni stringa sufficientemente lunga nel linguaggio ha una sottostringa non vuota che può essere "replicata", ossia ripetuta per un qualunque numero di volte senza che le stringhe risultanti escano dal linguaggio. Si può dunque usare il *pumping lemma* per dimostrare che diversi linguaggi *non* sono regolari.

✦ *Operazioni che preservano la regolarità*: molte operazioni, applicate ai linguaggi regolari, producono come risultato un linguaggio regolare. Tra queste ci sono l'unione, la concatenazione, la chiusura, l'intersezione, la complementazione, la differenza, l'inversione, l'omomorfismo (sostituzione di ogni simbolo con una stringa associata) e l'omomorfismo inverso.

- *Verificare se un linguaggio regolare è vuoto*: esiste un algoritmo che, data la rappresentazione di un linguaggio regolare, come un automa oppure un'espressione regolare, decide se il linguaggio rappresentato è l'insieme vuoto oppure no.

- *Verificare l'appartenenza a un linguaggio regolare*: esiste un algoritmo che, data una stringa e la rappresentazione di un linguaggio regolare, decide se la stringa appartiene o no al linguaggio.

- *Distinguibilità di stati*: due stati di un DFA sono distinguibili se esiste una stringa di input che porta solo uno dei due in uno stato accettante. Partendo soltanto dal fatto che le coppie formate da uno stato accettante e uno non accettante sono distinguibili, e aggiungendo come nuove coppie quelle i cui successori su un certo simbolo di input sono distinguibili, possiamo scoprire tutte le coppie di stati distinguibili.

- *Minimizzazione di automi a stati finiti deterministici*: si possono ripartire gli stati di un DFA in blocchi di stati a due a due indistinguibili. I membri di due blocchi diversi sono sempre distinguibili. Se sostituiamo ogni blocco con un singolo stato, otteniamo un DFA equivalente, con un numero di stati non superiore a quello di ogni altro DFA per lo stesso linguaggio.

4.6 Bibliografia

A parte le ovvie proprietà di chiusura delle espressioni regolari (unione, concatenazione e star) dimostrate da Kleene [6], quasi tutti i risultati relativi alle proprietà di chiusura dei linguaggi regolari sono adattamenti di risultati simili per i linguaggi liberi dal contesto (classe che tratteremo nei prossimi capitoli). In particolare il *pumping lemma* per i linguaggi regolari è la semplificazione di un risultato corrispondente per i linguaggi liberi dal contesto, dimostrato da Bar-Hillel, Perles e Shamir [1]. Lo stesso articolo fornisce indirettamente diverse altre proprietà di chiusura presentate qui. La chiusura per omomorfismo inverso si deve invece a [2].

L'operazione di quoziente presentata nell'Esercizio 4.2.2 risale a [3]. Di fatto in quel lavoro si tratta di un'operazione più generale, in cui un simbolo a può essere sostituito da un linguaggio regolare. La serie di operazioni del tipo "eliminazione parziale", a partire dall'Esercizio 4.2.8 sulle prime metà delle stringhe di un linguaggio regolare, proviene da [8]. Seiferas e McNaughton [9] hanno stabilito nel caso generale quando un'operazione di eliminazione preserva i linguaggi regolari.

Gli algoritmi originali di decisione per i problemi del linguaggio vuoto, della finitezza e dell'appartenenza per i linguaggi regolari sono tratti da [7]. Quelli per minimizzare gli stati di un DFA compaiono nello stesso lavoro e in [5]. L'algoritmo più efficiente per trovare il DFA minimo si trova in [4].

1. Y. Bar-Hillel, M. Perles, E. Shamir, "On formal properties of simple phrase-structure grammars," *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), pp. 143–172.

2. S. Ginsburg, G. Rose, "Operations which preserve definability in languages," *J. ACM* **10**:2 (1963), pp. 175–195.

3. S. Ginsburg, E. H. Spanier, "Quotients of context-free languages," *J. ACM* **10**:4 (1963), pp. 487–492.

4. J. E. Hopcroft, "An $n \log n$ algorithm for minimizing the states in a finite automaton," in Z. Kohavi (ed.) *The Theory of Machines and Computations*, Academic Press, New York, pp. 189–196.

5. D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.* **257**:3-4 (1954), pp. 161–190 e 275–303.

6. S. C. Kleene, "Representation of events in nerve nets and finite automata," in C. E. Shannon, J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 3–42.

7. E. F. Moore, "Gedanken experiments on sequential machines," in C. E. Shannon, J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 129–153.

8. R. E. Stearns, J. Hartmanis, "Regularity-preserving modifications of regular expressions," *Information and Control* **6**:1 (1963), pp. 55–69.

9. J. I. Seiferas, R. McNaughton, "Regularity-preserving modifications," *Theoretical Computer Science* **2**:2 (1976), pp. 147–154.

Capitolo 5

Grammatiche e linguaggi liberi dal contesto

Volgiamo ora l'attenzione a una classe più ampia di linguaggi rispetto a quelli regolari: i linguaggi "liberi dal contesto". Si tratta di linguaggi che hanno una notazione naturale ricorsiva, chiamata "grammatiche libere dal contesto". Le grammatiche libere dal contesto sono state cruciali nei compilatori sin dagli anni '60: grazie a loro la realizzazione di un *parser* (una funzione che estrae la struttura di un programma) è passata da un'attività di implementazione *ad hoc* che richiedeva molto tempo a un lavoro di routine che può essere svolto in un solo pomeriggio. Di recente le grammatiche libere dal contesto sono state usate per descrivere formati di documenti attraverso le cosiddette DTD (*Document Type Definition*, definizione di tipo di documento), utilizzate dagli utenti di XML (*eXtensible Markup Language*) per lo scambio di informazioni nel Web.

In questo capitolo introduciamo la notazione delle grammatiche libere dal contesto e mostriamo come definiscono i linguaggi. Presentiamo inoltre l'"albero sintattico" (*parse tree*), una rappresentazione grafica della struttura che una grammatica impone alle stringhe del suo linguaggio. L'albero sintattico è prodotto dal *parser* di un linguaggio di programmazione e costituisce il modo tipico di rappresentare la struttura dei programmi.

I linguaggi liberi dal contesto sono descritti compiutamente anche da una notazione in forma di automa, l'automa "a pila" (*pushdown automaton*). L'automa a pila sarà materia del Capitolo 6. Sebbene meno importanti degli automi a stati finiti, gli automi a pila, soprattutto perché sono equivalenti alle grammatiche libere dal contesto come modo di definire linguaggi, si rivelano molto utili nell'esplorazione delle proprietà di chiusura e di decisione dei linguaggi liberi dal contesto (vedi Capitolo 7).

5.1 Grammatiche libere dal contesto

Cominciamo col presentare informalmente la notazione delle grammatiche libere dal contesto. Passeremo alle definizioni formali dopo averne esaminato alcune importanti caratteristiche. Mostreremo come si definisce una grammatica in termini formali e presenteremo il processo di "derivazione", che determina quali stringhe appartengono al linguaggio di una grammatica.

5.1.1 Un esempio informale

Consideriamo il linguaggio delle palindrome. Una stringa è *palindroma* se si può leggere indifferentemente da sinistra a destra e da destra a sinistra, come per esempio `otto` o `madamimadam` ("Madam, I'm Adam", la prima frase che si suppone Eva abbia udito nel Giardino dell'Eden). In altri termini una stringa w è palindroma se e solo se $w = w^R$. Per semplificare le cose descriveremo solo le stringhe palindrome sull'alfabeto $\{0, 1\}$; questo linguaggio comprende stringhe come 0110, 11011 e ϵ, ma non 011 o 0101.

È facile verificare che il linguaggio L_{pal} delle palindrome di 0 e 1 non è un linguaggio regolare. Per farlo ci serviamo del *pumping lemma*. Se L_{pal} è regolare, sia n la costante associata, e consideriamo la stringa palindroma $w = 0^n10^n$. Per il lemma possiamo scomporre w in $w = xyz$, in modo tale che y consista di uno o più 0 dal primo gruppo. Di conseguenza xz, che dovrebbe trovarsi in L_{pal} se L_{pal} fosse regolare, avrebbe meno 0 a sinistra dell'unico 1 rispetto a quelli a destra. Dunque xz non può essere palindroma. Abbiamo così confutato l'ipotesi che L_{pal} sia un linguaggio regolare.

Per stabilire in quali casi una stringa di 0 e 1 si trova in L_{pal}, possiamo avvalerci di una semplice definizione ricorsiva. La base della definizione stabilisce che alcune stringhe semplici si trovano in L_{pal}; si sfrutta poi il fatto che se una stringa è palindroma deve cominciare e finire con lo stesso simbolo. Inoltre, quando il primo e l'ultimo simbolo vengono rimossi, la stringa risultante dev'essere palindroma.

BASE ϵ, 0 e 1 sono palindrome.

INDUZIONE Se w è palindroma, lo sono anche $0w0$ e $1w1$. Nessuna stringa di 0 e 1 è palindroma, salvo che non risulti dalla base e dalla regola di induzione appena esposte.

Una grammatica libera dal contesto è una notazione formale per esprimere simili definizioni ricorsive di linguaggi. Una grammatica consiste di una o più variabili che rappresentano classi di stringhe, ossia linguaggi. Nell'esempio ci occorre una sola variabile, P, che rappresenta l'insieme delle palindrome, cioè la classe delle stringhe che formano il linguaggio L_{pal}. Opportune regole stabiliscono come costruire le stringhe in ogni classe. La costruzione può impiegare simboli dell'alfabeto, stringhe di cui si conosce già l'appartenenza a una delle classi, oppure entrambi gli elementi.

Esempio 5.1 Le regole che definiscono le palindrome, espresse nella notazione delle grammatiche libere dal contesto, sono riportate nella Figura 5.1. Il loro significato verrà chiarito nel Paragrafo 5.1.2.

$$
\begin{array}{lllll}
1. & P & \rightarrow & \epsilon \\
2. & P & \rightarrow & 0 \\
3. & P & \rightarrow & 1 \\
4. & P & \rightarrow & 0P0 \\
5. & P & \rightarrow & 1P1
\end{array}
$$

Figura 5.1 Una grammatica libera dal contesto per le stringhe palindrome.

La prime tre regole costituiscono la base e indicano che la classe delle palindrome include le stringhe ϵ, 0 e 1. Nessuno dei membri destri delle regole (le porzioni che seguono le frecce) contiene una variabile; questo è il motivo per cui formano la base della definizione.

Le ultime due regole costituiscono la parte induttiva. Per esempio la regola 4 dice che, se si prende una qualunque stringa w dalla classe P, anche $0w0$ si trova nella classe P. Analogamente la regola 5 indica che anche $1w1$ è in P. □

5.1.2 Definizione delle grammatiche libere dal contesto

La descrizione grammaticale di un linguaggio consiste di quattro componenti importanti.

1. Un insieme finito di simboli che formano le stringhe del linguaggio da definire. Nell'esempio delle palindrome l'insieme è $\{0, 1\}$. Chiameremo quest'alfabeto i *terminali* o *simboli terminali*.

2. Un insieme finito di *variabili*, talvolta dette anche *non terminali* oppure *categorie sintattiche*. Ogni variabile rappresenta un linguaggio, ossia un insieme di stringhe. Nell'esempio precedente c'è una sola variabile, P, usata per rappresentare la classe delle stringhe palindrome sull'alfabeto $\{0, 1\}$.

3. Una variabile, detta *simbolo iniziale*, che rappresenta il linguaggio da definire. Le altre variabili rappresentano classi ausiliarie di stringhe, che contribuiscono a definire il linguaggio del simbolo iniziale. Nell'esempio, P, l'unica variabile, è il simbolo iniziale.

4. Un insieme finito di *produzioni*, o *regole*, che rappresentano la definizione ricorsiva di un linguaggio. Ogni produzione consiste di tre parti.

   (a) Una variabile che viene definita (parzialmente) dalla produzione ed è spesso detta la *testa* della produzione.

(b) Il simbolo di produzione $\rightarrow$.

(c) Una stringa di zero o più terminali e variabili, detta il *corpo* della produzione, che rappresenta un modo di formare stringhe nel linguaggio della variabile di testa. Le stringhe si formano lasciando immutati i terminali e sostituendo ogni variabile del corpo con una stringa appartenente al linguaggio della variabile stessa.

Esempi di produzioni sono illustrati nella Figura 5.1.

I quattro componenti appena descritti formano una *grammatica libera dal contesto*, o semplicemente *grammatica*, o *CFG*, (*Context-Free Grammar*). Rappresenteremo una CFG per mezzo dei suoi quattro componenti, ossia $G = (V, T, P, S)$, dove V è l'insieme delle variabili, T i terminali, P l'insieme delle produzioni ed S il simbolo iniziale.

Esempio 5.2 La grammatica G_{pal} per le palindrome è rappresentata da

$$G_{pal} = (\{P\}, \{0, 1\}, A, P)$$

dove A rappresenta l'insieme delle cinque produzioni nella Figura 5.1. □

Esempio 5.3 Esaminiamo una CFG più complessa, che rappresenta, semplificandole, le espressioni in un tipico linguaggio di programmazione. Dapprima ci limitiamo agli operatori $+$ e $*$, corrispondenti a somma e moltiplicazione. Ammettiamo che gli operandi siano identificatori, ma al posto dell'insieme completo di identificatori tipici (una lettera seguita da zero o più lettere e cifre) accettiamo solo le lettere a e b e le cifre 0 e 1. Ogni identificatore deve iniziare per a o b e può continuare con una qualunque stringa in $\{a, b, 0, 1\}^*$.

In questa grammatica sono necessarie due variabili. La prima, che chiameremo E, rappresenta le espressioni. È il simbolo iniziale e rappresenta il linguaggio delle espressioni che stiamo definendo. L'altra variabile, I, rappresenta gli identificatori. Il suo linguaggio è regolare; è il linguaggio dell'espressione regolare

$$(\mathbf{a} + \mathbf{b})(\mathbf{a} + \mathbf{b} + \mathbf{0} + \mathbf{1})^*$$

Eviteremo di usare direttamente le espressioni regolari nelle grammatiche, preferendo piuttosto un insieme di produzioni che abbiano lo stesso significato dell'espressione in esame.

La grammatica per le espressioni è definita formalmente da $G = (\{E, I\}, T, P, E)$, dove T è l'insieme di simboli $\{+, *, (,), a, b, 0, 1\}$ e P è l'insieme di produzioni nella Figura 5.2. L'interpretazione delle produzioni è la seguente.

La regola (1) è la regola di base per le espressioni, e afferma che un'espressione può essere un singolo identificatore. Le regole dalla (2) alla (4) descrivono il caso induttivo

$$
\begin{array}{lrcl}
1. & E & \rightarrow & I \\
2. & E & \rightarrow & E + E \\
3. & E & \rightarrow & E * E \\
4. & E & \rightarrow & (E) \\
\\
5. & I & \rightarrow & a \\
6. & I & \rightarrow & b \\
7. & I & \rightarrow & Ia \\
8. & I & \rightarrow & Ib \\
9. & I & \rightarrow & I0 \\
10. & I & \rightarrow & I1
\end{array}
$$

Figura 5.2 Una grammatica libera dal contesto per espressioni semplici.

per le espressioni. La regola (2) afferma che un'espressione può essere formata da due espressioni connesse dal segno $+$; la regola (3) dice la stessa cosa per il segno di moltiplicazione. La regola (4) dichiara che, se si prende una qualunque espressione e la si racchiude fra parentesi, il risultato è ancora un'espressione.

Le regole dalla (5) alla (10) descrivono gli identificatori. La base è data dalle regole (5) e (6), secondo le quali a e b sono identificatori. Le rimanenti quattro regole sono il caso induttivo, e affermano che se abbiamo un identificatore possiamo farlo seguire da a, b, 0 oppure 1, e il risultato è comunque un altro identificatore. □

5.1.3 Derivazioni per mezzo di una grammatica

Le produzioni di una CFG si applicano per dedurre che determinate stringhe appartengono al linguaggio di una certa variabile. La deduzione può seguire due strade. Quella più convenzionale si serve delle regole utilizzando il corpo per passare alla testa. In altre parole prendiamo stringhe di cui conosciamo l'appartenenza al linguaggio di ognuna delle variabili del corpo, le concateniamo nell'ordine adeguato, con i terminali che compaiono nel corpo, e deduciamo che la stringa risultante è nel linguaggio della variabile che compare in testa. Chiameremo questa procedura *inferenza ricorsiva*.

Il secondo modo per definire il linguaggio di una grammatica applica le produzioni dalla testa al corpo. Espandiamo il simbolo iniziale usando una delle sue produzioni (cioè usando una produzione la cui testa sia il simbolo iniziale). Espandiamo ulteriormente la stringa risultante sostituendo una delle variabili con il corpo di una delle sue produzioni, e così via, fino a derivarne una stringa fatta interamente di terminali. Il linguaggio della grammatica è l'insieme di tutte le stringhe di terminali ottenute con questa procedura. Questa tecnica è detta *derivazione*.

Notazione compatta per le produzioni

È opportuno pensare a una produzione come "appartenente" alla variabile della sua testa. Useremo spesso termini come "le produzioni per A" oppure "le A-produzioni" per riferirci alle produzioni la cui testa è la variabile A. Possiamo scrivere le produzioni di una grammatica elencando ogni variabile una volta, e facendola seguire dai corpi delle sue produzioni, separati da barre verticali. In altre parole le produzioni $A \to \alpha_1, A \to \alpha_2, \dots, A \to \alpha_n$ possono essere sostituite dalla notazione $A \to \alpha_1|\alpha_2|\cdots|\alpha_n$. Per esempio la grammatica per le palindrome della Figura 5.1 può essere scritta come $P \to \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$.

Cominceremo da un esempio di inferenza ricorsiva. Poiché spesso è più naturale considerare una grammatica secondo lo schema della derivazione, come passo successivo svilupperemo la notazione per descrivere le derivazioni.

Esempio 5.4 Consideriamo alcune inferenze possibili nella grammatica per le espressioni della Figura 5.2. La Figura 5.3 riassume queste inferenze. Per esempio la riga (i) dice che possiamo dedurre che la stringa a è nel linguaggio per I usando la produzione 5. Le righe dalla (ii) alla (iv) affermano che possiamo dedurre che $b00$ è un identificatore usando una volta la produzione 6 (per ottenere la b) e poi applicando due volte la produzione 9 (per accodare i due 0).

	Stringa ricavata	Per il linguaggio di	Produzione impiegata	Stringhe impiegate
(i)	a	I	5	—
(ii)	b	I	6	—
(iii)	$b0$	I	9	(ii)
(iv)	$b00$	I	9	(iii)
(v)	a	E	1	(i)
(vi)	$b00$	E	1	(iv)
(vii)	$a + b00$	E	2	$(v), (vi)$
$(viii)$	$(a + b00)$	E	4	(vii)
(ix)	$a * (a + b00)$	E	3	$(v), (viii)$

Figura 5.3 Inferenza di stringhe dalla grammatica della Figura 5.2.

Ogni identificatore è un'espressione; le righe (v) e (vi) sfruttano quindi la produzione 1 per dedurre che anche le stringhe a e $b00$, che risultano essere identificatori per inferenza

nelle righe (i) e (iv), sono nel linguaggio della variabile E. La riga (vii) usa la produzione 2 per inferire che la somma di questi identificatori è un'espressione; la riga $(viii)$ usa la produzione 4 per inferire che la stessa stringa, posta tra parentesi, è un'espressione, e la riga (ix) usa la produzione 3 per moltiplicare l'identificatore a per l'espressione che abbiamo trovato nella riga $(viii)$. □

Il processo di derivazione di stringhe per applicazione di produzioni dalla testa al corpo richiede la definizione di un nuovo simbolo di relazione, $\Rightarrow$. Supponiamo che $G = (V, T, P, S)$ sia una CFG. Sia $\alpha A \beta$ una stringa di terminali e variabili, dove A è una variabile. In altre parole α e β sono stringhe in $(V \cup T)^*$, e A è in V. Sia $A \rightarrow \gamma$ una produzione di G. Allora scriviamo $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$. Se G risulta chiara dal contesto, scriviamo semplicemente $\alpha A \beta \Rightarrow \alpha\gamma\beta$. Si noti che un passo di derivazione sostituisce una variabile in un punto qualsiasi della stringa con il corpo di una delle sue produzioni.

Possiamo estendere la relazione $\Rightarrow$ fino a farle rappresentare zero, uno o più passi di derivazione, analogamente a come la funzione di transizione δ di un automa a stati finiti si estende a $\hat{\delta}$. Per le derivazioni usiamo il simbolo * per denotare "zero o più passi", come segue.

BASE Per qualsiasi stringa α di terminali e variabili, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. In altri termini: qualunque stringa deriva se stessa.

INDUZIONE Se $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ e $\beta \underset{G}{\Rightarrow} \gamma$, allora $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. Ossia, se α può diventare β in zero o più passi, e un passo ulteriore trasforma β in γ, allora α può diventare γ. Detto in altri termini, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ significa che esiste una sequenza di stringhe $\gamma_1, \gamma_2, \ldots, \gamma_n$, con $n \geq 1$, tale che

1. $\alpha = \gamma_1$
2. $\beta = \gamma_n$
3. per $i = 1, 2, \ldots, n-1$, abbiamo $\gamma_i \Rightarrow \gamma_{i+1}$.

Se la grammatica G è chiara dal contesto, allora scriviamo $\overset{*}{\Rightarrow}$ anziché $\overset{*}{\underset{G}{\Rightarrow}}$.

Esempio 5.5 L'inferenza che $a * (a + b00)$ appartiene al linguaggio della variabile E si riflette in una derivazione della stringa, a partire dalla stringa E. Eccone un esempio.

$$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow$$

$$a * (E) \Rightarrow a * (E + E) \Rightarrow a * (I + E) \Rightarrow a * (a + E) \Rightarrow$$

$$a * (a + I) \Rightarrow a * (a + I0) \Rightarrow a * (a + I00) \Rightarrow a * (a + b00)$$

Al primo passo E viene sostituita dal corpo della produzione 3 (dalla Figura 5.2). Al secondo passo si usa la produzione 1 per rimpiazzare la prima E con I, e così via. Si noti che abbiamo sistematicamente sostituito la variabile più a sinistra nella stringa. A ogni passo è comunque possibile scegliere quale variabile sostituire, e al posto di questa usare qualunque produzione. Per esempio al secondo passo avremmo potuto sostituire la seconda E con (E) servendoci della produzione 4. In tal caso avremmo avuto $E * E \Rightarrow E * (E)$. Potevamo anche scegliere una sostituzione incapace di condurre alla stessa stringa di terminali, per esempio la produzione 2 al primo passo: $E \Rightarrow E + E$. Infatti nessuna sostituzione delle due E può trasformare $E + E$ in $a * (a + b00)$.

Possiamo usare la relazione $\overset{*}{\Rightarrow}$ per abbreviare la derivazione. Dalla base sappiamo che $E \overset{*}{\Rightarrow} E$. L'uso reiterato della parte induttiva fornisce $E \overset{*}{\Rightarrow} E * E$, $E \overset{*}{\Rightarrow} I * E$, e così via, finché si ottiene $E \overset{*}{\Rightarrow} a * (a + b00)$.

I due punti di vista, inferenza ricorsiva e derivazione, sono equivalenti. In altre parole si deduce che una stringa di terminali w si trova nel linguaggio di una certa variabile A se e solo se $A \overset{*}{\Rightarrow} w$. La dimostrazione è però laboriosa, e la rimandiamo al Paragrafo 5.2. □

5.1.4 Derivazioni a sinistra e a destra

Per ridurre il numero di scelte possibili nella derivazione di una stringa, spesso è comodo imporre che a ogni passo si sostituisca la variabile all'estrema sinistra con il corpo di una delle sue produzioni. Tale derivazione viene detta *derivazione a sinistra* (*leftmost derivation*), e si indica tramite le relazioni $\underset{lm}{\Rightarrow}$ e $\underset{lm}{\overset{*}{\Rightarrow}}$, rispettivamente per uno o molti passi. Se la grammatica G in esame non è chiara dal contesto, possiamo porre il nome G sotto la freccia.

Analogamente è possibile imporre che a ogni passo venga sostituita la variabile più a destra con uno dei suoi corpi. In tal caso chiamiamo la derivazione *a destra* (*rightmost*), e usiamo i simboli $\underset{rm}{\Rightarrow}$ e $\underset{rm}{\overset{*}{\Rightarrow}}$ per indicare rispettivamente uno o più passi di derivazione a destra. Qualora non fosse evidente, il nome della grammatica può comparire anche in questo caso sotto i simboli.

Esempio 5.6 La derivazione dell'Esempio 5.5 è una derivazione a sinistra. Possiamo dunque descriverla come segue.

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E \underset{lm}{\Rightarrow}$$

$$a * (E) \underset{lm}{\Rightarrow} a * (E + E) \underset{lm}{\Rightarrow} a * (I + E) \underset{lm}{\Rightarrow} a * (a + E) \underset{lm}{\Rightarrow}$$

$$a * (a + I) \underset{lm}{\Rightarrow} a * (a + I0) \underset{lm}{\Rightarrow} a * (a + I00) \underset{lm}{\Rightarrow} a * (a + b00)$$

Notazione per le derivazioni delle CFG

Per aiutarci a ricordare il ruolo dei simboli usati nel trattare le CFG, si usano comunemente alcune convenzioni, che elenchiamo.

1. Le lettere minuscole in prossimità dell'inizio dell'alfabeto, a, b, e così via, sono simboli terminali. Assumiamo inoltre che cifre o altri caratteri, come + o le parentesi, sono terminali.

2. Le lettere maiuscole in prossimità dell'inizio dell'alfabeto, A, B, e così via, sono variabili.

3. Le lettere minuscole in prossimità della fine dell'alfabeto, come w o z, sono stringhe di terminali. Questa convenzione ci ricorda che i terminali sono analoghi ai simboli di input di un automa.

4. Le lettere maiuscole in prossimità della fine dell'alfabeto, come X o Y, sono terminali oppure variabili.

5. Le lettere minuscole greche, come α e β, sono stringhe che consistono di terminali e variabili.

Dato che non si tratta di un fattore importante, non esiste alcuna notazione specifica per le stringhe che consistono solo di variabili. Può accadere che una stringa denominata α, o con un'altra lettera greca, contenga solo variabili.

Possiamo inoltre riassumere la derivazione a sinistra scrivendo $E \underset{lm}{\overset{*}{\Rightarrow}} a*(a+b00)$, oppure esprimerne alcuni passi tramite espressioni come $E * E \underset{lm}{\overset{*}{\Rightarrow}} a * (E)$.

Esiste una derivazione a destra che applica le stesse sostituzioni per ogni variabile, sebbene in ordine diverso. Eccola.

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E) \underset{rm}{\Rightarrow}$$

$$E * (E + I) \underset{rm}{\Rightarrow} E * (E + I0) \underset{rm}{\Rightarrow} E * (E + I00) \underset{rm}{\Rightarrow} E * (E + b00) \underset{rm}{\Rightarrow}$$

$$E * (I + b00) \underset{rm}{\Rightarrow} E * (a + b00) \underset{rm}{\Rightarrow} I * (a + b00) \underset{rm}{\Rightarrow} a * (a + b00)$$

Questa derivazione permette di concludere $E \underset{rm}{\overset{*}{\Rightarrow}} a * (a + b00)$. □

Qualunque derivazione ha una derivazione a sinistra e una a destra equivalenti. In altre parole, se w è una stringa terminale e A una variabile, allora $A \overset{*}{\Rightarrow} w$ se e solo se $A \underset{lm}{\overset{*}{\Rightarrow}} w$, e $A \overset{*}{\Rightarrow} w$ se e solo se $A \underset{rm}{\overset{*}{\Rightarrow}} w$. Dimostreremo anche queste affermazioni nel Paragrafo 5.2.

5.1.5 Il linguaggio di una grammatica

Se $G = (V, T, P, S)$ è una CFG, il *linguaggio* di G, denotato con $L(G)$, è l'insieme delle stringhe terminali che hanno una derivazione dal simbolo iniziale. In altri termini

$$L(G) = \{w \text{ in } T^* \mid S \underset{G}{\overset{*}{\Rightarrow}} w\}$$

Se un linguaggio L è il linguaggio di una grammatica libera dal contesto, allora L è detto *linguaggio libero dal contesto*, o CFL (*Context-Free Language*). Per esempio abbiamo affermato che la grammatica della Figura 5.1 definisce il linguaggio delle palindrome sull'alfabeto $\{0, 1\}$. Di conseguenza l'insieme delle palindrome è un linguaggio libero dal contesto. Dimostriamo l'enunciato.

Teorema 5.7 $L(G_{pal})$, dove G_{pal} è la grammatica dell'Esempio 5.1, è l'insieme delle palindrome su $\{0, 1\}$.

DIMOSTRAZIONE Dimostreremo che una stringa w in $\{0, 1\}^*$ è in $L(G_{pal})$ se e solo se è palindroma, ossia $w = w^R$.

(Se) Supponiamo che w sia palindroma. Mostriamo per induzione su $|w|$ che w è in $L(G_{pal})$.

BASE Usiamo le lunghezze 0 e 1 come base. Se $|w| = 0$ o $|w| = 1$, allora w è ϵ, 0, o 1. Dato che esistono le produzioni $P \to \epsilon$, $P \to 0$ e $P \to 1$, concludiamo che $P \overset{*}{\Rightarrow} w$ in tutti i casi di base.

INDUZIONE Supponiamo che $|w| \geq 2$. Poiché $w = w^R$, w deve iniziare e finire con lo stesso simbolo. In altri termini $w = 0x0$ oppure $w = 1x1$. Oltre a ciò x deve essere palindroma, ossia $x = x^R$. Si noti che abbiamo bisogno che $|w| \geq 2$ per concludere che esistono due simboli distinti agli estremi di w.

Se $w = 0x0$ invochiamo l'ipotesi induttiva per sostenere che $P \overset{*}{\Rightarrow} x$. Allora esiste una derivazione di w da P, cioè $P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$. Se $w = 1x1$ il ragionamento è lo stesso, ma usiamo la produzione $P \to 1P1$ al primo passo. In entrambi i casi concludiamo che w è in $L(G_{pal})$. La dimostrazione è completa.

(Solo se) Ora assumiamo che w sia in $L(G_{pal})$; cioè $P \overset{*}{\Rightarrow} w$. Dobbiamo concludere che w è palindroma. La dimostrazione è un'induzione sul numero dei passi in una derivazione di w da P.

BASE Se la derivazione è un solo passo, allora dobbiamo usare una delle tre produzioni che non abbiano P nel loro corpo. In altre parole la derivazione è $P \Rightarrow \epsilon$, $P \Rightarrow 0$ o $P \Rightarrow 1$. Dato che ϵ, 0 e 1 sono tutti palindromi, la base è dimostrata.

INDUZIONE Supponiamo ora che la derivazione compia $n + 1$ passi, dove $n \geq 1$, e l'enunciato sia vero per tutte le derivazioni di n passi. Ossia, se $P \overset{*}{\Rightarrow} x$ in n passi, allora x è un palindromo.

Consideriamo una derivazione di $(n + 1)$ passi, che deve essere della forma

$$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$$

oppure $P \Rightarrow 1P1 \overset{*}{\Rightarrow} 1x1 = w$, dato che $n + 1$ passi significa almeno due passi e le produzioni $P \to 0P0$ e $P \to 1P1$ sono le uniche che permettono passi aggiuntivi. Si osservi che in entrambi i casi $P \overset{*}{\Rightarrow} x$ in n passi.

Per l'ipotesi induttiva sappiamo che x è palindromo, cioè $x = x^R$. Ma se è così, allora anche $0x0$ e $1x1$ sono palindromi. Per esempio $(0x0)^R = 0x^R0 = 0x0$. Concludiamo che w è palindromo e completiamo così la dimostrazione. □

5.1.6 Forme sentenziali

Le derivazioni dal simbolo iniziale producono stringhe che hanno un ruolo speciale e che chiameremo "forme sentenziali". Ossia, se $G = (V, T, P, S)$ è una CFG, allora qualunque stringa α in $(V \cup T)^*$ tale che $S \overset{*}{\Rightarrow} \alpha$ è una *forma sentenziale*. Se $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$, allora α è una *forma sentenziale sinistra*; se $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$, allora α è una *forma sentenziale destra*. Si noti che il linguaggio $L(G)$ è formato dalle forme sentenziali che sono in T^*, cioè che consistono unicamente di terminali.

Esempio 5.8 Consideriamo la grammatica per le espressioni della Figura 5.2. La stringa $E * (I + E)$ è un esempio di forma sentenziale perché esiste una derivazione

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

Dato che all'ultimo passo viene sostituita la E centrale, questa derivazione non è né a sinistra né a destra.

Come esempio di forma sentenziale sinistra consideriamo $a * E$, con la derivazione a sinistra

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

La derivazione

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

mostra che $E * (E + E)$ è una forma sentenziale destra. □

La forma delle dimostrazioni sulle grammatiche

Il Teorema 5.7 è un tipico caso di dimostrazione che una grammatica definisce un particolare linguaggio, descritto informalmente. Si parte da un'ipotesi induttiva che enuncia le proprietà di cui sono dotate le stringhe derivate da ciascuna variabile. Nell'esempio in questione c'è una sola variabile, P. Dunque abbiamo dovuto soltanto affermare che le sue stringhe sono palindrome.

Si dimostra la parte "se": se una stringa w soddisfa l'enunciato informale sulle stringhe di una delle variabili A, allora $A \stackrel{*}{\Rightarrow} w$. Nell'esempio, dato che P è il simbolo iniziale, abbiamo dichiarato "$P \stackrel{*}{\Rightarrow} w$", dicendo che w appartiene al linguaggio della grammatica. Di solito la parte "se" si dimostra per induzione sulla lunghezza di w. Se ci sono k variabili l'enunciato induttivo da dimostrare ha k parti, che devono essere dimostrate per induzione mutua.

Si deve inoltre dimostrare la parte "solo-se": se $A \stackrel{*}{\Rightarrow} w$, allora w soddisfa l'enunciato informale sulle stringhe derivate dalla variabile A. Nell'esempio, dovendo trattare solo il simbolo iniziale P, abbiamo supposto che w fosse nel linguaggio di G_{pal} come equivalente di $P \stackrel{*}{\Rightarrow} w$. La dimostrazione di questa parte si compie di solito per induzione sul numero dei passi nella derivazione. Se la grammatica contiene produzioni in cui due o più variabili compaiono nelle stringhe derivate, allora una derivazione di n passi va scomposta in più parti, con una derivazione per ognuna delle variabili. Queste derivazioni possono avere meno di n passi, e si deve dunque compiere un'induzione supponendo l'enunciato valido per tutti i valori minori o uguali a n, come discusso nel Paragrafo 1.4.2.

5.1.7 Esercizi

Esercizio 5.1.1 Ideate una grammatica libera dal contesto per ognuno dei seguenti linguaggi.

* a) L'insieme $\{0^n 1^n \mid n \geq 1\}$, ossia l'insieme di tutte le stringhe di uno o più 0 seguiti da un uguale numero di 1.

*! b) L'insieme $\{a^i b^j c^k \mid i \neq j \text{ o } j \neq k\}$, ossia l'insieme delle stringhe di a seguite da un certo numero di b seguite da un certo numero di c, tali che il numero di a sia diverso dal numero di b o il numero di b sia diverso dal numero di c, o entrambi.

! c) L'insieme di tutte le stringhe di a e b che *non* sono della forma ww, cioè non sono uguali a una stringa ripetuta.

!! d) L'insieme di tutte le stringhe con un numero di 0 doppio rispetto al numero di 1.

Esercizio 5.1.2 La seguente grammatica genera il linguaggio dell'espressione regolare $\mathbf{0^*1(0+1)^*}$:

$$\begin{array}{lcl} S & \rightarrow & A1B \\ A & \rightarrow & 0A \mid \epsilon \\ B & \rightarrow & 0B \mid 1B \mid \epsilon \end{array}$$

Scrivete le derivazioni a sinistra e a destra delle seguenti stringhe:

* a) 00101

b) 1001

c) 00011.

! **Esercizio 5.1.3** Dimostrate che ogni linguaggio regolare è un linguaggio libero dal contesto. *Suggerimento*: costruite una CFG per induzione sul numero di operatori nell'espressione regolare.

! **Esercizio 5.1.4** Una CFG è detta *lineare a destra* se il corpo di ogni produzione ha al massimo una variabile, e la variabile si trova all'estremità destra. In altre parole tutte le produzioni di una grammatica lineare a destra sono della forma $A \rightarrow wB$ o $A \rightarrow w$, dove A e B sono variabili e w una stringa di zero o più terminali.

a) Dimostrate che ogni grammatica lineare a destra genera un linguaggio regolare. *Suggerimento*: costruite un ϵ-NFA che simula derivazioni a sinistra, usando il suo stato per rappresentare l'unica variabile nella forma sentenziale sinistra corrente.

b) Mostrate che ogni linguaggio regolare ha una grammatica lineare a destra. *Suggerimento*: cominciate da un DFA e fate in modo che le variabili della grammatica rappresentino gli stati.

*! **Esercizio 5.1.5** Sia $T = \{0, 1, (,), +, *, \emptyset, e\}$. Possiamo considerare T come l'insieme dei simboli usati nelle espressioni regolari sull'alfabeto $\{0, 1\}$; l'unica differenza è che si usa e al posto del simbolo ϵ per evitare confusione in ciò che segue. Il vostro compito è definire una CFG, con T come insieme di terminali, che generi esattamente le espressioni regolari con alfabeto $\{0, 1\}$.

Esercizio 5.1.6 Abbiamo definito la relazione $\overset{*}{\Rightarrow}$ con una base "$\alpha \Rightarrow \alpha$" e un'induzione che dice "$\alpha \overset{*}{\Rightarrow} \beta$ e $\beta \Rightarrow \gamma$ implicano $\alpha \overset{*}{\Rightarrow} \gamma$". La relazione $\overset{*}{\Rightarrow}$ può essere definita in altri modi che a loro volta equivalgono a dire "$\overset{*}{\Rightarrow}$ significa zero o più $\Rightarrow$-passi". Dimostrate i seguenti enunciati.

a) $\alpha \overset{*}{\Rightarrow} \beta$ se e solo se esiste una sequenza di una o più stringhe

$$\gamma_1, \gamma_2, \dots, \gamma_n$$

tali che $\alpha = \gamma_1$, $\beta = \gamma_n$, e per $i = 1, 2, \dots, n-1$ abbiamo $\gamma_i \Rightarrow \gamma_{i+1}$.

b) Se $\alpha \overset{*}{\Rightarrow} \beta$, e $\beta \overset{*}{\Rightarrow} \gamma$, allora $\alpha \overset{*}{\Rightarrow} \gamma$. *Suggerimento*: procedete per induzione sul numero dei passi nella derivazione $\beta \overset{*}{\Rightarrow} \gamma$.

! **Esercizio 5.1.7** Consideriamo la CFG G definita dalle produzioni:

$$S \to aS \mid Sb \mid a \mid b$$

a) Dimostrate per induzione sulla lunghezza della stringa che nessuna stringa in $L(G)$ ha ba come sottostringa.

b) Descrivete $L(G)$ in termini informali e giustificate la vostra risposta servendovi della parte (a).

!! **Esercizio 5.1.8** Considerate la CFG G definita dalle produzioni

$$S \to aSbS \mid bSaS \mid \epsilon$$

Dimostrate che $L(G)$ è l'insieme di tutte le stringhe con lo stesso numero di a e di b.

5.2 Alberi sintattici

La rappresentazione ad albero delle derivazioni si è rivelata particolarmente utile. L'albero mostra in modo chiaro come i simboli di una stringa terminale sono raccolti in sottostringhe, ciascuna appartenente al linguaggio di una delle variabili della grammatica. Forse è ancora più importante che un albero di questo tipo, detto "albero sintattico" (*parse tree*), sia la struttura dati ideale per rappresentare il programma sorgente in un compilatore. La struttura ad albero del programma sorgente facilita la traduzione del programma stesso in codice eseguibile, delegando in modo naturale il processo di traduzione a funzioni ricorsive.

In questo paragrafo presentiamo gli alberi sintattici e dimostriamo che la loro esistenza è strettamente legata alle derivazioni e alle inferenze ricorsive. Studieremo poi la questione dell'ambiguità nelle grammatiche e nei linguaggi, che costituisce un'applicazione importante degli alberi sintattici. In alcune grammatiche una stringa terminale può avere più di un albero sintattico; ciò rende una grammatica inadatta a un linguaggio di programmazione perché il compilatore non sarebbe in grado di stabilire la struttura di certi programmi, e quindi non potrebbe dedurre con sicurezza il codice eseguibile appropriato.

Terminologia relativa agli alberi

Supponiamo che il lettore conosca già la nozione di albero e che le definizioni più comuni in quest'ambito gli siano familiari. Quanto segue sarà tuttavia un utile ripasso terminologico.

- Gli alberi sono collezioni di *nodi*, con una relazione *genitore-figlio*. Un nodo ha al massimo un genitore, disegnato sopra il nodo, e zero o più figli, disegnati al di sotto. Una linea collega un genitore a ogni figlio. Le Figure 5.4, 5.5 e 5.6 sono esempi di alberi.
- Esiste un unico nodo senza genitori, posto alla sommità dell'albero: la *radice*. I nodi privi di figli sono detti *foglie*. I nodi che non sono foglie sono *nodi interni*.
- Il figlio di un figlio di un $\cdots$ di un nodo è un *discendente* del nodo. Il genitore di un genitore di un $\cdots$ è un *antenato*. Ogni nodo è discendente e antenato di se stesso.
- I figli di un nodo vengono ordinati da sinistra e disegnati di conseguenza. Se il nodo N è a sinistra del nodo M, allora si considera che tutti i discendenti di N siano alla sinistra di tutti i discendenti di M.

5.2.1 Costruzione di alberi sintattici

Fissiamo una grammatica $G = (V, T, P, S)$. Gli *alberi sintattici* di G sono alberi che soddisfano le seguenti condizioni.

1. Ciascun nodo interno è etichettato da una variabile in V.
2. Ciascuna foglia è etichettata da una variabile, da un terminale, o da ϵ. Se una foglia è etichettata ϵ, deve essere l'unico figlio del suo genitore.
3. Se un nodo interno è etichettato A e i suoi figli sono etichettati, a partire da sinistra,

$$X_1, X_2, \ldots, X_k$$

allora $A \rightarrow X_1X_2 \cdots X_k$ è una produzione in P. Si noti che un X può essere ϵ solo nel caso in cui è l'etichetta di un figlio unico, e quindi $A \rightarrow \epsilon$ è una produzione di G.

Esempio 5.9 La Figura 5.4 presenta un albero sintattico per la grammatica delle espressioni della Figura 5.2. La radice è etichettata dalla variabile E. La produzione applicata alla radice è $E \to E + E$: i tre figli della radice hanno, a partire da sinistra, le etichette E, $+$, ed E. Per il figlio più a sinistra della radice si applica la produzione $E \to I$: il nodo ha un solo figlio, etichettato I. □

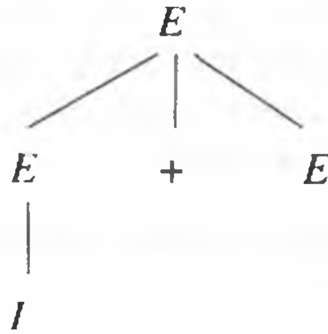


Figura 5.4 Un albero sintattico che illustra la derivazione di $I + E$ da E.

Esempio 5.10 La Figura 5.5 presenta un albero sintattico per la grammatica delle palindrome della Figura 5.1. Alla radice si applica la produzione $P \to 0P0$ e al figlio di mezzo della radice $P \to 1P1$. Al livello più basso è stata applicata la produzione $P \to \epsilon$. Questo caso, in cui il nodo etichettato dalla testa di una produzione ha un unico figlio, etichettato ϵ, è il solo in cui un nodo etichettato ϵ può fare la sua comparsa in un albero sintattico. □

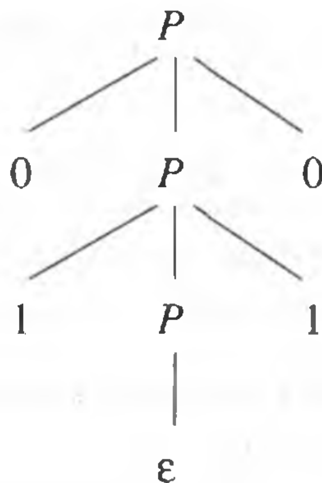


Figura 5.5 Un albero sintattico che illustra la derivazione $P \overset{*}{\Rightarrow} 0110$.

5.2.2 Il prodotto di un albero sintattico

Se concateniamo le foglie di un albero sintattico a partire da sinistra otteniamo una stringa, detta il *prodotto* dell'albero, che è sempre una stringa derivata dalla variabile della radice. Dimostreremo questa asserzione fra poco. Di particolare importanza sono gli alberi sintattici che soddisfano queste due condizioni.

1. Il prodotto è una stringa terminale. In questo caso tutte le foglie sono etichettate da un terminale o da ϵ.

2. La radice è etichettata dal simbolo iniziale.

Questi sono gli alberi sintattici i cui prodotti sono stringhe nel linguaggio della grammatica associata. Fra breve dimostreremo che si può descrivere il linguaggio di una grammatica anche come insieme dei prodotti degli alberi sintattici che hanno il simbolo iniziale alla radice e una stringa terminale come prodotto.

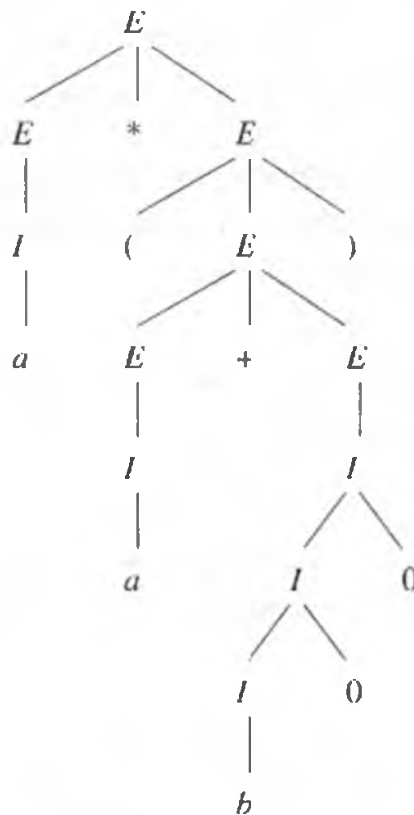


Figura 5.6 Un albero sintattico che illustra come $a * (a + b00)$ sia nel linguaggio della grammatica delle espressioni.

Esempio 5.11 La Figura 5.6 è un esempio di albero con una stringa terminale come prodotto e un simbolo iniziale alla radice; esso è basato sulla grammatica delle espressioni

presentata nella Figura 5.2. Il prodotto di quest'albero è la stringa $a * (a + b00)$, derivata nell'Esempio 5.5. Vedremo che questo particolare albero sintattico è una rappresentazione di quella derivazione. ☐

5.2.3 Inferenza, derivazioni e alberi sintattici

Ognuna delle nozioni presentate finora per descrivere il funzionamento di una grammatica dà essenzialmente gli stessi risultati sulle stringhe. In altre parole, data una grammatica $G = (V, T, P, S)$, dimostriamo che i seguenti enunciati si equivalgono:

1. la procedura di inferenza ricorsiva stabilisce che la stringa terminale w è nel linguaggio della variabile A
2. $A \overset{*}{\Rightarrow} w$
3. $A \underset{lm}{\overset{*}{\Rightarrow}} w$
4. $A \underset{rm}{\overset{*}{\Rightarrow}} w$
5. esiste un albero sintattico con radice A e prodotto w.

Se escludiamo l'inferenza ricorsiva, definita solo per stringhe terminali, le altre condizioni – l'esistenza di derivazioni generiche, a sinistra o a destra, e di alberi sintattici – sono equivalenti anche se w contiene variabili.

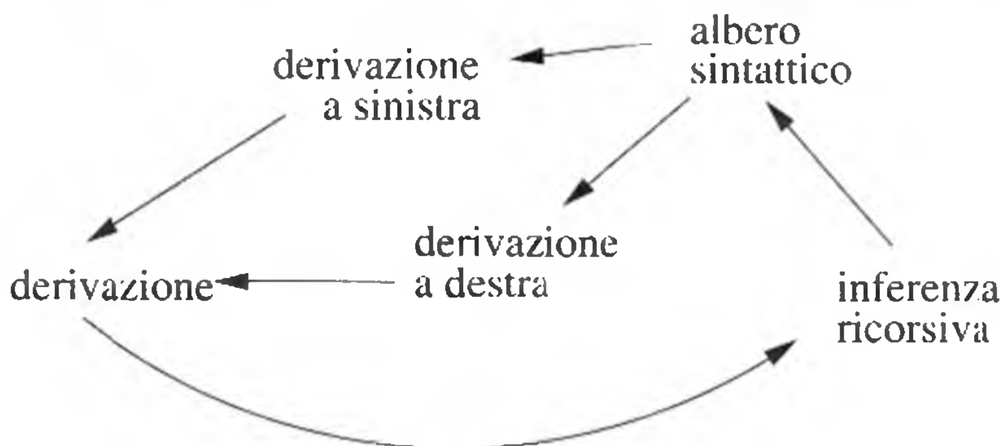


Figura 5.7 Dimostrazione dell'equivalenza di alcuni enunciati sulle grammatiche.

Dimostriamo queste equivalenze secondo lo schema della Figura 5.7. Ogni arco nel diagramma denota un teorema secondo il quale, se w soddisfa la condizione nella coda dell'arco, allora soddisfa anche la condizione alla testa. Per esempio nel Teorema 5.12 dimostreremo che, se si deduce che w è nel linguaggio di A per inferenza ricorsiva, allora esiste un albero sintattico con radice A e prodotto w.

Due degli archi sono molto semplici e se ne tralascerà dunque la dimostrazione formale. Se w ha una derivazione a sinistra da A, allora ha di certo una derivazione da A, dato che una derivazione a sinistra *è* una derivazione. Analogamente, se w ha una derivazione a destra, ha di certo una derivazione. Passiamo ora alla dimostrazione dei passi più ardui dell'equivalenza.

5.2.4 Dalle inferenze agli alberi

Teorema 5.12 Sia $G = (V, T, P, S)$ una CFG. Se la procedura di inferenza ricorsiva indica che la stringa terminale w è nel linguaggio della variabile A, allora esiste un albero sintattico con radice A e prodotto w.

DIMOSTRAZIONE La dimostrazione è un'induzione sul numero dei passi usati per dedurre che w è nel linguaggio di A.

BASE Un solo passo. In questo caso deve essere stata usata soltanto la base della procedura di inferenza. Di conseguenza deve esistere una produzione $A \to w$. L'albero della Figura 5.8, in cui esiste una sola foglia per ogni posizione di w, soddisfa le condizioni degli alberi sintattici per la grammatica G, e ha evidentemente prodotto w e radice A. Nel caso speciale che $w = \epsilon$, l'albero ha una foglia singola etichettata ϵ, ed è quindi un albero sintattico lecito, con radice A e prodotto w.

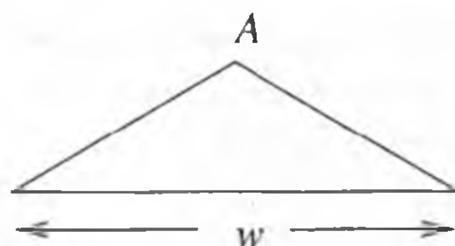


Figura 5.8 Albero costruito nel caso di base del Teorema 5.12.

INDUZIONE Supponiamo di aver dedotto che w è nel linguaggio di A dopo $n + 1$ passi di inferenza, e che l'enunciato del teorema sia valido per tutte le stringhe x e variabili B tali che l'appartenenza di x al linguaggio B si deduca in n, o meno, passi di inferenza. Consideriamo l'ultimo passo dell'inferenza che w è nel linguaggio di A. Questa inferenza impiega una certa produzione per A, poniamo $A \to X_1X_2 \cdots X_k$, dove ogni X_i è una variabile oppure un terminale.

Possiamo scomporre w in $w_1w_2 \cdots w_k$ soddisfacendo le seguenti clausole.

1. Se X_i è un terminale, allora $w_i = X_i$; cioè w_i consiste solamente di questo terminale.

2. Se X_i è una variabile, allora w_i è una stringa di cui è stata precedentemente dedotta l'appartenenza al linguaggio di X_i. In altri termini l'inferenza relativa a w_i ha richiesto al massimo n degli $n + 1$ passi dell'inferenza per la quale w si trova nel linguaggio di A. Non richiede tutti gli $n+1$ passi perché il passo finale, che si avvale della produzione $A \rightarrow X_1 X_2 \cdots X_k$, non è sicuramente parte dell'inferenza di w_i. Di conseguenza possiamo applicare l'ipotesi induttiva a w_i e X_i, e concludere che esiste un albero sintattico con prodotto w_i e radice X_i.

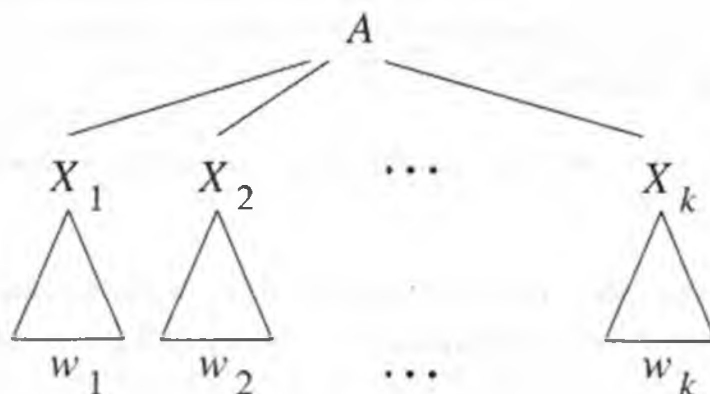


Figura 5.9 L'albero usato nella parte induttiva della dimostrazione del Teorema 5.12.

Costruiamo poi un albero con radice A e prodotto w, come suggerito nella Figura 5.9. C'è una radice etichettata A, i cui figli sono $X_1, X_2, \ldots, X_k$. Si tratta di una scelta valida, dato che $A \rightarrow X_1 X_2 \cdots X_k$ è una produzione di G.

Il nodo di ciascuna X_i diventa la radice di un sottoalbero con prodotto w_i. Nel caso (1), in cui X_i è un terminale, questo sottoalbero è un albero banale, con un solo nodo, etichettato X_i. Il sottoalbero consiste quindi in un solo nodo, figlio della radice. Poiché nel caso (1) $w_i = X_i$, la condizione che il prodotto del sottoalbero sia w_i viene soddisfatta.

Nel caso (2) X_i è una variabile. Invochiamo allora l'ipotesi induttiva per sostenere che esiste un albero con radice X_i e prodotto w_i. Nella Figura 5.9 quest'albero è agganciato al nodo di X_i.

L'albero costruito in questo modo ha radice A e prodotto formato dai prodotti dei sottoalberi, concatenati da sinistra a destra. La stringa è $w_1 w_2 \cdots w_k$, ovvero w. □

5.2.5 Dagli alberi alle derivazioni

Vediamo ora come si costruisce una derivazione a sinistra a partire da un albero sintattico. Il metodo per costruire una derivazione a destra si avvale degli stessi princìpi e quindi non lo tratteremo. Per capire come si costruiscono le derivazioni, consideriamo in primo luogo

come la derivazione di una stringa da una variabile può essere incorporata in un'altra derivazione. Illustriamo il punto con un esempio.

Esempio 5.13 Consideriamo ancora una volta la grammatica delle espressioni della Figura 5.2. Si può verificare facilmente che esiste una derivazione

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

Di conseguenza per tutte le stringhe α e β è vero anche che

$$\alpha E\beta \Rightarrow \alpha I\beta \Rightarrow \alpha Ib\beta \Rightarrow \alpha ab\beta$$

La giustificazione è data dal fatto che possiamo applicare le stesse produzioni sostituendo ogni testa con il relativo corpo, sia nel contesto di α e β sia isolatamente.[1]

Per esempio, se abbiamo una derivazione che comincia con $E \Rightarrow E+E \Rightarrow E+(E)$, possiamo applicare la derivazione di ab dalla seconda E trattando "E + (" come α e")" come β. La derivazione può continuare così:

$$E+(E) \Rightarrow E+(I) \Rightarrow E+(Ib) \Rightarrow E+(ab)$$

□

Ora possiamo dimostrare un teorema che ci permette di convertire un albero sintattico in una derivazione a sinistra. La dimostrazione è un'induzione sull'*altezza* dell'albero, ovvero sulla lunghezza massima di un cammino che parte dalla radice e procede verso il basso fino a una foglia passando attraverso i discendenti. Per esempio l'altezza dell'albero della Figura 5.6 è 7. In quest'albero il cammino più lungo dalla radice a una foglia raggiunge la foglia etichettata b. Si noti che, per convenzione, la lunghezza dei cammini conta gli archi, non i nodi, per cui un cammino che consista di un nodo singolo è di lunghezza 0.

Teorema 5.14 Sia $G = (V, T, P, S)$ una CFG, e supponiamo che esista un albero sintattico con radice etichettata da una variabile A e con prodotto w, dove w è in T^*. Allora esiste una derivazione a sinistra $A \underset{lm}{\overset{*}{\Rightarrow}} w$ nella grammatica G.

DIMOSTRAZIONE Svolgiamo un'induzione sull'altezza dell'albero.

BASE La base corrisponde all'altezza 1, la più bassa che un albero sintattico con prodotto fatto di terminali può avere. In questo caso l'albero è simile a quello della Figura 5.8, con una radice etichettata A e figli che formano w, da sinistra a destra. Poiché quest'albero è un albero sintattico, $A \rightarrow w$ dev'essere una produzione. Dunque $A \underset{lm}{\Rightarrow} w$ è una derivazione a sinistra, di un solo passo, di w da A.

[1] In effetti la locuzione "libero dal contesto" deriva appunto dalla possibilità di sostituire stringhe con variabili senza tener conto del contesto. Una classe più generale di grammatiche, dette "dipendenti dal contesto", ammette sostituzioni solo se determinate stringhe sono presenti sia a sinistra sia a destra. Nella pratica odierna le grammatiche dipendenti dal contesto non sono molto rilevanti.

INDUZIONE Un albero di altezza n, con $n > 1$, è simile a quello della Figura 5.9. Esso ha una radice etichettata A, con figli etichettati $X_1, X_2, \ldots, X_k$ a partire da sinistra. Le X sono terminali oppure variabili.

1. Se X_i è un terminale, definiamo w_i come la stringa formata solamente da X_i.

2. Se X_i è una variabile, allora dev'essere la radice di un sottoalbero con prodotto fatto di terminali, che chiameremo w_i. Si noti che in questo caso il sottoalbero è di altezza inferiore a n, dunque possiamo applicare l'ipotesi induttiva. In altre parole esiste una derivazione a sinistra $X_i \underset{lm}{\overset{*}{\Rightarrow}} w_i$.

Si osservi che $w = w_1 w_2 \cdots w_k$.

Costruiamo una derivazione a sinistra di w, partendo dal passo $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_k$. Allora, per ogni $i = 1, 2, \ldots, k$, mostriamo in quest'ordine che

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

La dimostrazione è un'altra induzione, stavolta su i. Per la base, $i = 0$, sappiamo già che $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_k$. Per l'induzione ipotizziamo che

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k$$

a) Se X_i è un terminale, non facciamo nulla. In seguito però considereremo X_i come la stringa terminale w_i. Perciò abbiamo già

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

b) Se X_i è una variabile, continuiamo con una derivazione di w_i da X_i, nel contesto della derivazione che si sta costruendo. In altre parole, se la derivazione è

$$X_i \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \cdots \underset{lm}{\Rightarrow} w_i$$

procediamo con

$$\begin{array}{l}
w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
w_1 w_2 \cdots w_{i-1} \alpha_1 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
w_1 w_2 \cdots w_{i-1} \alpha_2 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
\cdots \\
w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k
\end{array}$$

Il risultato è una derivazione $A \overset{*}{\underset{lm}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} \cdots X_k$.

Quando $i = k$, il risultato è una derivazione a sinistra di w da A. □

Esempio 5.15 Costruiamo la derivazione a sinistra per l'albero della Figura 5.6. Ci limitiamo al passo finale, in cui costruiamo la derivazione per l'intero albero dalle derivazioni che corrispondono ai sottoalberi della radice. In altri termini assumiamo che, applicando ricorsivamente il metodo del Teorema 5.14, abbiamo dedotto che il sottoalbero radicato nel primo figlio della radice ha la derivazione a sinistra $E \underset{lm}{\Rightarrow} I \underset{lm}{\Rightarrow} a$, mentre il sottoalbero radicato nel terzo figlio della radice ha la derivazione a sinistra

$$E \underset{lm}{\Rightarrow} (E) \underset{lm}{\Rightarrow} (E+E) \underset{lm}{\Rightarrow} (I+E) \underset{lm}{\Rightarrow} (a+E) \underset{lm}{\Rightarrow}$$

$$(a+I) \underset{lm}{\Rightarrow} (a+I0) \underset{lm}{\Rightarrow} (a+I00) \underset{lm}{\Rightarrow} (a+b00)$$

Per costruire una derivazione a sinistra per l'intero albero, cominciamo con il passo alla radice: $E \underset{lm}{\Rightarrow} E * E$. Sostituiamo poi la prima E secondo la sua derivazione, facendo seguire ogni passo da $*E$ per rispettare il contesto in cui è usata la derivazione. La derivazione a sinistra si presenta dunque finora come

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E$$

Dato che il simbolo * nella produzione usata alla radice non richiede alcuna derivazione, abbiamo già trattato i primi due figli della radice. Completiamo la derivazione applicando la derivazione $E \overset{*}{\underset{lm}{\Rightarrow}} (a+b00)$ in un contesto in cui è preceduta da $a*$ e seguita dalla stringa vuota. Questa derivazione è già stata trattata nell'Esempio 5.6:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow}$$

$$a*(E) \underset{lm}{\Rightarrow} a*(E+E) \underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} a*(a+E) \underset{lm}{\Rightarrow}$$

$$a*(a+I) \underset{lm}{\Rightarrow} a*(a+I0) \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00)$$

□

Un teorema analogo ci permette di convertire un albero in una derivazione a destra. La costruzione di una derivazione a destra a partire da un albero è molto simile alla costruzione di una derivazione a sinistra. Dopo essere partiti con il passo $A \underset{rm}{\Rightarrow} X_1 X_2 \cdots X_k$, espandiamo prima X_k usando una derivazione a destra, poi espandiamo X_{k-1}, e così via, fino a X_1. Possiamo quindi enunciare il teorema senza darne la dimostrazione.

Teorema 5.16 Sia $G = (V, T, P, S)$ una CFG, e supponiamo che esista un albero sintattico con radice etichettata da una variabile A e con prodotto w, dove w è in T^*. Allora esiste una derivazione a destra $A \overset{*}{\underset{rm}{\Rightarrow}} w$ nella grammatica G. □

5.2.6 Dalle derivazioni alle inferenze ricorsive

Completiamo ora il ciclo suggerito dalla Figura 5.7 mostrando che, se esiste una derivazione $A \overset{*}{\Rightarrow} w$ per una CFG, tramite la procedura di inferenza ricorsiva si può stabilire che w è nel linguaggio di A . Prima di passare al teorema e alla dimostrazione, facciamo qualche importante considerazione sulle derivazioni.

Sia data una derivazione $A \Rightarrow X_1X_2 \cdots X_k \overset{*}{\Rightarrow} w$. Allora è possibile scomporre w in segmenti, $w = w_1w_2 \cdots w_k$, tali che $X_i \overset{*}{\Rightarrow} w_i$. Si noti che se X_i è un terminale, allora $w_i = X_i$ e la derivazione consta di zero passi. Dimostrare questa osservazione non è difficile. Si dimostra per induzione sul numero di passi della derivazione che se $X_1X_2 \cdots X_k \overset{*}{\Rightarrow} \alpha$, allora tutte le posizioni di α provenienti dall'espansione di X_i si trovano a sinistra di tutte le posizioni che provengono dall'espansione di X_j, se $i < j$.

Se X_i è una variabile, possiamo ottenere la derivazione $X_i \overset{*}{\Rightarrow} w_i$ partendo dalla derivazione $A \overset{*}{\Rightarrow} w$ ed eliminando:

a) tutte le posizioni delle forme sentenziali che si trovano a sinistra o a destra delle posizioni derivate da X_i

b) tutti i passi che non sono rilevanti per la derivazione di w_i da X_i.

Illustriamo la procedura con un esempio.

Esempio 5.17 Consideriamo la seguente derivazione per la grammatica delle espressioni della Figura 5.2:

$$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow I * E + E \Rightarrow I * I + E \Rightarrow$$

$$I * I + I \Rightarrow a * I + I \Rightarrow a * b + I \Rightarrow a * b + a$$

Consideriamo la terza forma sentenziale, $E * E + E$, e la E mediana in questa forma.[2]

Partendo da $E * E + E$ possiamo seguire i passi della derivazione descritta sopra, ma eliminando le posizioni derivate da $E*$ a sinistra della E centrale o da $+E$ alla sua destra. I passi della derivazione diventano allora E, E, I, I, I, b, b. In altre parole il primo passo non cambia la E centrale, il successivo la trasforma in I, i due passi a seguire la lasciano

[2]Nel discutere la ricerca di sottoderivazioni da derivazioni più grandi, abbiamo ipotizzato di trattare una variabile nella terza forma sentenziale di una derivazione data. L'idea è comunque applicabile a una variabile in qualunque passo di una derivazione.

immutata come I, quello dopo la trasforma in b, e il passo finale non muta quanto derivato dalla E centrale.

Se prendiamo solo i passi che trasformano ciò che deriva dalla E centrale, la sequenza di stringhe E, E, I, I, I, b, b diventa la derivazione $E \Rightarrow I \Rightarrow b$, che descrive correttamente come la E centrale si trasforma nel corso della derivazione completa. □

Teorema 5.18 Sia $G = (V, T, P, S)$ una CFG, e supponiamo che esista una derivazione $A \overset{*}{\underset{G}{\Rightarrow}} w$, dove w è in T^*. Allora la procedura di inferenza ricorsiva, applicata a G, determina che w è nel linguaggio della variabile A.

DIMOSTRAZIONE La dimostrazione è un'induzione sulla lunghezza della derivazione $A \overset{*}{\Rightarrow} w$.

BASE Se la derivazione è formata da un unico passo, allora $A \to w$ dev'essere una produzione. Dato che w consiste solo di terminali, si conclude che w è nel linguaggio di A nella base della procedura di inferenza ricorsiva.

INDUZIONE Supponiamo che la derivazione compia $n + 1$ passi e che l'enunciato sia valido per ogni derivazione di n, o meno, passi. Scriviamo la derivazione come $A \Rightarrow X_1X_2 \cdots X_k \overset{*}{\Rightarrow} w$. Allora possiamo scomporre w in $w = w_1w_2 \cdots w_k$, come abbiamo visto prima del teorema, soddisfacendo le seguenti clausole.

a) Se X_i è un terminale, allora $w_i = X_i$.

b) Se X_i è una variabile, allora $X_i \overset{*}{\Rightarrow} w_i$. Dato che sicuramente il primo passo della derivazione $A \overset{*}{\Rightarrow} w$ non è parte della derivazione $X_i \overset{*}{\Rightarrow} w_i$, sappiamo che questa derivazione ha al massimo n passi. Perciò vi si può applicare l'ipotesi induttiva, e concludere che w_i è nel linguaggio di X_i.

Abbiamo ora una produzione $A \to X_1X_2 \cdots X_k$, dove w_i è uguale a X_i, oppure se ne conosce l'appartenenza al linguaggio di X_i. Nella successiva iterazione della procedura di inferenza ricorsiva scopriremo che $w_1w_2 \cdots w_k$ è nel linguaggio di A. Dato che $w_1w_2 \cdots w_k = w$, abbiamo mostrato che si è dedotta l'appartenenza di w al linguaggio di A. □

5.2.7 Esercizi

Esercizio 5.2.1 Scrivete gli alberi sintattici per la grammatica e ognuna delle stringhe nell'Esercizio 5.1.2.

! **Esercizio 5.2.2** Sia G una CFG priva di produzioni con ϵ come membro destro. Dimostrate che se w è in $L(G)$, la sua lunghezza è n, e ha una derivazione di m passi, allora w ha un albero sintattico con $n + m$ nodi.

! **Esercizio 5.2.3** Supponiamo che tutto sia come nell'Esercizio 5.2.2, ma che G possa avere alcune produzioni con ϵ come membro destro. Dimostrate che un albero sintattico per una stringa w diversa da ϵ può avere fino a $n + 2m - 1$ nodi, ma non di più.

! **Esercizio 5.2.4** Nel Paragrafo 5.2.6 abbiamo detto che se $X_1 X_2 \cdots X_k \stackrel{*}{\Rightarrow} \alpha$, allora tutte le posizioni di α che derivano dall'espansione di X_i si trovano a sinistra di tutte le posizioni che derivano dall'espansione di X_j, se $i < j$. Dimostratelo. *Suggerimento*: svolgete un'induzione sul numero dei passi nella derivazione.

5.3 Applicazioni delle grammatiche libere dal contesto

Inizialmente le grammatiche libere dal contesto erano state concepite da N. Chomsky come un modo per descrivere il linguaggio naturale. Quelle speranze sono state deluse. D'altra parte il moltiplicarsi di concetti definiti ricorsivamente nell'informatica ha reso le CFG sempre più utili per descriverne esempi. Tratteremo brevemente due casi, uno datato e uno recente.

1. Le grammatiche si impiegano per descrivere i linguaggi di programmazione, ma un aspetto più importante è la possibilità di trasformare automaticamente una CFG in un *parser*, cioè in quel componente di un compilatore che rileva la struttura del codice sorgente e la rappresenta per mezzo di un albero sintattico. Questa è una delle prime applicazioni delle CFG, ed è in effetti uno dei primi casi in cui un'idea teorica si è fatta strada nella pratica dell'informatica.

2. È facile prevedere che lo sviluppo di XML (*eXtensible Markup Language*) agevolerà il commercio elettronico permettendo ai partner di condividere convenzioni sul formato degli ordini, delle descrizioni dei prodotti e di altri tipi di documenti. Una parte essenziale di XML è la DTD (*Document Type Definition*), in sostanza una grammatica libera dal contesto che descrive i *tag* ammessi e i modi in cui essi possono strutturarsi. I *tag* sono parole racchiuse fra parentesi angolari. Per esempio `<EM>` e `</EM>` vengono impiegate in HTML per delimitare un testo da porre in evidenza. I tag di XML non riguardano però l'impaginazione di un testo, ma il suo significato. Ad esempio una sequenza di caratteri da interpretare come un numero telefonico potrebbe essere racchiusa dai tag `<PHONE>` e `</PHONE>`.

5.3.1 Parser

La struttura di molti aspetti di un linguaggio di programmazione si può descrivere per mezzo di espressioni regolari. Nell'esempio 3.9 abbiamo discusso di come gli identificatori possano essere rappresentati in questo modo. D'altra parte ci sono anche aspetti

molto importanti di linguaggi di programmazione tipici che non si possono rappresentare con le sole espressioni regolari. Eccone due esempi.

Esempio 5.19 Di norma le parentesi si possono annidare e devono essere bilanciate, cioè dev'essere possibile associare una parentesi aperta a quella chiusa immediatamente alla sua destra, cancellarle, e ripetere il procedimento. Se in questo modo si eliminano tutte le parentesi, allora la stringa è bilanciata, altrimenti no. Esempi di stringhe di parentesi bilanciate sono $(())$, $()()$, $(()())$ e ϵ, mentre $)($ e $(()$ non lo sono.

La grammatica $G_{bal} = (\{B\}, \{(,)\}, P, B)$, in cui P consiste nelle produzioni

$$B \rightarrow BB \mid (B) \mid \epsilon$$

genera tutte e sole le stringhe di parentesi bilanciate. La prima produzione, $B \rightarrow BB$, afferma che la concatenazione di due stringhe di parentesi bilanciate è bilanciata; ciò ha senso perché possiamo abbinare le parentesi separatamente in ciascuna stringa. La seconda produzione, $B \rightarrow (B)$, afferma che, se poniamo una coppia di parentesi ai due capi di una stringa bilanciata, il risultato è bilanciato. Anche questa regola è sensata perché, se abbiniamo le parentesi nella stringa interna e le eliminiamo, rimangono solo la prima e l'ultima, adiacenti. La terza produzione, $B \rightarrow \epsilon$, serve da fondamento, e afferma che la stringa vuota è bilanciata.

Questo ragionamento informale dovrebbe convincerci che G_{bal} genera solo stringhe di parentesi bilanciate. Ci serve una dimostrazione dell'inverso: che ogni stringa di parentesi bilanciate è generata da questa grammatica. Non è difficile ideare una dimostrazione per induzione sulla lunghezza della stringa bilanciata; lasciamo al lettore l'esercizio.

Abbiamo già detto che l'insieme delle stringhe di parentesi bilanciate non è un linguaggio regolare. Ora lo dimostreremo. Se $L(G_{bal})$ fosse regolare, esisterebbe una costante n associata al linguaggio dal *pumping lemma*. Consideriamo la stringa bilanciata $w = (^n)^n$, formata da n parentesi aperte seguite da n parentesi chiuse corrispondenti. Se scomponiamo $w = xyz$ secondo il *pumping lemma*, y è formata da sole parentesi aperte, e quindi xz ha più parentesi chiuse che aperte. Questa stringa non è dunque bilanciata, contraddicendo l'ipotesi che il linguaggio delle parentesi bilanciate sia regolare. □

Naturalmente un linguaggio di programmazione non è fatto di sole parentesi, ma esse sono una parte fondamentale delle espressioni aritmetiche e condizionali. La grammatica della Figura 5.2 è più rappresentativa della struttura delle espressioni aritmetiche, anche se si limita a due operatori, + e *, e se comprende la struttura di dettaglio degli identificatori, che di solito è trattata dall'analizzatore lessicale di un compilatore, come abbiamo spiegato nel Paragrafo 3.3.2. Il linguaggio descritto nella Figura 5.2 non è però regolare. Per esempio, secondo questa grammatica, $(^n a)^n$ è un'espressione accettabile. Possiamo ricorrere al *pumping lemma* per provare che se il linguaggio fosse regolare anche una stringa da cui abbiamo eliminato alcune parentesi aperte, lasciando a e tutte le parentesi chiuse, sarebbe accettabile, mentre non lo è.

Molti aspetti di un tipico linguaggio di programmazione si comportano come parentesi bilanciate. In espressioni di vario genere compaiono innanzitutto le parentesi in senso proprio. Altri esempi sono i segnali di inizio e di fine dei blocchi di codice, come **begin** e **end** in Pascal o le parentesi graffe { . . . } nel C. Le parentesi graffe presenti in un programma C devono formare una sequenza bilanciata, con { come parentesi aperta e } come parentesi chiusa.

In alcuni casi compare anche una figura affine, in cui le "parentesi" possono essere bilanciate, con l'eccezione che ci possono essere parentesi aperte in eccesso. Ne è un esempio la struttura di **if** e **else** in C. Una clausola **if** può essere priva della clausola **else**, oppure può essere bilanciata da un **else** corrispondente. Ecco una grammatica che genera le sequenze ammissibili di **if** e **else**, rappresentati rispettivamente da i ed e:

$$S \rightarrow \epsilon \mid SS \mid iS \mid iSeS$$

Per esempio $ieie$, iie e iei sono sequenze lecite di **if** e **else**, e ciascuna è generata dalla grammatica. Esempi di sequenze illecite, non generate dalla grammatica, sono ei e $ieeii$.

Un modo semplice (lasciamo la verifica della sua correttezza come esercizio) per stabilire se una sequenza di i e di e è generata dalla grammatica consiste nel considerare gli e a partire da sinistra. Si cerca il primo i a sinistra del primo e. Se non c'è, la stringa non supera la prova e non appartiene al linguaggio. Se c'è, lo cancelliamo insieme all'e in esame. Se non ci sono altri e, la stringa passa la prova e appartiene al linguaggio. Se ce ne sono altri, si prende in esame il prossimo.

Esempio 5.20 Consideriamo la stringa iee. Il primo e è abbinato all'i alla sua sinistra. Cancellandoli si ottiene la stringa e. Poiché ci sono altri e, consideriamo il successivo. Alla sua sinistra non ci sono i: la verifica è fallita, e iee non appartiene al linguaggio. La conclusione è corretta perché in un programma C non ci possono essere più **else** di **if**.

Come altro esempio consideriamo $iieie$. Dopo aver abbinato il primo e con l'i alla sua sinistra, resta iie. Dopo aver abbinato l'e rimanente con l'i alla sua sinistra, resta i. Non ci sono altri e, quindi la verifica è riuscita. Anche questa conclusione è corretta perché la sequenza $iieie$ corrisponde a un programma C con una struttura simile a quella della Figura 5.10. L'algoritmo di accoppiamento rivela inoltre quale **if** corrisponde a un dato **else** e passa l'informazione al compilatore. Questa informazione è essenziale per generare la logica del controllo del flusso voluta dal programmatore. □

5.3.2 YACC: un generatore di parser

La generazione di un parser (o analizzatore sintattico, la funzione che crea alberi sintattici a partire dal codice sorgente) è stata istituzionalizzata dal comando YACC, presente in tutti i sistemi UNIX. L'input di YACC è una CFG, in una notazione che differisce da

```
if (Condizione) {
    ...
    if (Condizione) Istruzione;
    else Istruzione;
    ...
    if (Condizione) Istruzione;
    else Istruzione;
    ...
}
```

Figura 5.10 Una struttura if-else; i due **else** sono abbinati agli **if** precedenti; il primo **if** è isolato.

quella impiegata qui soltanto per alcuni dettagli. A ogni produzione è associata un'azione, cioè un frammento di codice C, eseguita quando si crea un nodo dell'albero sintattico che, insieme ai suoi figli, corrisponde alla produzione. Di solito l'azione consiste nel codice per costruire il nodo, ma in alcune applicazioni di YACC l'albero non viene di fatto costruito e l'azione è di altro tipo, come emettere un frammento di codice oggetto.

Esempio 5.21 La Figura 5.11 illustra un esempio di CFG nella notazione di YACC. La grammatica è quella della Figura 5.2. Abbiamo omesso le azioni, lasciando solo le parentesi graffe (obbligatorie) e la loro posizione nell'input di YACC.

```
Exp : Id              {...}
    | Exp '+' Exp     {...}
    | Exp '*' Exp     {...}
    | '(' Exp ')'     {...}
    ;
Id  : 'a'             {...}
    | 'b'             {...}
    | Id 'a'          {...}
    | Id 'b'          {...}
    | Id '0'          {...}
    | Id '1'          {...}
    ;
```

Figura 5.11 Esempio di grammatica nella notazione di YACC.

Notiamo alcune corrispondenze fra la notazione di YACC e la nostra.

- I due punti sono il simbolo delle produzioni, in luogo di $\rightarrow$.
- Le produzioni con la stessa testa sono riunite e i loro corpi sono separati dalla barra verticale. Noi ammettiamo questa convenzione in via facoltativa.
- L'elenco dei corpi di una stessa testa si chiude con il punto e virgola. Noi non abbiamo un simbolo di chiusura.
- I terminali sono racchiusi fra apici. Una coppia di apici può racchiudere più caratteri. Anche se l'esempio non lo mostra, YACC permette di definire anche terminali simbolici. La presenza di questi terminali nel codice sorgente è rilevata dall'analizzatore lessicale e segnalata al parser mediante il valore da esso restituito.
- Le stringhe di lettere e cifre non incluse fra apici sono nomi di variabili. Ci siamo serviti di questa possibilità per dare alle due variabili nomi descrittivi, `Exp` e `Id`, ma avremmo potuto chiamarle E e I.

□

5.3.3 Linguaggi di markup

Consideriamo ora una famiglia di linguaggi detti linguaggi di *markup*. Le "stringhe" di questi linguaggi sono documenti corredati di segnali, detti *tag*. I tag forniscono informazioni sul significato di porzioni del documento.

Uno dei più noti linguaggi di markup è l'HTML (*HyperText Markup Language*). Questo linguaggio ha due scopi principali: creare collegamenti fra documenti diversi e descrivere la veste grafica di un documento. Daremo solo un quadro semplificato della struttura di HTML; gli esempi che seguono dovrebbero comunque suggerirne la struttura e il modo in cui una CFG può sia descrivere i documenti HTML validi sia guidarne il trattamento, cioè la visualizzazione su uno schermo o sulla pagina stampata.

Esempio 5.22 La Figura 5.12(a) presenta un brano testuale contenente un elenco; la Figura 5.12(b) ne presenta l'espressione in HTML. Si può notare che l'HTML consiste di testo normale costellato di tag. I tag accoppiati sono della forma <x> e </x> per una stringa x.[3] Per esempio la coppia di tag `<EM>` e `</EM>` segnala che il testo racchiuso deve essere posto in risalto, cioè composto in corsivo o in un altro tipo di carattere. La coppia `<OL>` e `</OL>` segnala invece un elenco ordinato, cioè una enumerazione di voci.

Vediamo anche due esempi di tag singoli, `<P>` e `<LI>`, che introducono rispettivamente un capoverso e una voce d'elenco. HTML permette, anzi incoraggia, l'impiego dei tag di chiusura `</P>` e `</LI>` al termine di un capoverso e di una voce d'elenco, ma

[3] Talvolta il tag di apertura <x> reca più informazioni del semplice nome x, ma negli esempi non consideriamo questa possibilità.

non lo richiede. Per complicare un po' l'esempio di grammatica da sviluppare abbiamo omesso i tag di chiusura. □

Le cose che *detesto*:

1. il pane ammuffito

2. coloro che guidano lentamente nella corsia di sorpasso.

(a) Il testo come appare

```
<P>Le cose che <EM>detesto</EM>:
<OL>
<LI>il pane ammuffito
<LI>coloro che guidano lentamente
nella corsia di sorpasso.
</OL>
```

(b) Sorgente HTML

Figura 5.12 Un documento HTML e la sua versione stampata.

A un documento HTML sono associate diverse classi di stringhe. Elencheremo soltanto quelle necessarie a capire un testo come quello dell'Esempio 5.22. Per ogni classe introdurremo una variabile con un nome descrittivo.

1. $Text$ è una stringa di caratteri che si può interpretare alla lettera, cioè non contiene tag. Un esempio di elemento di tipo $Text$ nella Figura 5.12(a) è "il pane ammuffito".

2. $Char$ è una stringa fatta di un solo carattere lecito in un testo HTML. Notiamo che gli spazi sono compresi fra i caratteri.

3. Doc rappresenta documenti, che sono sequenze di "elementi". La definizione degli elementi, data in seguito, è mutuamente ricorsiva con quella di Doc.

4. $Element$ è una stringa di tipo $Text$, o una coppia di tag con il documento racchiuso, o un tag singolo seguito da un documento.

5. $ListItem$ è il tag `<LI>` seguito da un documento, che forma una voce di elenco.

6. $List$ è una sequenza di zero o più voci di elenco.

La Figura 5.13 illustra una CFG che descrive la struttura della parte di HTML trattata fin qui. Alla riga (1) si suggerisce che un carattere può essere "a" o "A", oppure un altro carattere compreso nel corredo di HTML. Le due produzioni alla riga (2) dicono che *Text* può essere la stringa vuota o un carattere lecito seguito da un testo. In altri termini *Text* consta di zero o più caratteri. Notiamo che $<$ e $>$ non sono caratteri leciti, ma sono rappresentati dalle sequenze `<` e `>`. In questo modo non possiamo inserire per errore un tag in *Text*.

1. *Char* → $a \mid A \mid \cdots$
2. *Text* → $\epsilon \mid$ *Char Text*
3. *Doc* → $\epsilon \mid$ *Element Doc*
4. *Element* → *Text* |
`<EM>` *Doc* `</EM>` |
`<P>` *Doc* |
`<OL>` *List* `</OL>` | $\cdots$
5. *ListItem* → `<LI>` *Doc*
6. *List* → $\epsilon \mid$ *ListItem List*

Figura 5.13 Porzione di una grammatica per HTML.

La riga (3) dice che un documento è una sequenza di zero o più "elementi". Secondo la riga (4) un elemento è un testo, un documento posto in risalto, un'apertura di paragrafo seguita da un documento, o un elenco. I punti di sospensione indicano altre produzioni per *Element*, corrispondenti agli altri tag di HTML. Alla riga (5) scopriamo che una voce di elenco è fatta dal tag `<LI>` seguito da un documento; infine la riga (6) dice che un elenco è una sequenza di zero o più voci.

Per alcuni aspetti di HTML non occorrono le grammatiche libere dal contesto: sono sufficienti le espressioni regolari. Per esempio le righe (1) e (2) della Figura 5.13 dicono che *Text* rappresenta lo stesso linguaggio dell'espressione regolare $(\mathbf{a} + \mathbf{A} + \cdots)^*$. Per altri aspetti, invece, le CFG sono necessarie. Per esempio le coppie formate da un tag di apertura e il corrispondente tag di chiusura, come `<EM>` e `</EM>`, si comportano come parentesi bilanciate, che sappiamo non regolari.

5.3.4 XML e DTD

Il fatto che l'HTML sia descritto da una grammatica non è in sé notevole. Dato che praticamente tutti i linguaggi di programmazione si possono descrivere per mezzo di una CFG, sarebbe sorprendente se ciò *non* valesse per l'HTML. Se invece ci occupiamo di un altro importante linguaggio di markup, XML, scopriamo che le CFG svolgono una parte cruciale nel suo impiego pratico.

Lo scopo di XML non è quello di descrivere l'apparenza di un documento; questo è compito di HTML. XML, invece, descrive la "semantica" di un testo. Per esempio un testo come "12 Maple St." sembra un indirizzo. Ma lo è veramente? In XML una frase che rappresenta un indirizzo è racchiusa fra tag appropriati. Per esempio:

```
<ADDR>12 Maple St.</ADDR>
```

D'altra parte non è ovvio che `<ADDR>` significhi un indirizzo stradale. Se per esempio il documento riguardasse l'allocazione di memoria, il tag `<ADDR>` potrebbe riferirsi a un indirizzo in memoria. Per chiarire il senso dei diversi tipi di tag e le strutture che possono trovarsi racchiuse fra coppie di tag, si possono sviluppare convenzioni, nella forma di DTD, condivise entro un ambito applicativo specifico.

Una DTD è in sostanza una grammatica libera dal contesto, con una propria notazione per descrivere variabili e produzioni. Nel prossimo esempio mostreremo una semplice DTD e introdurremo alcune regole linguistiche per descrivere DTD. Il linguaggio per le DTD ha a sua volta una grammatica libera dal contesto, ma non siamo interessati a descriverla. Il linguaggio per descrivere DTD è essenzialmente una notazione di tipo CFG: intendiamo vedere come usarlo per esprimere CFG.

La forma di una DTD è la seguente.

```
<!DOCTYPE nome-della-DTD [
    elenco di definizioni di elementi
]>
```

La definizione di un elemento ha invece questa forma.

```
<!ELEMENT nome-elemento (descrizione dell'elemento)>
```

Le descrizioni di elementi sono sostanzialmente espressioni regolari. Elenchiamo ora le basi di tali espressioni.

1. Nomi di altri elementi, a rappresentare il fatto che gli elementi di un tipo possono comparire in elementi di altro tipo, così come in HTML un testo in risalto può trovarsi in un elenco.

2. Il termine speciale #PCDATA, che denota qualsiasi testo non contenga tag di XML. Questo termine ha lo stesso compito della variabile $Text$ nell'Esempio 5.22.

Gli operatori ammessi sono i seguenti.

1. Il simbolo |, che denota l'unione, come nella notazione di UNIX delle espressioni regolari trattata nel Paragrafo 3.3.1.

2. La virgola, che denota la concatenazione.

3. Tre varianti dell'operatore di chiusura, come nel Paragrafo 3.3.1. Esse sono $*$, l'operatore usuale, che significa "zero o più occorrenze di"; $+$, che significa "una o più occorrenze di"; ?, che significa "zero o una occorrenza di".

Le parentesi possono associare gli operatori ai loro argomenti; altrimenti si applicano le solite regole di precedenza delle espressioni regolari.

Esempio 5.23 Supponiamo che alcuni produttori di computer concordino di definire una DTD per pubblicare sul Web le descrizioni dei PC in catalogo. La descrizione di un PC deve contenere un codice del prodotto e i dati caratteristici, per esempio la quantità di RAM, il numero e la dimensione dei dischi, e così via. La Figura 5.14 illustra un'ipotetica DTD, molto semplice, per personal computer.

```
<!DOCTYPE PcSpecs [
    <!ELEMENT PCS (PC*)>
    <!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
    <!ELEMENT MODEL (#PCDATA)>
    <!ELEMENT PRICE (#PCDATA)>
    <!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
    <!ELEMENT MANF (#PCDATA)>
    <!ELEMENT MODEL (#PCDATA)>
    <!ELEMENT SPEED (#PCDATA)>
    <!ELEMENT RAM (#PCDATA)>
    <!ELEMENT DISK (HARDDISK | CD | DVD)>
    <!ELEMENT HARDDISK (MANF, MODEL, SIZE)
    <!ELEMENT SIZE (#PCDATA)>
    <!ELEMENT CD (SPEED)>
    <!ELEMENT DVD (SPEED)>
]>
```

Figura 5.14 Una DTD per personal computer.

Il nome della DTD è PcSpecs. Il primo elemento, analogo al simbolo iniziale di una CFG, è PCS (elenco di specifiche di PC). La sua definizione, PC*, dice che una PCS è formata da zero o più voci di tipo PC.

Segue la definizione dell'elemento PC. Essa consiste nella concatenazione di cinque entità. Le prime quattro sono altri elementi, corrispondenti a modello, prezzo, tipo di processore e RAM del PC. Ciascuno deve comparire una volta, nell'ordine dato, perché la virgola rappresenta la concatenazione. L'ultimo costituente, DISK+, indica la presenza di una o più voci relative ai dischi.

Molti costituenti sono puramente testuali: MODEL, PRICE e RAM sono di questo tipo. L'elemento PROCESSOR, invece, è strutturato. Dalla sua definizione notiamo che esso consiste di produttore, modello e velocità, in quest'ordine; ogni elemento è testuale.

La voce più complessa è DISK. Innanzitutto un disco può essere un disco fisso, un CD o un DVD, come indicato dalla regola per l'elemento DISK, che consiste nell'OR di tre altri elementi. I dischi fissi, a loro volta, hanno una struttura in cui si specificano il produttore, il modello e la dimensione, mentre CD e DVD sono rappresentati solo dalla velocità.

La Figura 5.15 riporta un esempio di documento XML conforme alla DTD della Figura 5.14. Notiamo che ogni elemento è rappresentato nel documento da un tag con il nome dell'elemento e da un secondo tag con una sbarra obliqua in più, come in HTML. Per esempio osserviamo i tag <PCS>...</PCS> al livello più esterno. Fra i due tag si trova un elenco di voci, una per ogni PC venduto dal produttore; ne abbiamo riportata esplicitamente una sola.

Nella voce <PC> illustrata leggiamo senza difficoltà che il codice del modello è 4560, il prezzo è $2295, e il processore è un Intel Pentium a 800MHz. Il PC ha una RAM di 256 MB, un disco Maxtor Diamond di 30,5 GB e un lettore di CD-ROM 32x. Non è tanto importante che noi possiamo leggere queste informazioni, quanto che un programma possa leggere il documento e, guidato dalla grammatica nella DTD della Figura 5.14 letta in precedenza, sappia interpretarne opportunamente il contenuto. □

Il lettore ha forse notato che le regole per gli elementi in una DTD simile a quella della Figura 5.14 non somigliano alle produzioni delle grammatiche libere dal contesto. Molte sono nella forma corretta; per esempio

```
<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
```

è analoga alla produzione

$$Processor \rightarrow Manf\ Model\ Speed$$

D'altra parte la regola

```
<!ELEMENT DISK (HARDDISK | CD | DVD)>
```

```
<PCS>
    <PC>
        <MODEL>4560</MODEL>
        <PRICE>$2295</PRICE>
        <PROCESSOR>
            <MANF>Intel</MANF>
            <MODEL>Pentium</MODEL>
            <SPEED>800MHz</SPEED>
        </PROCESSOR>
        <RAM>256</RAM>
        <DISK><HARDDISK>
            <MANF>Maxtor</MANF>
            <MODEL>Diamond</MODEL>
            <SIZE>30.5Gb</SIZE>
        </HARDDISK></DISK>
        <DISK><CD>
            <SPEED>32x</SPEED>
        </CD></DISK>
    </PC>
    <PC>
        ...
    </PC>
</PCS>
```

Figura 5.15 Porzione di un documento conforme alla struttura della DTD nella Figura 5.14.

definisce `DISK` in modo difforme dal corpo di una produzione. In questo caso la soluzione è semplice: interpretiamo la regola come tre produzioni; la barra verticale svolge lo stesso compito affidatole nella notazione abbreviata di produzioni con testa comune, che abbiamo definito in precedenza. La regola è quindi equivalente a tre produzioni:

$$Disk \rightarrow HardDisk \mid Cd \mid Dvd$$

Il caso più difficile è

```
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

in cui il "corpo" contiene un operatore di chiusura. La soluzione consiste nel sostituire `DISK+` con una variabile, $Disks$, che genera, mediante due produzioni, uno o più esemplari della variabile $Disk$. Le produzioni equivalenti sono

$$Pc \rightarrow Model\ Price\ Processor\ Ram\ Disks$$
$$Disks \rightarrow Disk \mid Disk\ Disks$$

C'è una tecnica generale per trasformare una CFG con espressioni regolari come corpi delle produzioni in una CFG normale. Presentiamo l'idea informalmente; il lettore può cercare di formalizzare sia il significato di CFG aventi produzioni i cui corpi sono formati da espressioni regolari sia la prova che l'estensione non produce linguaggi al di là dei CFL. Mostriamo induttivamente come convertire una produzione con espressioni regolari come corpo in una serie di produzioni usuali equivalenti. L'induzione si compie sulla dimensione dell'espressione nel corpo.

BASE Se il corpo è una concatenazione di elementi, la produzione è già in una forma lecita per le CFG: non occorre mutarla.

INDUZIONE Altrimenti ci sono cinque casi, a seconda dell'ultimo operatore.

1. La produzione ha forma $A \rightarrow E_1, E_2$, dove E_1 ed E_2 sono espressioni ammesse nel linguaggio delle DTD. Questo è il caso della concatenazione. Si introducono due nuove variabili, B e C, che non compaiono altrove nella grammatica. Si sostituisce $A \rightarrow E_1, E_2$ con le produzioni

$$A \rightarrow BC$$
$$B \rightarrow E_1$$
$$C \rightarrow E_2$$

   La prima, $A \rightarrow BC$, è lecita nelle CFG. Le altre due possono non esserlo, ma i loro corpi sono più corti del corpo della produzione originaria; possiamo quindi convertirli induttivamente nella forma delle CFG.

2. La produzione è della forma $A \rightarrow E_1 \mid E_2$. Per l'operatore di unione si sostituisce la produzione con altre due:

$$A \rightarrow E_1$$
$$A \rightarrow E_2$$

   Anche qui le due nuove produzioni possono non essere lecite, ma i loro corpi sono più brevi dell'originale. Possiamo quindi applicare ricorsivamente le regole fino a trasformarle in produzioni nella forma richiesta dalle CFG.

3. La produzione è della forma $A \rightarrow (E_1)^*$. Introduciamo una nuova variabile, B, che non compare altrove, e sostituiamo la produzione con

$$A \to BA$$
$$A \to \epsilon$$
$$B \to E_1$$

4. La produzione è della forma $A \to (E_1)^+$. Introduciamo una nuova variabile, B, che non compare altrove, e sostituiamo la produzione con

$$A \to BA$$
$$A \to B$$
$$B \to E_1$$

5. La produzione è della forma $A \to (E_1)?$. La sostituiamo con

$$A \to \epsilon$$
$$A \to E_1$$

Esempio 5.24 Consideriamo il problema di convertire la regola per DTD

```
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

in un insieme di produzioni lecite per le CFG. Possiamo trattare il corpo della regola come la concatenazione di due espressioni, rispettivamente `MODEL, PRICE, PROCESSOR, RAM` e `DISK+`. Associamo due variabili, A e B, alle due sottoespressioni, e definiamo le produzioni

$$Pc \to AB$$
$$A \to Model\ Price\ Processor\ Ram$$
$$B \to Disk^+$$

Solo l'ultima non è in forma accettabile. Introduciamo un'altra variabile, C, e le produzioni

$$B \to CB \mid C$$
$$C \to Disk$$

In questo caso particolare, poiché l'espressione derivata da A è una semplice concatenazione di variabili e $Disk$ è una variabile, A e C non sono in realtà necessarie. Possiamo invece definire le seguenti produzioni:

$$Pc \to Model\ Price\ Processor\ Ram\ B$$
$$B \to Disk\ B \mid Disk$$

□

5.3.5 Esercizi

Esercizio 5.3.1 Dimostrate che una stringa di parentesi bilanciate, nel senso dell'Esempio 5.19, è generata dalla grammatica $B \rightarrow BB \mid (B) \mid \epsilon$. *Suggerimento*: procedete per induzione sulla lunghezza della stringa.

* **Esercizio 5.3.2** Consideriamo l'insieme di tutte le stringhe di parentesi bilanciate di due tipi, tonde e quadre. Possiamo trovare stringhe di questo tipo prendendo le espressioni del C (in cui le parentesi tonde si adoperano per delimitare gruppi e per gli argomenti delle chiamate di funzioni, mentre le parentesi quadre segnalano gli indici degli array) e cancellando tutti i simboli tranne le parentesi. Per esempio

```
f(a[i]*(b[i][j],c[g(x)]),d[i])
```

diventa la stringa di parentesi bilanciate `([]([][][()])[])`. Ideate una grammatica per l'insieme di tutte e sole le stringhe di parentesi, tonde e quadre, bilanciate.

! **Esercizio 5.3.3** Nel Paragrafo 5.3.1 abbiamo esaminato la grammatica

$$S \rightarrow \epsilon \mid SS \mid iS \mid iSeS$$

e affermato che possiamo verificare l'appartenenza al suo linguaggio L ripetendo le operazioni elencate qui sotto, a partire da una stringa w, che cambia a ogni ripetizione.

1. Se la stringa in esame comincia per e, w non appartiene a L.
2. Se la stringa non contiene e (può contenere degli i), w appartiene a L.
3. Altrimenti cancella il primo e e l'i alla sua immediata sinistra. Ripeti la procedura con la nuova stringa.

Dimostrate che la procedura stabilisce correttamente quali stringhe appartengono a L.

Esercizio 5.3.4 Aggiungete le forme seguenti alla grammatica per HTML della Figura 5.13.

* a) Una voce di elenco dev'essere terminata dal tag di chiusura `</LI>`.

b) Un elemento può consistere in un elenco non ordinato, oltre che ordinato. Gli elenchi non ordinati sono racchiusi fra i tag `<UL>` e `</UL>`.

! c) Un elemento può consistere in una tabella. Le tabelle sono racchiuse fra `<TABLE>` e il `</TABLE>` più vicino. Fra i due tag si trovano una o più righe, ciascuna racchiusa fra `<TR>` e `</TR>`. La prima riga fa da intestazione con uno o più campi, ciascuno introdotto dal tag `<TH>` (assumiamo che non ci sia il tag di chiusura, anche se dovrebbe esserci). I campi delle righe successive sono introdotti dal tag `<TD>`.

Esercizio 5.3.5 Convertite la DTD della Figura 5.16 in una grammatica libera dal contesto.

```
<!DOCTYPE CourseSpecs [
    <!ELEMENT COURSES (COURSE+)>
    <!ELEMENT COURSE (CNAME, PROF, STUDENT*, TA?)>
    <!ELEMENT CNAME (#PCDATA)>
    <!ELEMENT PROF (#PCDATA)>
    <!ELEMENT STUDENT (#PCDATA)>
    <!ELEMENT TA (#PCDATA)>
]>
```

Figura 5.16 Una DTD per corsi universitari.

5.4 Ambiguità nelle grammatiche e nei linguaggi

Come abbiamo visto, spesso le applicazioni delle CFG si fondano sulla capacità di una grammatica di associare una struttura a un file. Nel Paragrafo 5.3, per esempio, abbiamo studiato come usare le grammatiche per imporre una struttura a programmi e documenti. Implicitamente abbiamo ipotizzato che una grammatica associ in modo univoco una struttura a ogni stringa del suo linguaggio. Vedremo però che non tutte le grammatiche generano strutture univoche.

Se una grammatica non genera strutture univoche, talvolta possiamo modificarla per raggiungere lo scopo. Purtroppo in certi casi l'operazione non è possibile. Ci sono infatti dei CFL "inerentemente ambigui"; ogni grammatica di un tale linguaggio associa più di una struttura ad alcune stringhe del linguaggio.

5.4.1 Grammatiche ambigue

Torniamo all'esempio corrente: la grammatica delle espressioni nella Figura 5.2. Questa grammatica genera espressioni con qualsiasi sequenza degli operatori $*$ e $+$; le produzioni $E \to E + E \mid E * E$ permettono di generarle in qualsiasi ordine.

Esempio 5.25 Consideriamo per esempio la forma sentenziale $E + E * E$, che ha due derivazioni da E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$

Notiamo che nella derivazione (1) la seconda E è sostituita da $E * E$, mentre nella derivazione (2) la prima E è sostituita da $E + E$. La Figura 5.17 presenta i due alberi sintattici, diversi tra loro.

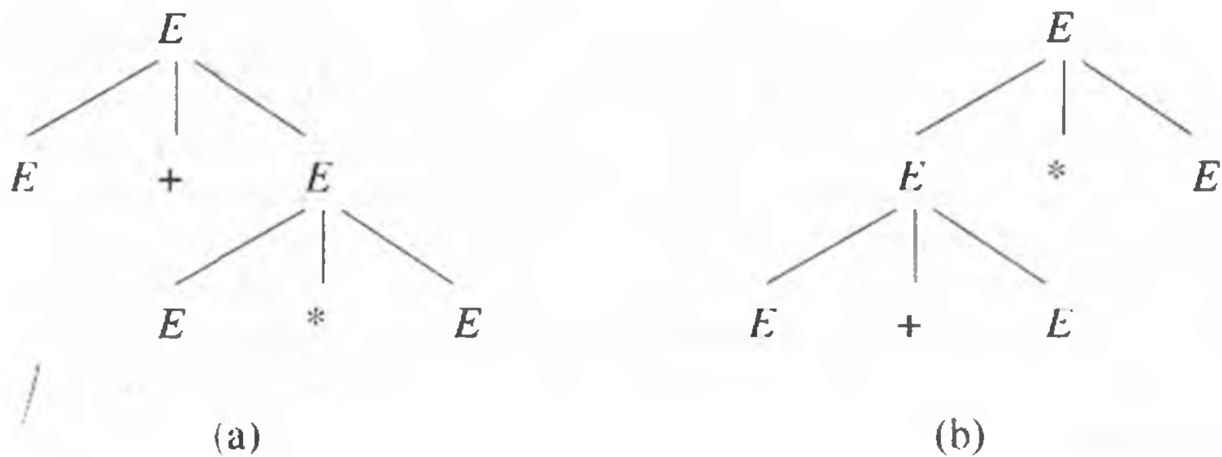


Figura 5.17 Due alberi sintattici con lo stesso prodotto.

La differenza fra le due derivazioni è rilevante. Rispetto alla struttura delle espressioni, la derivazione (1) dice che la seconda e la terza espressione vanno moltiplicate, e il risultato va sommato alla prima; la derivazione (2), invece, somma le due prime espressioni e moltiplica il risultato per la terza. Più concretamente la prima derivazione suggerisce di raggruppare $1 + 2 * 3$ in questo modo: $1 + (2 * 3) = 7$, mentre la seconda suggerisce il raggruppamento $(1 + 2) * 3 = 9$. Solo il primo modo corrisponde alla nozione usuale di raggruppamento delle espressioni aritmetiche.

La grammatica della Figura 5.2 dà due strutture diverse a una stringa di terminali derivata sostituendo le tre espressioni in $E + E * E$ con identificatori; dunque questa grammatica non genera strutture univoche. In particolare, pur associando il raggruppamento corretto a un'espressione aritmetica, essa ne associa anche uno scorretto. Per poter impiegare questa grammatica in un compilatore dovremmo modificarla in modo che generi solo il raggruppamento corretto. □

L'esistenza di derivazioni distinte per la stessa stringa (e non di alberi sintattici distinti) non comporta di per sé un difetto nella grammatica. Il prossimo esempio illustra questo punto.

Esempio 5.26 Nella stessa grammatica delle espressioni, la stringa $a + b$ ammette molte derivazioni distinte. Ecco due esempi:

1. $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$

2. $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

Per quanto diverse, le due derivazioni non danno luogo a strutture realmente differenti; ciascuna dice che a e b sono identificatori e che se ne devono sommare i valori. Di fatto, quando si applica la costruzione dei Teoremi 5.18 e 5.12, entrambe producono lo stesso albero. □

I due esempi precedenti suggeriscono che l'ambiguità non risulta da derivazioni multiple, ma dall'esistenza di due o più alberi sintattici. Perciò diciamo che una CFG $G = (V, T, P, S)$ è *ambigua* se esiste almeno una stringa w in T^* per la quale possiamo trovare due alberi sintattici distinti, ciascuno con la radice etichettata S e con prodotto w. Se ogni stringa ha al massimo un albero sintattico, la grammatica è *non ambigua*.

Nell'Esempio 5.25 abbiamo avviato una dimostrazione dell'ambiguità della grammatica della Figura 5.2. Resta solo da provare che gli alberi nella Figura 5.17 possono essere completati fino a prodotti costituiti da soli terminali. La Figura 5.18 è un esempio del completamento.

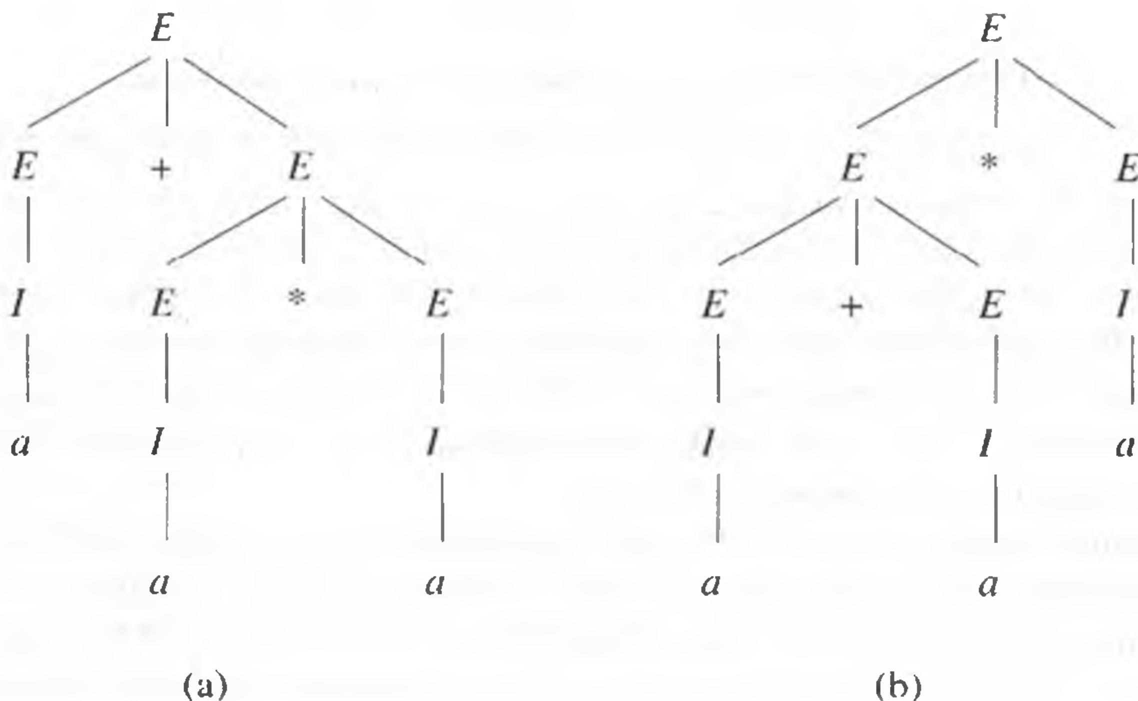


Figura 5.18 Alberi con prodotto $a + a * a$, che dimostrano l'ambiguità della grammatica delle espressioni.

5.4.2 Eliminare le ambiguità da una grammatica

In un mondo ideale sapremmo definire un algoritmo per eliminare le ambiguità dalle CFG come abbiamo definito, nel Paragrafo 4.4, un algoritmo per eliminare gli stati inutili di un automa finito. Sorprendentemente, come mostreremo nel Paragrafo 9.5.2, non esiste neppure un algoritmo che dica se una CFG è ambigua. Non solo: vedremo nel Paragrafo 5.4.4 che ci sono linguaggi liberi dal contesto che hanno solo CFG ambigue; per questi linguaggi eliminare l'ambiguità è impossibile.

Per fortuna la situazione non è nella pratica così critica. Per le strutture sintattiche tipiche dei linguaggi di programmazione usuali, esistono tecniche ben note per eliminare

l'ambiguità. Il problema della grammatica delle espressioni (Figura 5.2) è esemplare. A titolo di esempio esamineremo come eliminarne l'ambiguità.

Notiamo anzitutto che nella grammatica in esame si hanno due cause di ambiguità.

1. La precedenza degli operatori non è rispettata. Mentre la Figura 5.17(a) raggruppa correttamente l'operatore $*$ prima di $+$, la Figura 5.17(b) forma un albero sintattico altrettanto valido, ma raggruppa $+$ prima di $*$. Dobbiamo fare in modo che in una grammatica non ambigua sia lecita solo la struttura di (a).

2. Una sequenza di operatori identici si può raggruppare sia da sinistra sia da destra. Se per esempio gli $*$ nella Figura 5.17 fossero sostituiti da $+$, vedremmo due diversi alberi per la stringa $E + E + E$. Poiché la somma e la moltiplicazione sono associative, non importa se raggruppiamo da sinistra o da destra, ma per eliminare l'ambiguità dobbiamo fare una scelta. La soluzione convenzionale prevede di raggruppare da sinistra; l'unico raggruppamento corretto dei due segni $+$ è quindi la struttura della Figura 5.17(b).

Il problema di imporre una precedenza si risolve introducendo alcune variabili, ognuna delle quali rappresenta le espressioni con lo stesso grado di "forza di legamento", secondo lo schema seguente.

1. Un *fattore* è un'espressione che non si può scomporre rispetto a un operatore adiacente, $*$ o $+$. Nel linguaggio delle espressioni i soli fattori sono i seguenti.

   (a) Identificatori. Non è possibile separare le lettere di un identificatore inserendo un operatore.

   (b) Qualsiasi espressione fra parentesi, indipendentemente dal suo contenuto. Lo scopo delle parentesi è proprio quello di impedire che ciò che racchiudono diventi l'operando di un operatore esterno.

2. Un *termine* è un'espressione che non può essere scomposta dall'operatore $+$. Nell'esempio, in cui $+$ e $*$ sono i soli operatori, un termine è il prodotto di uno o più fattori. Per esempio il termine $a * b$ può essere scomposto se applichiamo l'associatività a sinistra e poniamo $a1*$ alla sua sinistra. Infatti $a1 * a * b$ è raggruppato come $(a1 * a) * b$, che separa $a * b$. Se invece poniamo un termine additivo come $a1+$ a sinistra o $+a1$ a destra, non possiamo spezzarlo. Il raggruppamento corretto di $a1 + a * b$ è $a1 + (a * b)$, mentre quello di $a * b + a1$ è $(a * b) + a1$.

3. Per *espressione* intenderemo d'ora in avanti qualsiasi espressione, comprese quelle che possono essere spezzate da un $*$ o da un $+$ adiacente. Nell'esempio, perciò, un'espressione è la somma di uno o più termini.

Esempio 5.27 La Figura 5.19 presenta una grammatica non ambigua che genera lo stesso linguaggio di quella della Figura 5.2. I linguaggi delle variabili F, T ed E sono i fattori, i termini e le espressioni, come definiti più sopra. Questa grammatica ammette un solo albero sintattico per la stringa $a + a * a$; lo vediamo nella Figura 5.20.

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

Figura 5.19 Una grammatica delle espressioni non ambigua.

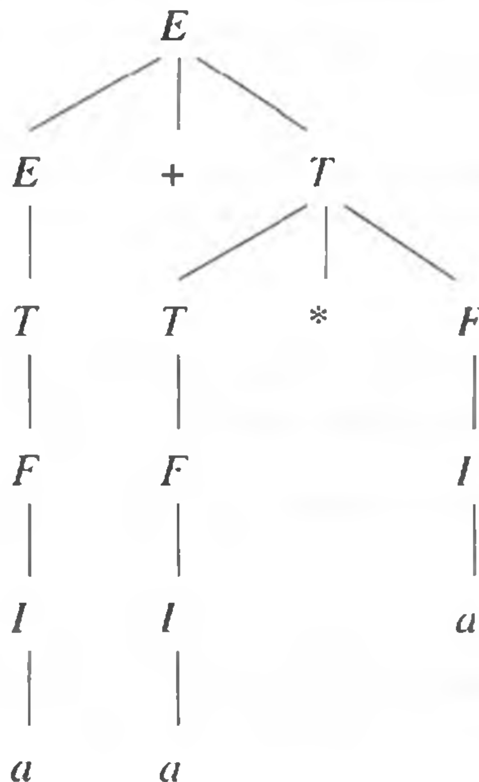


Figura 5.20 L'unico albero sintattico di $a + a * a$.

Non è affatto evidente che questa grammatica è non ambigua. Alcune osservazioni spiegano perché nessuna stringa nel linguaggio può avere due alberi sintattici distinti.

- Qualsiasi stringa derivata da T, cioè un termine, dev'essere una sequenza di uno o più fattori legati da $*$. Un fattore, secondo la definizione e secondo le produzioni per F nella Figura 5.19, è un singolo identificatore o un'espressione qualsiasi fra parentesi.

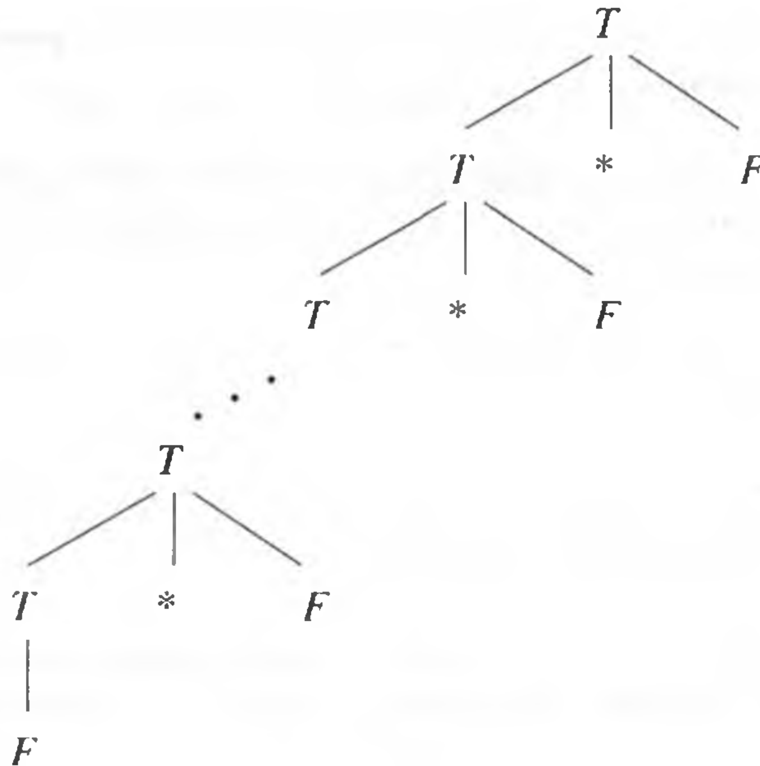


Figura 5.21 La forma di tutti gli alberi sintattici di un termine.

- Per come sono fatte le produzioni di T, l'unico albero sintattico di una sequenza di fattori è quello che spezza $f_1 * f_2 * \cdots * f_n$, con $n > 1$, nel termine $f_1 * f_2 * \cdots * f_{n-1}$ e nel fattore f_n. Infatti F non può generare espressioni come $f_{n-1} * f_n$ senza introdurre parentesi che le racchiudano. Non è possibile perciò che applicando la produzione $T \to T * F$, da F derivi un'espressione che non sia l'ultimo fattore. In altre parole l'albero sintattico di un termine può avere solo la forma della Figura 5.21.

- Allo stesso modo un'espressione è una sequenza di termini legati da $+$. Quando applichiamo la produzione $E \to E + T$ per derivare $t_1 + t_2 + \cdots + t_n$, da T deve derivare soltanto t_n, mentre dalla E nel corpo deriva $t_1 + t_2 + \cdots + t_{n-1}$. Il motivo è di nuovo che da T non può derivare la somma di due o più termini senza parentesi che li racchiudano.

□

5.4.3 Derivazioni a sinistra come modo per esprimere l'ambiguità

Le derivazioni non sono necessariamente uniche, anche in grammatiche non ambigue, ma in una grammatica non ambigua le derivazioni a sinistra sono uniche, così come le

derivazioni a destra. Tratteremo solo le derivazioni a sinistra, limitandoci per quelle a destra a enunciare il risultato.

Esempio 5.28 Come esempio consideriamo i due alberi sintattici della Figura 5.18, che hanno entrambi il prodotto $E + E * E$. Le derivazioni a sinistra corrispondenti agli alberi (a) e (b) sono le seguenti.

a) $E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} I + E \underset{lm}{\Rightarrow} a + E \underset{lm}{\Rightarrow} a + E * E \underset{lm}{\Rightarrow} a + I * E \underset{lm}{\Rightarrow} a + a * E \underset{lm}{\Rightarrow}$
$a + a * I \underset{lm}{\Rightarrow} a + a * a$

b) $E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} E + E * E \underset{lm}{\Rightarrow} I + E * E \underset{lm}{\Rightarrow} a + E * E \underset{lm}{\Rightarrow} a + I * E \underset{lm}{\Rightarrow}$
$a + a * E \underset{lm}{\Rightarrow} a + a * I \underset{lm}{\Rightarrow} a + a * a$

Notiamo che le due derivazioni sono diverse. Questo esempio non dimostra il teorema, ma chiarisce come le differenze negli alberi impongano di seguire passi diversi in una derivazione a sinistra. □

Teorema 5.29 Per ogni grammatica $G = (V, T, P, S)$ e per ogni stringa w in T^*, w ha due alberi sintattici distinti se e solo se ha due distinte derivazioni a sinistra da S.

DIMOSTRAZIONE (Solo se) Esaminando la costruzione di una derivazione a sinistra a partire da un albero sintattico nella dimostrazione del Teorema 5.14, osserviamo che, in corrispondenza del primo nodo in cui gli alberi applicano produzioni diverse, anche le due derivazioni a sinistra applicano produzioni diverse, e sono perciò distinte.

(Se) Non abbiamo ancora trattato la costruzione diretta di un albero sintattico da una derivazione a sinistra, ma lo schema non è difficile. Si comincia da un albero con la sola radice etichettata S. Si esamina la derivazione un passo per volta. A ogni passo si deve sostituire una variabile, corrispondente al nodo più a sinistra nell'albero in costruzione, privo di figli, ma etichettato da una variabile. Dalla produzione usata in questo passo della derivazione si stabilisce quali sono i figli del nodo. Se ci sono due derivazioni distinte, al primo passo in cui si differenziano, i nodi da costruire avranno sequenze diverse di figli; questa differenza garantisce che gli alberi sintattici sono distinti. □

5.4.4 Ambiguità inerente

Un linguaggio L libero dal contesto si dice *inerentemente ambiguo* se tutte le sue grammatiche sono ambigue. Se anche una sola grammatica per L è non ambigua, L è non ambiguo. Abbiamo osservato, per esempio, che il linguaggio delle espressioni generato dalla grammatica della Figura 5.2 è in realtà non ambiguo. Infatti, anche se quella grammatica è ambigua, ce n'è un'altra non ambigua per lo stesso linguaggio: quella della Figura 5.19.

Risoluzione delle ambiguità in YACC

Poiché la grammatica delle espressioni che abbiamo esaminato è ambigua, possiamo chiederci se l'esempio di programma per YACC della Figura 5.11 sia realistico. È vero, la grammatica soggiacente è ambigua, ma l'utilità di YACC come generatore di parser deriva in gran parte dai semplici meccanismi che offre all'utente per risolvere le cause più diffuse di ambiguità. Per la grammatica delle espressioni basta sapere quanto segue.

a) $*$ ha precedenza su $+$. In altre parole $*$ si raggruppa prima di ogni $+$ adiacente, da qualsiasi lato. Questa regola impone di usare la derivazione (1) anziché la (2) nell'Esempio 5.25.

b) Sia $*$ sia $+$ sono associativi a sinistra. In altre parole una sequenza di espressioni, tutte legate da $*$, si raggruppano da sinistra; lo stesso vale per le sequenze legate da $+$.

YACC permette di definire la precedenza degli operatori elencandoli in ordine crescente di priorità. Tecnicamente la precedenza di un operatore si applica in ogni produzione nel cui corpo l'operatore sia l'ultimo terminale a destra. Mediante le parole `%left` e `%right` si può dichiarare che un operatore è associativo a sinistra o a destra. Ad esempio, per dichiarare che $+$ e $*$ sono entrambi associativi a sinistra e che $*$ ha precedenza su $+$, si devono porre i seguenti enunciati prima della grammatica della Figura 5.11.

```
%left '+'
%left '*'
```

Non dimostreremo l'esistenza di linguaggi inerentemente ambigui. Tratteremo invece un esempio di linguaggio di cui si può dimostrare l'ambiguità inerente e spiegheremo in termini intuitivi perché qualsiasi grammatica che lo generi dev'essere ambigua. Il linguaggio L è

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

Il linguaggio L consiste quindi delle stringhe in $\mathbf{a}^+\mathbf{b}^+\mathbf{c}^+\mathbf{d}^+$ tali che vale una delle seguenti condizioni.

1. Ci sono tanti a quanti b e tanti c quanti d.

2. Ci sono tanti a quanti d e tanti b quanti c.

Il linguaggio L è libero dal contesto. La grammatica più evidente per L, presentata nella Figura 5.22, impiega insiemi separati di produzioni per generare i due tipi di stringhe di L.

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

Figura 5.22 Una grammatica per un linguaggio inerentemente ambiguo.

Questa grammatica è ambigua. Per esempio la stringa $aabbccdd$ ha due derivazioni a sinistra:

1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

e i due alberi sintattici illustrati nella Figura 5.23.

La dimostrazione che tutte le grammatiche per L sono ambigue è complessa; possiamo però descriverne la sostanza. Dobbiamo provare che, fatta eccezione per un numero finito di stringhe, tutte le stringhe contenenti lo stesso numero di a, b, c e d possono essere generate in due modi diversi: nel primo caso si fa in modo che il numero di a sia uguale al numero di b e il numero di c uguale al numero di d; nel secondo si generano tanti a quanti d e tanti b quanti c.

Per esempio il solo modo di generare stringhe con lo stesso numero di a e di b si serve di una variabile come A nella grammatica della Figura 5.22. Esistono alcune varianti, naturalmente, ma il quadro di fondo non muta. Vediamo alcuni esempi.

- Si possono evitare alcune stringhe brevi, per esempio modificando la produzione di base $A \rightarrow ab$ in $A \rightarrow aaabbb$.

- Possiamo far sì che A divida il suo compito con altre variabili, per esempio definendo A_1 e A_2, destinate a generare rispettivamente un numero dispari e un numero pari di a: $A_1 \rightarrow aA_2b \mid ab$; $A_2 \rightarrow aA_1b \mid ab$.

- Possiamo anche fare in modo che il numero di a e di b generati da A non sia lo stesso, ma differisca di una quantità finita. Possiamo per esempio partire dalla produzione $S \rightarrow AbB$ e poi servirci di $A \rightarrow aAb \mid a$ per generare un a in più dei b.

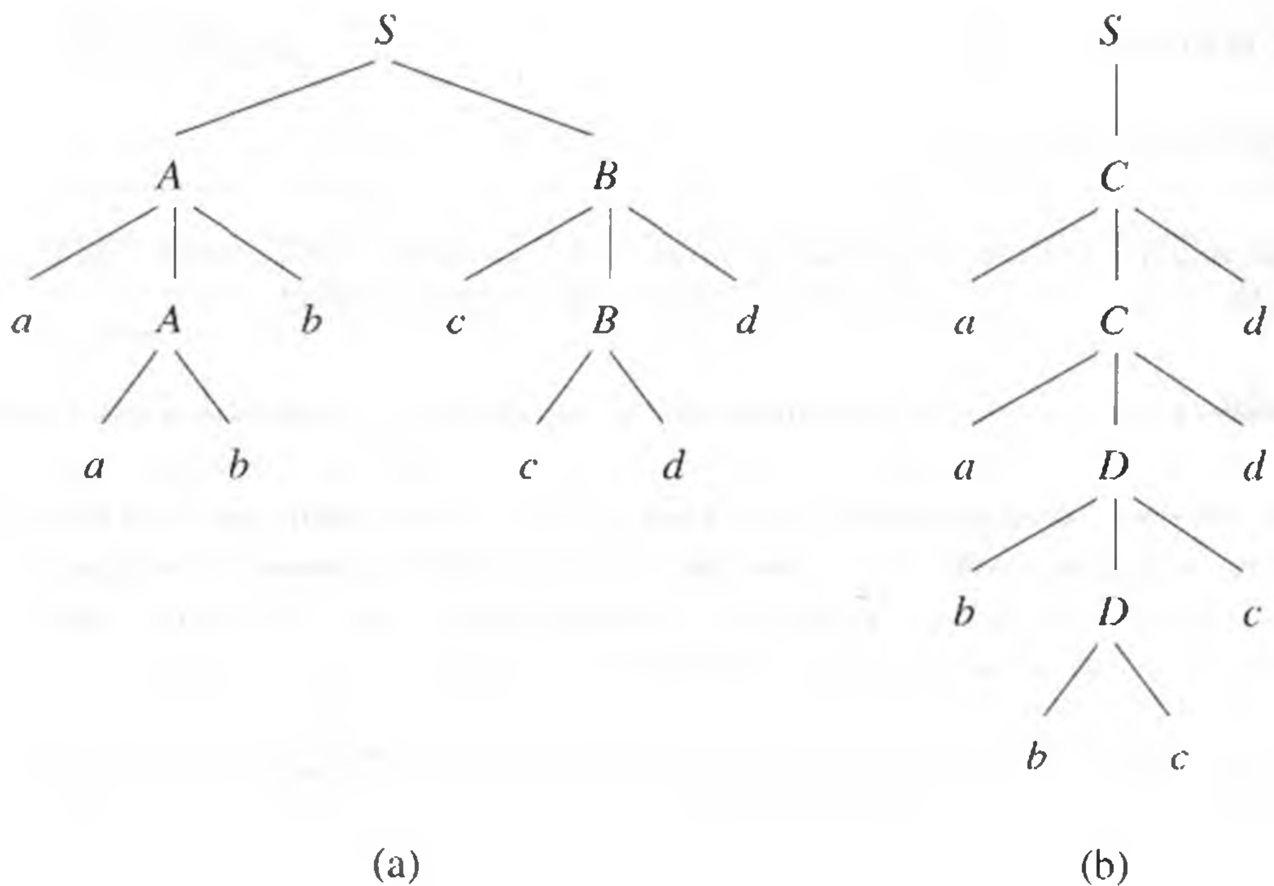


Figura 5.23 Due alberi sintattici per *aabbccdd*.

In ogni caso non possiamo fare a meno di un meccanismo per generare gli a in modo che corrispondano al numero di b.

Analogamente possiamo sostenere che dev'esserci una variabile come B, che genera lo stesso numero di c e di d. La grammatica deve inoltre comprendere variabili che svolgano il compito di C (generare lo stesso numero di a e di d) e quello di D (generare lo stesso numero di b e di c). Una volta formalizzato, questo ragionamento dimostra che, comunque si modifichi la grammatica, essa deve generare almeno alcune delle stringhe della forma $a^n b^n c^n d^n$ nei due modi della grammatica della Figura 5.22.

5.4.5 Esercizi

* **Esercizio 5.4.1** Considerate la grammatica

$$S \rightarrow aS \mid aSbS \mid \epsilon$$

Questa grammatica è ambigua. Mostrate che, in particolare, la stringa aab ha due:

a) alberi sintattici

b) derivazioni a sinistra

c) derivazioni a destra.

! **Esercizio 5.4.2** Dimostrate che la grammatica dell'Esercizio 5.4.1 genera tutte e sole le stringhe di a e b tali che ogni prefisso contiene almeno tanti a quanti b.

*! **Esercizio 5.4.3** Trovate una grammatica non ambigua per il linguaggio dell'Esercizio 5.4.1

!! **Esercizio 5.4.4** Nella grammatica dell'Esercizio 5.4.1 alcune stringhe di a e b hanno un solo albero sintattico. Ideate un modo efficiente per stabilire se una data stringa possiede questa proprietà. Il metodo "prova tutti gli alberi sintattici per vedere quali hanno come prodotto la stringa assegnata" non è abbastanza efficiente.

! **Esercizio 5.4.5** Questo problema riguarda la grammatica dell'Esercizio 5.1.2, che riportiamo qui.

$$\begin{array}{lcl} S & \rightarrow & A1B \\ A & \rightarrow & 0A \mid \epsilon \\ B & \rightarrow & 0B \mid 1B \mid \epsilon \end{array}$$

a) Mostrate che questa grammatica è non ambigua.

b) Trovate una grammatica ambigua per lo stesso linguaggio e dimostratene l'ambiguità.

*! **Esercizio 5.4.6** La grammatica con cui avete risolto l'Esercizio 5.1.5 è non ambigua? Se non lo fosse, rendetela non ambigua.

Esercizio 5.4.7 La grammatica che segue genera espressioni in notazione prefissa con gli operandi x e y e gli operatori binari $+$, $-$, e $*$.

$$E \rightarrow +EE \mid *EE \mid -EE \mid x \mid y$$

a) Trovate una derivazione a sinistra e una a destra, e un albero di derivazione, per la stringa `+*-xyxy`.

! b) Dimostrate che la grammatica è non ambigua.

5.5 Riepilogo

- *Grammatiche libere dal contesto*: una CFG è un modo di descrivere un linguaggio per mezzo di regole ricorsive chiamate produzioni. Una CFG consiste di un insieme di variabili, un insieme di simboli terminali e una variabile iniziale, oltre alle produzioni. Ogni produzione consiste di una variabile di testa e di un corpo, formato da una stringa di zero o più variabili e terminali.

- *Derivazioni e linguaggi*: a partire dal simbolo iniziale deriviamo le stringhe terminali sostituendo ripetutamente una variabile con il corpo di una produzione che ha tale variabile come testa. Il linguaggio della CFG è l'insieme delle stringhe terminali che è possibile derivare in questo modo, ed è detto linguaggio libero dal contesto.

- *Derivazioni a sinistra e a destra*: se in una stringa sostituiamo sempre la variabile più a sinistra (o, rispettivamente, la più a destra), otteniamo una derivazione a sinistra (rispettivamente, a destra). Ogni stringa nel linguaggio di una CFG ha almeno una derivazione a sinistra e almeno una derivazione a destra.

- *Forme sentenziali*: ogni passo di una derivazione è una stringa di variabili e terminali chiamata forma sentenziale. Se la derivazione è a sinistra (a destra), allora la stringa è una forma sentenziale sinistra (destra).

- *Alberi sintattici*: un albero sintattico è un albero che mostra gli elementi essenziali di una derivazione. I nodi interni sono etichettati da variabili e le foglie sono etichettate da terminali o da ϵ. Per ogni nodo interno deve esistere una produzione tale che la testa della produzione è l'etichetta del nodo, e le etichette dei suoi nodi figli, lette da sinistra a destra, formano il corpo della produzione.

- *Equivalenza di alberi sintattici e derivazioni*: una stringa terminale è nel linguaggio di una grammatica se e solo se è il prodotto di almeno un albero sintattico. Perciò l'esistenza di derivazioni a sinistra, derivazioni a destra e alberi sintattici sono condizioni equivalenti, ognuna delle quali definisce esattamente le stringhe nel linguaggio di una CFG.

- *Grammatiche ambigue*: per determinate CFG è possibile trovare una stringa terminale con più di un albero sintattico o, il che è equivalente, più di una derivazione a sinistra o più di una derivazione a destra. Una grammatica di questo tipo è detta ambigua.

- *Eliminazione dell'ambiguità*: per molte grammatiche utili, come quelle che descrivono la struttura dei programmi in un tipico linguaggio di programmazione,

è possibile trovare una grammatica non ambigua che genera lo stesso linguaggio. Purtroppo la grammatica non ambigua è spesso più complessa della più semplice grammatica ambigua per il linguaggio. Esistono anche alcuni linguaggi liberi dal contesto, di solito piuttosto artificiali, che sono inerentemente ambigui, vale a dire che ammettono solo grammatiche ambigue.

- *Parser*: il concetto di grammatica libera dal contesto è essenziale per realizzare compilatori e altri strumenti per linguaggi di programmazione. Strumenti come YACC prendono una CFG come input e producono un parser, il componente di un compilatore che estrae la struttura del programma in compilazione.

- *Document Type Definition*: lo standard XML per la condivisione di informazioni mediante documenti Web ha una notazione, detta DTD, per descrivere la struttura dei documenti attraverso l'inserimento di tag semantici nel documento. Una DTD è essenzialmente una grammatica libera dal contesto il cui linguaggio è una classe di documenti correlati.

5.6 Bibliografia

Il primo a proporre le grammatiche libere dal contesto come metodo descrittivo per i linguaggi naturali fu Chomsky [4]. Poco dopo un'idea simile venne usata per descrivere linguaggi di programmazione: il Fortran da Backus [2] e l'Algol da Naur [7]. A volte perciò la forma di una CFG è detta "di Backus-Naur".

L'ambiguità nelle grammatiche fu individuata come problema quasi contemporaneamente da Cantor [3] e Floyd [5]. Il primo a trattare l'ambiguità inerente fu Gross [6].

Per le applicazioni delle CFG nei compilatori si veda [1]. Le DTD sono definite nei documenti standard per XML [8].

1. A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

2. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proc. Intl. Conf. on Information Processing* (1959), UNESCO, pp. 125–132.

3. D. C. Cantor, "On the ambiguity problem of Backus systems," *J. ACM* **9**:4 (1962), pp. 477–479.

4. N. Chomsky, "Three models for the description of language," *IRE Trans. on Information Theory* **2**:3 (1956), pp. 113–124.

5. R. W. Floyd, "On ambiguity in phrase-structure languages," *Comm. ACM* **5**:10 (1962), pp. 526–534.

6. M. Gross, "Inherent ambiguity of minimal linear grammars," *Information and Control* **7**:3 (1964), pp. 366–368.

7. P. Naur et al., "Report on the algorithmic language ALGOL 60," *Comm. ACM* **3**:5 (1960), pp. 299–314. Vedi anche *Comm. ACM* **6**:1 (1963), pp. 1–17.

8. World-Wide-Web Consortium, `http://www.w3.org/TR/REC-xml` (1998).

Capitolo 6

Automi a pila

I linguaggi liberi dal contesto sono definiti da automi del tipo detto "a pila" (*pushdown automaton*), un'estensione degli automi a stati finiti non deterministici con ϵ-transizioni, che è uno dei modi di definire i linguaggi regolari. Un automa a pila è essenzialmente un ϵ-NFA con l'aggiunta di una "pila" (*stack*[1]). Le operazioni possibili sullo stack: lettura, inserimento ed eliminazione, avvengono solo alla sommità, come nella struttura dati omonima.

In questo capitolo definiamo due versioni diverse di automa a pila: la prima accetta entrando in uno stato accettante, come fanno gli automi a stati finiti; la seconda accetta vuotando lo stack, a prescindere dallo stato in cui si trova. Mostreremo che queste due varianti accettano proprio i linguaggi liberi dal contesto; in altre parole le grammatiche possono essere convertite in automi a pila equivalenti, e viceversa. Inoltre considereremo brevemente la sottoclasse degli automi a pila deterministici, che accettano tutti i linguaggi regolari, ma solo un sottoinsieme proprio dei CFL. Poiché gli automi a pila deterministici operano in modo simile ai *parser* nei compilatori, è importante osservare quali costrutti linguistici sono in grado di riconoscere e quali no.

6.1 Definizione di automa a pila

In questo paragrafo presentiamo gli automi a pila, dapprima in termini informali, poi formalmente.

---

[1] Poiché è entrato nell'uso comune, abbiamo preferito impiegare in questo contesto il termine "stack" anziché "pila".

6.1.1 Introduzione informale

Un automa a pila è in sostanza un automa a stati finiti non deterministico con ϵ-transizioni e una capacità aggiuntiva: uno stack in cui può memorizzare una stringa di simboli. A differenza dell'automa a stati finiti, grazie allo stack l'automa a pila può "ricordare" una quantità illimitata di informazioni. Tuttavia, diversamente da un computer generico, che a sua volta può ricordare quantità di informazioni arbitrariamente grandi, l'automa a pila può accedere alle informazioni nello stack solo in modo "last-in-first-out" (ultimo arrivato, primo servito).

Ne risulta che esistono linguaggi che potrebbero essere riconosciuti da un programma, ma non lo sono da nessun automa a pila. In effetti gli automi a pila riconoscono tutti e soli i linguaggi liberi dal contesto. Molti linguaggi *sono* liberi dal contesto, inclusi alcuni linguaggi che non sono regolari, come abbiamo visto, ma esistono anche linguaggi di facile descrizione che non sono liberi dal contesto, come vedremo nel Paragrafo 7.2. Un esempio di linguaggio non libero dal contesto è $\{0^n1^n2^n \mid n \geq 1\}$, l'insieme delle stringhe che contengono gruppi uguali di 0, 1 e 2.

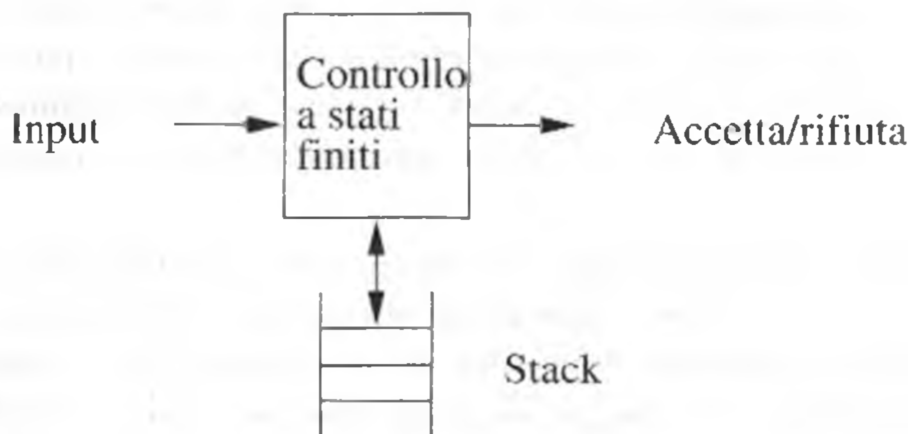


Figura 6.1 Un automa a pila è in sostanza un automa a stati finiti con una struttura dati a stack.

Possiamo immaginare informalmente un automa a pila come il dispositivo illustrato nella Figura 6.1. Un "controllo a stati finiti" legge l'input, un simbolo per volta. L'automa a pila può osservare il simbolo alla sommità dello stack e può basare la transizione sullo stato corrente, sul simbolo di input e sul simbolo alla sommità dello stack. In alternativa può fare una transizione "spontanea", usando ϵ come input in luogo di un simbolo effettivo. In una transizione l'automa compie tre operazioni.

1. Consuma dall'input il simbolo che usa nella transizione. Nel caso speciale di ϵ non si consuma alcun simbolo di input.

2. Passa a un nuovo stato, che può essere o no uguale al precedente.

3. Sostituisce il simbolo in cima allo stack con una stringa. La stringa può essere ϵ, che corrisponde a togliere dallo stack. Può anche essere lo stesso simbolo che c'era prima alla sommità dello stack; in questo caso lo stack non subisce nessun cambiamento effettivo. Oppure può essere un altro simbolo, con l'effetto di trasformare la sommità dello stack senza inserire o togliere. Infine il simbolo in cima allo stack può essere sostituito da due o più simboli, con l'effetto di cambiare (eventualmente) il simbolo alla sommità, e di inserire poi uno o più simboli nuovi.

Esempio 6.1 Consideriamo il linguaggio

$$L_{wwr} = \{ww^R \mid w \text{ è in } (\mathbf{0} + \mathbf{1})^*\}$$

Si tratta del linguaggio detto "w-w rovesciato", costituito dalle palindrome di lunghezza pari sull'alfabeto $\{0, 1\}$. È un CFL generato dalla grammatica della Figura 5.1 omettendo le produzioni $P \to 0$ e $P \to 1$. Descriviamo un automa a pila che accetta L_{wwr}.[2]

1. Partiamo da uno stato q_0 che rappresenta la "congettura" di non avere ancora esaurito la stringa w; supponiamo cioè di non aver visto la fine della stringa che deve essere seguita dal suo inverso. Finché ci troviamo nello stato q_0 leggiamo i simboli e li accumuliamo nello stack uno per volta.

2. In qualunque momento possiamo "scommettere" di aver visto la prima metà, cioè la fine di w. A questo punto w si troverà nello stack, con il suo estremo destro alla sommità e il suo estremo sinistro in fondo. Indichiamo questa scelta portandoci spontaneamente nello stato q_1. Dato che si tratta di un automa non deterministico, facciamo in effetti due congetture: supponiamo di aver visto la fine di w, ma rimaniamo anche nello stato q_0 e continuiamo a leggere i simboli di input e a memorizzarli nello stack.

3. Una volta che ci troviamo nello stato q_1, confrontiamo il successivo simbolo di input con il simbolo alla sommità dello stack. Se corrispondono, consumiamo il simbolo di input, eliminiamo l'elemento in cima allo stack e procediamo. Se invece non corrispondono, abbiamo formulato una congettura erronea, dato che w non è seguita da w^R, come avevamo ipotizzato. Questo ramo quindi muore, sebbene altri rami dell'automa non deterministico possano sopravvivere e alla fine condurre all'accettazione.

4. Se alla fine lo stack è vuoto, abbiamo effettivamente letto w seguito da w^R. Accettiamo dunque l'input letto fino a questo punto.

□

[2] Potremmo definire un automa a pila anche per L_{pal}, che è il linguaggio di cui abbiamo visto la grammatica nella Figura 5.1. D'altronde L_{wwr}, essendo un po' più semplice, ci permette di focalizzare meglio i concetti di fondo che riguardano gli automi a pila.

6.1.2 Definizione formale di automa a pila

La notazione formale per un *automa a pila* (*PDA, Pushdown Automaton*) include sette componenti:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Spieghiamo il loro significato.

Q: un insieme finito di *stati*, analoghi a quelli di un automa a stati finiti.

Σ: un insieme finito di *simboli di input*, anche questi analoghi al componente corrispondente di un automa a stati finiti.

Γ: un *alfabeto di stack*, finito. Questo componente, che non ha analoghi negli automi a stati finiti, è l'insieme dei simboli che possono essere inseriti nello stack.

δ: la *funzione di transizione*. Come per un automa a stati finiti, δ governa il comportamento dell'automa. In termini formali δ prende come argomento una tripla (q, a, X) in cui:

1. q è uno stato in Q
2. a è un simbolo di input in Σ oppure $a = \epsilon$, la stringa vuota, che assumiamo non sia un simbolo di input
3. X è un simbolo di stack, ossia un membro di Γ.

L'output di δ è un insieme finito di coppie (p, γ), dove p è il nuovo stato e γ è la stringa di simboli di stack che rimpiazza X alla sommità dello stack. Per esempio, se $\gamma = \epsilon$, allora il simbolo in cima allo stack viene eliminato; se $\gamma = X$ lo stack rimane immutato; se $\gamma = YZ$, allora X è sostituito da Z, e Y viene inserito nello stack.

q_0: lo *stato iniziale*. Il PDA inizia a operare da questo stato.

Z_0: il *simbolo iniziale*. Lo stack del PDA consiste all'inizio in una sola copia di questo simbolo e nient'altro.

F: l'insieme degli *stati accettanti*, o *stati finali*.

Esempio 6.2 Definiamo un PDA P che accetti il linguaggio L_{wwr} dell'Esempio 6.1. Dobbiamo chiarire anzitutto alcuni particolari che non sono presenti nell'esempio, ma risultano utili nella gestione dello stack. Useremo il simbolo di stack Z_0 per indicare il fondo dello stack. Questo simbolo fa sì che, dopo aver eliminato w dallo stack e dedotto

Vietato mescolare

In determinate situazioni un PDA può scegliere tra diverse coppie. Per esempio supponiamo che $\delta(q, a, X) = \{(p, YZ), (r, \epsilon)\}$. Nel compiere la mossa dobbiamo scegliere una coppia nella sua integrità: non possiamo prendere uno stato da una coppia e una stringa da un'altra. Di conseguenza nello stato q, con X alla sommità dello stack, leggendo a possiamo andare nello stato p e sostituire X con YZ, oppure possiamo passare allo stato r ed eliminare X. Non possiamo andare nello stato p ed eliminare X, e non possiamo passare allo stato r e sostituire X con YZ.

di aver letto ww^R, ci sia ancora qualcosa nello stack che permette una transizione verso lo stato accettante q_2. Conseguentemente il nostro PDA per L_{wwr} può essere descritto da

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

dove δ è definita dalle seguenti regole.

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ e $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. Una di queste regole viene applicata all'inizio, quando ci troviamo nello stato q_0 e vediamo il simbolo iniziale Z_0 alla sommità dello stack. Leggiamo il primo input e lo inseriamo nello stack, lasciando Z_0 a indicare il fondo.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$ e $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. Queste quattro regole simili permettono di rimanere nello stato q_0 e di leggere i simboli in input, inserendo ognuno di questi alla sommità dello stack e lasciando il simbolo precedente.

3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$ e $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$. Queste tre regole permettono a P di andare spontaneamente dallo stato q_0 allo stato q_1 (su input ϵ) lasciando intatto qualunque simbolo si trovi alla sommità dello stack.

4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$ e $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$. Nello stato q_1 possiamo ora abbinare i simboli di input con i simboli alla sommità dello stack ed eliminarli quando sono uguali.

5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$. Infine, se il segnale di fondo si trova in cima allo stack e siamo nello stato q_1, abbiamo trovato un input della forma ww^R. L'automa passa in q_2 e accetta.

□

6.1.3 Una notazione grafica per i PDA

Come abbiamo visto nell'Esempio 6.2, la definizione di δ non è di immediata comprensione. A volte un diagramma simile al diagramma di transizione di un automa a stati finiti chiarisce il comportamento di un PDA. Presentiamo dunque il *diagramma di transizione* per un PDA, dove valgono le clausole seguenti.

a) I nodi corrispondono agli stati del PDA.

b) Una freccia etichettata *Start* indica lo stato iniziale; come per gli automi a stati finiti, gli stati contrassegnati da un doppio cerchio sono accettanti.

c) Gli archi corrispondono alle transizioni del PDA. In particolare un arco etichettato $a, X/\alpha$ dallo stato q allo stato p significa che $\delta(q, a, X)$ contiene la coppia (p, α), ed eventualmente altre coppie. In altre parole l'etichetta dell'arco indica quale input viene usato, oltre alla sommità dello stack, prima e dopo la transizione.

L'unica informazione che il diagramma non fornisce è il simbolo iniziale dello stack. Per convenzione usiamo Z_0, salvo indicazione contraria.

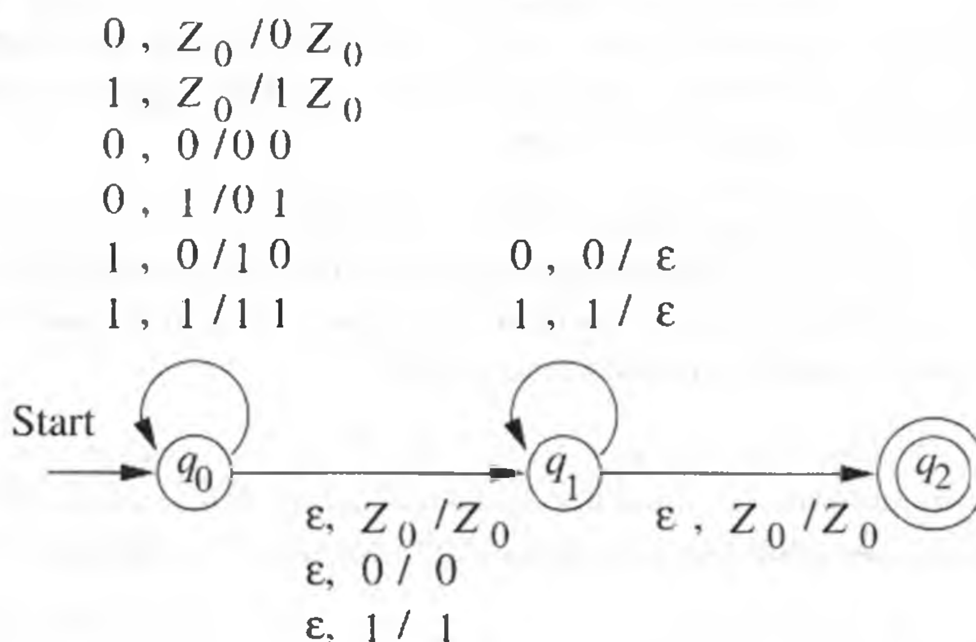


Figura 6.2 Rappresentazione di un PDA come diagramma di transizione generalizzato.

Esempio 6.3 Il PDA dell'Esempio 6.2 è rappresentato dal diagramma della Figura 6.2. □

6.1.4 Descrizioni istantanee di un PDA

Finora abbiamo solo una nozione informale di come un PDA "computa". Intuitivamente il PDA passa da una configurazione all'altra reagendo ai simboli di input (o a ϵ), ma, diversamente dagli automi a stati finiti, dove lo stato è l'unico elemento significativo, la configurazione di un PDA comprende sia lo stato sia il contenuto dello stack. Essendo di grandezza arbitraria, lo stack è spesso la parte più importante della configurazione di un PDA. È utile inoltre rappresentare come parte della configurazione la porzione residua dell'input.

Di conseguenza rappresentiamo una configurazione come una tripla (q, w, γ), dove

1. q è lo stato
2. w è l'input residuo
3. γ è il contenuto dello stack.

Per convenzione mostriamo la sommità dello stack all'estremo sinistro di γ e il fondo all'estremo destro. La tripla è detta *descrizione istantanea* (*ID, Instantaneous Description*) dell'automa a pila.

Per gli automi a stati finiti la notazione $\hat{\delta}$ è sufficiente a rappresentare sequenze di descrizioni istantanee attraverso le quali un automa si muove, perché la ID coincide con lo stato. Per i PDA, invece, occorre una notazione che descriva le trasformazioni di stato, l'input e lo stack. Per connettere le coppie di ID che rappresentano una o più mosse di un PDA adottiamo perciò una notazione particolare.

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Definiamo $\underset{P}{\vdash}$, o $\vdash$ quando P è sottinteso. Supponiamo che $\delta(q, a, X)$ contenga (p, α). Allora, per tutte le stringhe w in Σ^* e β in Γ^*:

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

Questa mossa riflette l'idea che, consumando a (che può essere ϵ) dall'input e sostituendo X alla sommità dello stack con α, si può andare dallo stato q allo stato p. Notiamo che quanto rimane in input, w, e quanto sta sotto la sommità dello stack, β, non influenza l'azione del PDA, ma viene semplicemente "conservato" e ha la possibilità di influenzare il seguito.

Definiamo inoltre il simbolo $\overset{*}{\underset{P}{\vdash}}$, o $\overset{*}{\vdash}$ quando il PDA P è sottinteso, per rappresentare zero o più mosse del PDA.

BASE $I \overset{*}{\vdash} I$ per qualunque ID I.

INDUZIONE $I \overset{*}{\vdash} J$ se esiste una ID K tale che $I \vdash K$ e $K \overset{*}{\vdash} J$.

Vale a dire $I \overset{*}{\vdash} J$ se esiste una sequenza di ID $K_1, K_2, \ldots, K_n$ tale che $I = K_1$, $J = K_n$, e per ogni $i = 1, 2, \ldots, n-1$ abbiamo $K_i \vdash K_{i+1}$.

Esempio 6.4 Consideriamo l'azione del PDA dell'Esempio 6.2 su input 1111. Poiché q_0 è lo stato iniziale e Z_0 è il simbolo iniziale, la ID iniziale è $(q_0, 1111, Z_0)$. Su questo input il PDA ha la possibilità di fare diverse congetture sbagliate. L'intera sequenza di ID che il PDA può raggiungere dalla ID iniziale $(q_0, 1111, Z_0)$ è rappresentata nella Figura 6.3. Le frecce indicano la relazione $\vdash$.

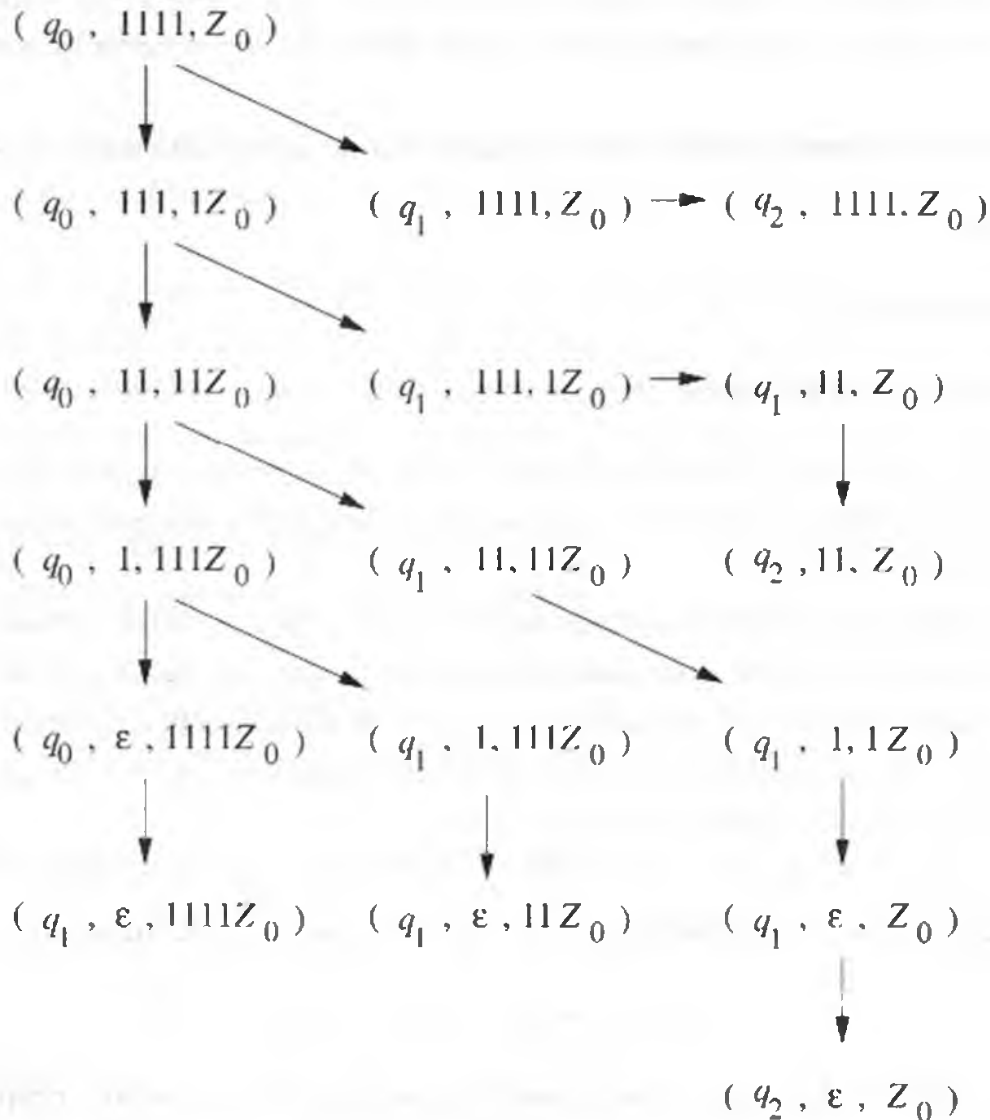


Figura 6.3 Le ID del PDA dell'Esempio 6.2 su input 1111.

A partire dalla ID iniziale ci sono due mosse possibili. La prima ipotizza che la prima metà non sia stata vista e conduce alla ID $(q_0, 111, 1Z_0)$: un 1 è stato rimosso dall'input ed è stato inserito nello stack.

La seconda equivale a supporre che la prima metà sia stata raggiunta. Il PDA va nello stato q_1 portando alla ID $(q_1, 1111, Z_0)$ senza consumo di input. Poiché accetta se si trova nello stato q_1 con Z_0 alla sommità dello stack, il PDA passa alla ID $(q_2, 1111, Z_0)$, che non è accettante perché l'input non è stato interamente consumato. Se l'input fosse stato

Notazione convenzionale per i PDA

Continueremo a usare le convenzioni sull'uso dei simboli introdotte per gli automi a stati finiti e le grammatiche. Nell'adattare la notazione sottolineiamo che i simboli di stack hanno un ruolo analogo all'unione dei terminali e delle variabili di una CFG. Di conseguenza:

1. i simboli dell'alfabeto di input saranno rappresentati dalle lettere iniziali dell'alfabeto, minuscole, come a e b
2. gli stati saranno di solito rappresentati da q e p, o da altre lettere prossime a queste nell'ordine alfabetico
3. le stringhe di simboli di input saranno rappresentate dalle ultime lettere dell'alfabeto, minuscole, per esempio w o z
4. i simboli di stack saranno rappresentati dalle ultime lettere dell'alfabeto, maiuscole, come X o Y
5. le stringhe di simboli di stack saranno rappresentate da lettere greche, come α o γ.

ϵ anziché 1111, la stessa sequenza di mosse avrebbe condotto alla ID (q_2, ϵ, Z_0), da cui sarebbe risultato chiaro che ϵ è accettata.

Il PDA può anche ipotizzare di aver visto la prima metà dopo aver letto un solo 1, cioè quando si trova nella ID $(q_0, 111, 1Z_0)$. Dal momento che non si può consumare tutto l'input, anche questa scelta porta al fallimento. La scelta corretta, cioè aver raggiunto la prima metà dopo aver letto due 1, fornisce la sequenza di ID $(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$. □

Nello studio dei PDA hanno particolare importanza tre principi relativi alle ID e alle transizioni.

1. Se una sequenza di ID (una *computazione*) è lecita per un PDA P, allora è lecita anche la computazione formata accodando una stringa (sempre la stessa) all'input (secondo componente) in ogni ID.
2. Se una computazione è lecita per un PDA P, allora è lecita anche la computazione formata aggiungendo gli stessi simboli sotto quelli nello stack in ogni ID.

3. Se una computazione è lecita per un PDA P, e resta una coda di input non consumata, possiamo rimuovere il residuo dall'input in ogni ID e ottenere una computazione lecita.

In termini intuitivi i dati che P non esamina non possono influenzare la sua computazione. Formalizziamo i punti (1) e (2) in un teorema.

Teorema 6.5 Se $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ è un PDA, e $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$, allora per ogni stringa w in Σ^* e γ in Γ^* è vero anche che

$$(q, xw, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta\gamma)$$

Si noti che se $\gamma = \epsilon$ abbiamo un enunciato formale del principio (1) descritto sopra, e se $w = \epsilon$ abbiamo il secondo principio.

DIMOSTRAZIONE La dimostrazione è un'induzione molto semplice sul numero di passi nella sequenza di ID che portano $(q, xw, \alpha\gamma)$ in $(p, yw, \beta\gamma)$. Ogni mossa nella sequenza $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$ è giustificata dalle transizioni di P senza ricorrere in nessun modo a w e γ. Di conseguenza, quando queste stringhe si trovano in input e sullo stack, ogni mossa è ancora giustificata. □

Notiamo per inciso che l'inverso di questo teorema è falso. Un PDA potrebbe compiere certe operazioni levando un simbolo dallo stack, usando alcuni simboli di γ e poi sostituendoli sullo stack, operazioni impossibili senza esaminare γ. Ma, secondo il principio (3), è possibile rimuovere l'input non utilizzato perché un PDA non può consumare simboli di input e poi ripristinarli. Enunciamo il principio (3) in termini formali.

Teorema 6.6 Se $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ è un PDA e

$$(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$$

allora è vero anche che $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$. □

6.1.5 Esercizi

Esercizio 6.1.1 Supponiamo che il PDA $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$ abbia la seguente funzione di transizione:

1. $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$

2. $\delta(q, 0, X) = \{(q, XX)\}$

ID per automi a stati finiti

Ci si può chiedere perché non abbiamo introdotto per gli automi a stati finiti una notazione come quella delle ID utilizzata per i PDA. Anche se un FA non contiene uno stack, potremmo usare una coppia (q, w), dove q è lo stato e w l'input residuo, come ID di un automa a stati finiti. Anche così, però, non avremmo nessuna ulteriore informazione sulla raggiungibilità tra ID rispetto alla notazione $\hat{\delta}$. Infatti, per ogni automa a stati finiti, si può dimostrare che $\hat{\delta}(q, w) = p$ se e solo se $(q, wx) \overset{*}{\vdash} (p, x)$ per ogni stringa x. L'indipendenza da x del comportamento di un FA è un teorema analogo ai Teoremi 6.5 e 6.6.

3. $\delta(q, 1, X) = \{(q, X)\}$
4. $\delta(q, \epsilon, X) = \{(p, \epsilon)\}$
5. $\delta(p, \epsilon, X) = \{(p, \epsilon)\}$
6. $\delta(p, 1, X) = \{(p, XX)\}$
7. $\delta(p, 1, Z_0) = \{(p, \epsilon)\}$.

A partire dalla ID iniziale (q, w, Z_0), mostrate tutte le ID raggiungibili quando l'input w è:

* a) 01

b) 0011

c) 010.

6.2 I linguaggi di un PDA

Abbiamo stabilito che un PDA accetta una stringa consumandola ed entrando in uno stato accettante. Chiamiamo questa soluzione "accettazione per stato finale". Esiste una seconda via per definire il linguaggio di un PDA, che ha importanti applicazioni. Per un PDA possiamo definire il linguaggio "accettato per stack vuoto", cioè l'insieme delle stringhe che portano il PDA a vuotare lo stack, a partire dalla ID iniziale.

I due metodi sono equivalenti: un linguaggio L ha un PDA che lo accetta per stato finale se e solo se L ha un PDA che lo accetta per stack vuoto. Tuttavia, per un dato PDA P, di solito i linguaggi che P accetta per stato finale e per stack vuoto sono diversi. In

questo paragrafo vedremo come convertire un PDA che accetta L per stato finale in un altro che accetta L per stack vuoto, e viceversa.

6.2.1 Accettazione per stato finale

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Allora $L(P)$, il *linguaggio accettato da P per stato finale*, è

$$\{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \alpha)\}$$

per uno stato q in F e una stringa qualsiasi α. In altre parole, a partire dalla ID iniziale con w in input, P consuma w ed entra in uno stato accettante. A questo punto il contenuto dello stack è irrilevante.

Esempio 6.7 Abbiamo affermato che il PDA dell'Esempio 6.2 accetta il linguaggio L_{wwr}, cioè il linguaggio delle stringhe in $\{0, 1\}^*$ che hanno la forma ww^R. Vediamo ora perché l'enunciato è vero. La dimostrazione è un enunciato "se e solo se": il PDA P dell'Esempio 6.2 accetta la stringa x per stato finale se e solo se x è della forma ww^R.

(Se) Questa parte è facile: occorre solo esibire la computazione accettante di P. Se $x = ww^R$, allora osserviamo che

$$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$$

In altri termini il PDA ha la possibilità di leggere w dall'input e memorizzarlo alla rovescia nello stack. Nel passo successivo il PDA passa spontaneamente allo stato q_1 e fa corrispondere w^R sull'input alla stessa stringa nello stack; infine va spontaneamente nello stato q_2.

(Solo se) Questa parte è più difficile. In primo luogo osserviamo che il solo modo di entrare in uno stato accettante q_2 è trovarsi nello stato q_1 e avere Z_0 alla sommità dello stack. Inoltre qualunque computazione accettante di P parta dallo stato q_0 fa una sola transizione a q_1 senza tornare a q_0. Perciò dobbiamo trovare le condizioni su x per cui $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0)$; si tratta proprio delle stringhe x che P accetta per stato finale. Dimostreremo per induzione su $|x|$ l'enunciato più generale:

- se $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \alpha)$, allora x è della forma ww^R.

BASE Se $x = \epsilon$, allora x è della forma ww^R (con $w = \epsilon$). Perciò la conclusione è vera e dunque l'enunciato è vero. Si noti che non dobbiamo provare che l'ipotesi $(q_0, \epsilon, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \alpha)$ è vera.

INDUZIONE Supponiamo $x = a_1 a_2 \cdots a_n$ per $n > 0$. Dalla ID (q_0, x, α), P può fare due mosse.

1. $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$. Quando si trova nello stato q_1, P può solo togliere simboli dallo stack. A ogni lettura di un simbolo in input lo stack si accorcia: poiché $|x| > 0$, se $(q_1, x, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \beta)$, allora β è più breve di α e non può essere uguale ad α.

2. $(q_0, a_1a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1\alpha)$. Una sequenza di mosse può terminare con (q_1, ϵ, α) solo se l'ultima mossa è un'eliminazione:

$$(q_1, a_n, a_1\alpha) \vdash (q_1, \epsilon, \alpha)$$

In tal caso si ha necessariamente $a_1 = a_n$. Sappiamo inoltre che

$$(q_0, a_2 \cdots a_n, a_1\alpha) \overset{*}{\vdash} (q_1, a_n, a_1\alpha)$$

Per il Teorema 6.6 possiamo eliminare dall'input il simbolo inutilizzato a_n. Quindi

$$(q_0, a_2 \cdots a_{n-1}, a_1\alpha) \overset{*}{\vdash} (q_1, \epsilon, a_1\alpha)$$

Poiché l'input per questa sequenza ha lunghezza minore di n, possiamo applicare l'ipotesi induttiva e concludere che $a_2 \cdots a_{n-1}$ è della forma yy^R per un y opportuno. Poiché $x = a_1yy^Ra_n$, e sappiamo che $a_1 = a_n$, concludiamo che x è della forma ww^R, con $w = a_1y$.

Abbiamo descritto il nucleo della dimostrazione che x è accettata solo se è uguale a ww^R per un certo w. Di conseguenza abbiamo la parte "solo se" della dimostrazione; insieme con la parte "se", già dimostrata, essa afferma che P accetta esattamente le stringhe in L_{wwr}. □

6.2.2 Accettazione per stack vuoto

Per ogni PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ definiamo

$$N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$$

per qualsiasi stato q. Quindi $N(P)$ è l'insieme degli input w che P può consumare, svuotando nel contempo lo stack.[3]

Esempio 6.8 Il PDA P dell'Esempio 6.2 non vuota mai il suo stack, dunque $N(P) = \emptyset$. Ma con una piccola modifica P può accettare L_{wwr} sia per stack vuoto sia per stato finale. Invece della transizione $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$ definiamo $\delta(q_1, \epsilon, Z_0) = \{(q_2, \epsilon)\}$. Ora, quando P accetta, elimina dallo stack l'ultimo simbolo, e $L(P) = N(P) = L_{wwr}$. □

Poiché l'insieme degli stati accettanti è irrilevante, se ci interessa solo il linguaggio che P accetta per stack vuoto, a volte ometteremo l'ultimo (il settimo) componente della specificazione di un PDA P. Scriviamo allora P come una sestupla $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

[3] N in $N(P)$ sta per "stack nullo", sinonimo di "stack vuoto".

6.2.3 Da stack vuoto a stato finale

Dimostriamo che la classe dei linguaggi che sono $L(P)$ per un PDA P è uguale alla classe dei linguaggi che sono $N(P)$ per un PDA P. Si tratta proprio della classe dei linguaggi liberi dal contesto, come vedremo nel Paragrafo 6.3. La prima costruzione spiega come costruire un PDA P_F che accetta un linguaggio L per stato finale a partire da un PDA P_N che accetta L per stack vuoto.

Teorema 6.9 Se $L = N(P_N)$ per un PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, allora esiste un PDA P_F tale che $L = L(P_F)$.

DIMOSTRAZIONE L'idea su cui poggia la dimostrazione è illustrata nella Figura 6.4. Ci serviamo di un nuovo simbolo X_0, che non deve appartenere a Γ; X_0 è sia il simbolo iniziale di P_F sia un segnale che indica quando P_N ha svuotato lo stack. Quando P_F vede X_0 alla sommità del proprio stack, sa che P_N vuoterebbe lo stack sullo stesso input.

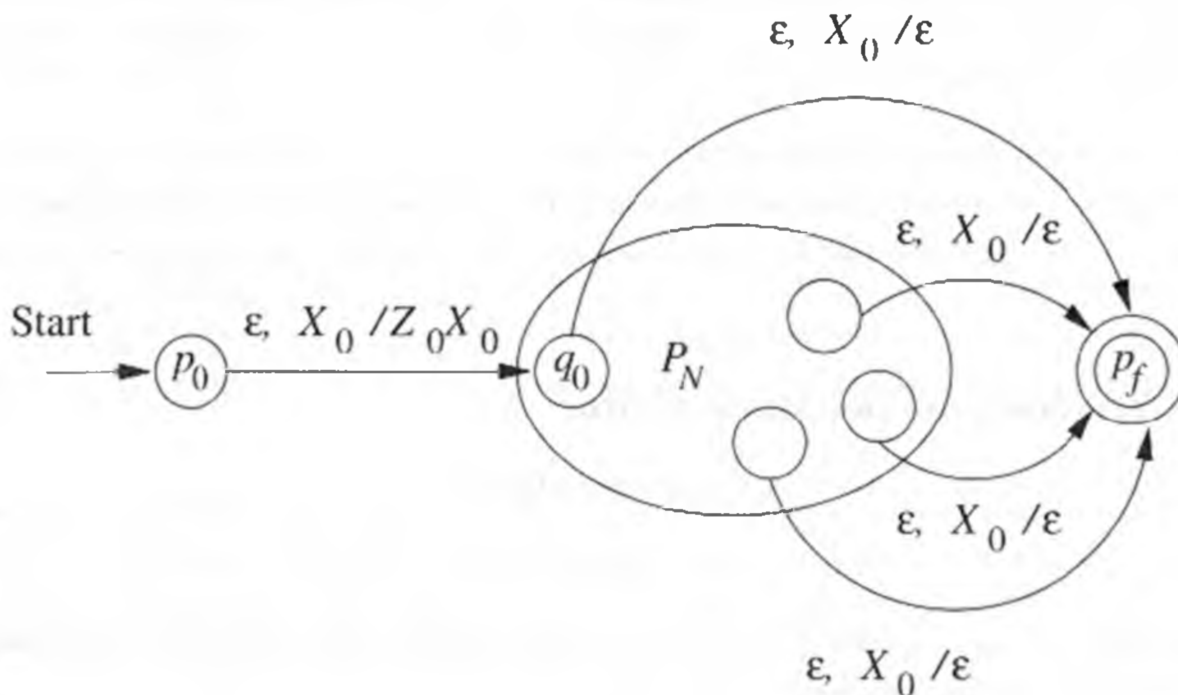


Figura 6.4 P_F simula P_N e accetta se P_N vuota il proprio stack.

Abbiamo inoltre bisogno di un nuovo stato iniziale, p_0, la cui unica funzione è inserire Z_0, il simbolo iniziale di P_N, in cima allo stack ed entrare nello stato q_0, lo stato iniziale di P_N. A questo punto P_F simula P_N finché lo stack di P_N è vuoto: P_F se ne accorge perché vede X_0 alla sommità dello stack. Infine occorre un altro nuovo stato p_f, che è lo stato accettante di P_F. Questo PDA entra nello stato p_f quando scopre che P_N avrebbe svuotato il proprio stack.

La specifica di P_F è:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

dove δ_F è definita come segue.

1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. Nel suo stato iniziale P_F compie una transizione spontanea allo stato iniziale di P_N, inserendone il simbolo iniziale Z_0 sullo stack.

2. Per tutti gli stati q in Q, gli input a in Σ o $a = \epsilon$, e i simboli di stack Y in Γ, $\delta_F(q, a, Y)$ contiene tutte le coppie in $\delta_N(q, a, Y)$.

3. Oltre alla regola (2), $\delta_F(q, \epsilon, X_0)$ contiene (p_f, ϵ) per ogni stato q in Q.

Dobbiamo dimostrare che w è in $L(P_F)$ se e solo se w è in $N(P_N)$.

(Se) Sappiamo che $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$ per uno stato q. Per il Teorema 6.5 possiamo inserire X_0 in fondo allo stack e ricavare $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, X_0)$. Per la regola (2), P_F contiene tutte le mosse di P_N; possiamo quindi concludere che $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0)$. Combinando questa sequenza di mosse con le mosse iniziale e finale delle regole (1) e (3) otteniamo:

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0) \underset{P_F}{\vdash} (p_f, \epsilon, \epsilon) \tag{6.1}$$

Di conseguenza P_F accetta w per stato finale.

(Solo-se) Per l'inverso dobbiamo solo osservare che le transizioni aggiunte nelle regole (1) e (3) limitano a pochi casi l'accettazione di w per stato finale. Dobbiamo usare la regola (3) nell'ultimo passo, a condizione però che lo stack di P_F contenga solo X_0. Il simbolo X_0 può comparire nello stack soltanto in fondo. Inoltre la regola (1) può e *deve* essere usata solamente al primo passo.

Di conseguenza una computazione di P_F che accetti w deve somigliare alla sequenza (6.1). Inoltre la parte interna della computazione, cioè tutti i passi tranne il primo e l'ultimo, dev'essere una computazione di P_N con X_0 in fondo allo stack. Ciò si deve al fatto che, tranne il primo e l'ultimo passo, P_F non può compiere nessuna transizione che non sia anche una transizione di P_N, e X_0 non può trovarsi in cima, altrimenti la computazione finirebbe al passo successivo. Concludiamo che $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$, ossia che w è in $N(P_N)$. □

Esempio 6.10 Definiamo un PDA che elabori sequenze di `if` ed `else` in un programma C, dove i sta per `if` ed e sta per `else`. Dal Paragrafo 5.3.1 sappiamo che c'è un problema ogni volta che il numero di `else` in un prefisso eccede il numero di `if`, perché

in quel caso non possiamo abbinare ogni `else` all'`if` che lo precede. Useremo perciò un simbolo di stack Z per contare la differenza tra il numero di i e il numero di e letti. Questo semplice PDA, di un solo stato, è rappresentato dal diagramma di transizione della Figura 6.5.

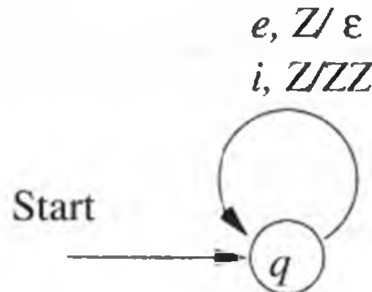


Figura 6.5 Un PDA che accetta, per stack vuoto, sequenze scorrette di if ed else.

Inseriamo una Z quando leggiamo una i e la eliminiamo quando leggiamo una e. Poiché partiamo con una Z nello stack, in effetti seguiamo la regola per cui se lo stack contiene Z^n, allora il numero di i letti supera quello degli e di $n-1$. In particolare lo stack è vuoto se abbiamo letto una e in più rispetto alle i, e l'input letto fino a questo punto è diventato illecito per la prima volta. Queste sono le stringhe che il PDA accetta per stack vuoto. La specifica formale di P_N è:

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

dove δ_N è definita come segue.

1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$. Questa regola inserisce una Z quando leggiamo una i.

2. $\delta_N(q, e, Z) = \{(q, \epsilon)\}$. Questa regola elimina una Z quando leggiamo una e.

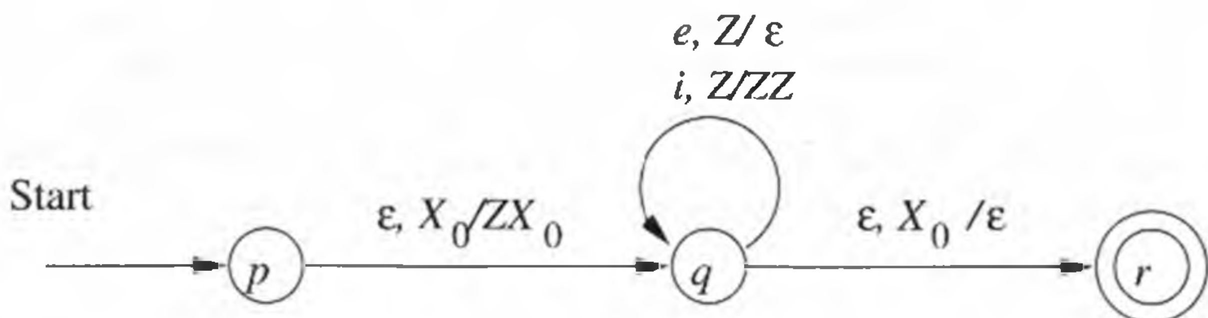


Figura 6.6 Costruzione di un PDA che accetta per stato finale dal PDA della Figura 6.5.

Costruiamo ora, a partire da P_N, un PDA P_F che accetta lo stesso linguaggio, ma per stato finale; il diagramma di transizione per P_F è illustrato dalla Figura 6.6.[4] Introduciamo un nuovo stato iniziale p e uno stato accettante r. Useremo X_0 come segnale di fondo dello stack. P_F è definito in termini formali da

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

dove δ_F consiste di quattro regole.

1. $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$. Questa regola fa partire P_F simulando P_N, con X_0 come indicatore di fondo dello stack.

2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$. Questa regola inserisce una Z quando leggiamo una i, simulando P_N.

3. $\delta_F(q, e, Z) = \{(q, \epsilon)\}$. Questa regola elimina una Z quando leggiamo una e, simulando P_N.

4. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$. In questo modo P_F accetta nei casi in cui P_N svuota il proprio stack.

□

6.2.4 Da stato finale a stack vuoto

Procediamo ora nella direzione opposta: prendiamo un PDA P_F che accetta un linguaggio L per stato finale e ne costruiamo un altro, P_N, che accetta L per stack vuoto. La costruzione è semplice ed è illustrata nella Figura 6.7. Si aggiunge una ϵ-transizione a un nuovo stato p da ogni stato accettante di P_F. Quando si trova nello stato p, P_N svuota lo stack senza consumare input. Di conseguenza, se P_F entra in uno stato accettante dopo aver consumato l'input w, P_N svuota lo stack dopo aver consumato w.

Per evitare il caso in cui P_F svuota lo stack per una stringa che non va accettata, dotiamo P_N di un indicatore di fondo dello stack, X_0. L'indicatore fa da simbolo iniziale per P_N; come nella costruzione del Teorema 6.9, P_N deve partire in un nuovo stato p_0, che ha la sola funzione di inserire il simbolo iniziale di P_F sullo stack e passare allo stato iniziale di P_F. La costruzione è illustrata nella Figura 6.7 e definita formalmente nel teorema che segue.

Teorema 6.11 Sia $L = L(P_F)$ per un PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Allora esiste un PDA P_N tale che $L = N(P_N)$.

---

[4] Mentre la costruzione nel Teorema 6.9 utilizza p_0 e p_f, qui si fa uso dei nuovi stati p ed r. Il lettore non se ne preoccupi: i nomi degli stati sono ovviamente arbitrari.

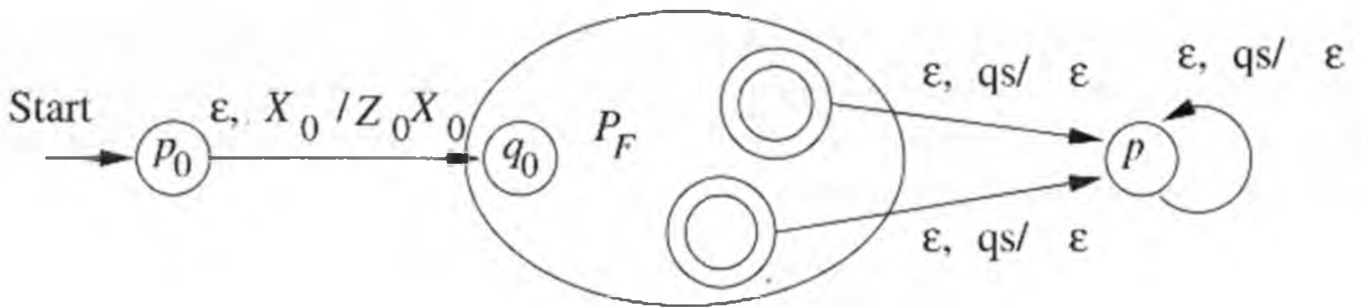


Figura 6.7 P_N simula P_F e vuota lo stack quando, e solo quando, P_F entra in uno stato accettante ("qs" = qualsiasi).

DIMOSTRAZIONE La costruzione è quella suggerita nella Figura 6.7. Sia

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

dove δ_N è definita come segue.

1. $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. Cominciamo inserendo il simbolo iniziale di P_F nello stack e andando nello stato iniziale di P_F.

2. Per ogni stato q in Q, ogni simbolo di input a in Σ (o $a = \epsilon$) e ogni Y in Γ, $\delta_N(q, a, Y)$ contiene tutte le coppie presenti in $\delta_F(q, a, Y)$. In altre parole P_N simula P_F.

3. Per tutti gli stati accettanti q in F e i simboli di stack Y in Γ (o $Y = X_0$), $\delta_N(q, \epsilon, Y)$ contiene (p, ϵ). Per questa regola, ogni volta che P_F accetta, P_N può cominciare a vuotare il suo stack senza consumare ulteriore input.

4. Per tutti i simboli di stack Y in Γ (o $Y = X_0$), $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$. Giunto nello stato p, il che è possibile solo quando P_F ha accettato, P_N elimina ogni simbolo nel suo stack fino a vuotarlo. Non si consuma altro input.

Dobbiamo ora dimostrare che w è in $N(P_N)$ se e solo se w è in $L(P_F)$. Si procede secondo la linea della dimostrazione del Teorema 6.9. La parte "se" è una simulazione diretta, mentre la parte "solo-se" richiede un esame delle poche operazioni che il PDA costruito, P_N, può compiere.

(Se) Supponiamo $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, \alpha)$ per uno stato accettante q e una stringa di stack α. Avvalendoci del fatto che ogni transizione di P_F è una mossa di P_N, e invocando il Teorema 6.5 per tenere X_0 sotto i simboli di Γ nello stack, sappiamo che $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha X_0)$. Allora in P_N valgono le relazioni

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \epsilon, \epsilon)$$

La prima mossa deriva dalla regola (1) della costruzione di P_N, mentre l'ultima sequenza di mosse deriva dalle regole (3) e (4). Di conseguenza w è accettata da P_N per stack vuoto.

(Solo se) P_N può vuotare il proprio stack solo entrando nello stato p, perché in fondo allo stack c'è X_0 e P_F non ha mosse per X_0. P_N può entrare nello stato p solo se P_F entra in uno stato accettante. La prima mossa di P_N è senz'altro quella derivata dalla regola (1). Ecco dunque come si configura ogni computazione accettante di P_N:

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \underset{P_N}{\overset{*}{\vdash}} (q, \epsilon, \alpha X_0) \underset{P_N}{\overset{*}{\vdash}} (p, \epsilon, \epsilon)$$

dove q è uno stato accettante di P_F.

Oltre a ciò, tra le ID (q_0, w, Z_0X_0) e $(q, \epsilon, \alpha X_0)$, tutte le mosse sono mosse di P_F. In particolare X_0 non è mai in cima allo stack prima di raggiungere la ID $(q, \epsilon, \alpha X_0)$.[5] Concludiamo quindi che la stessa computazione può avvenire in P_F, ovviamente senza X_0 nello stack; in altre parole $(q_0, w, Z_0) \underset{P_F}{\overset{*}{\vdash}} (q, \epsilon, \alpha)$. Cioè P_F accetta w per stato finale, e quindi w è in $L(P_F)$. □

6.2.5 Esercizi

Esercizio 6.2.1 Definite un PDA per ognuno dei seguenti linguaggi. A seconda della convenienza si può accettare per stato finale o per stack vuoto.

* a) $\{0^n1^n \mid n \geq 1\}$.

b) L'insieme di tutte le stringhe di 0 e di 1 tali che nessun prefisso abbia più 1 che 0.

c) L'insieme di tutte le stringhe di 0 e di 1 con lo stesso numero di 0 e di 1.

! **Esercizio 6.2.2** Definite un PDA per ognuno dei seguenti linguaggi.

* a) $\{a^ib^jc^k \mid i = j \text{ oppure } j = k\}$. Si noti che questo linguaggio è diverso da quello dell'Esercizio 5.1.1(b).

b) L'insieme di tutte le stringhe con un numero doppio di 0 rispetto agli 1.

!! **Esercizio 6.2.3** Definite un PDA per ognuno dei seguenti linguaggi.

a) $\{a^ib^jc^k \mid i \neq j \text{ oppure } j \neq k\}$.

b) L'insieme di tutte le stringhe di a e b che *non* sono della forma ww, cioè non sono formate da una stringa ripetuta.

[5] Anche se α può essere ϵ, nel qual caso P_F vuota lo stack nel momento stesso in cui accetta.

*! **Esercizio 6.2.4** Sia P un PDA con il linguaggio per stack vuoto $L = N(P)$, e supponiamo che ϵ non sia in L. Descrivete come modificare P in modo tale che accetti $L \cup \{\epsilon\}$ per stack vuoto.

Esercizio 6.2.5 Sia $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ un PDA in cui δ è definita come segue.

$$
\begin{array}{lll}
\delta(q_0, a, Z_0) = (q_1, AAZ_0) & \delta(q_0, b, Z_0) = (q_2, BZ_0) & \delta(q_0, \epsilon, Z_0) = (f, \epsilon) \\
\delta(q_1, a, A) = (q_1, AAA) & \delta(q_1, b, A) = (q_1, \epsilon) & \delta(q_1, \epsilon, Z_0) = (q_0, Z_0) \\
\delta(q_2, a, B) = (q_3, \epsilon) & \delta(q_2, b, B) = (q_2, BB) & \delta(q_2, \epsilon, Z_0) = (q_0, Z_0) \\
\delta(q_3, \epsilon, B) = (q_2, \epsilon) & \delta(q_3, \epsilon, Z_0) = (q_1, AZ_0) &
\end{array}
$$

Poiché c'è una sola mossa per ogni insieme, abbiamo omesso le parentesi graffe dalle regole.

* a) Scrivete una traccia di esecuzione (una sequenza di ID) a dimostrazione che la stringa *bab* è in $L(P)$.

b) Scrivete una traccia di esecuzione a dimostrazione che *abb* è in $L(P)$.

c) Scrivete il contenuto dello stack dopo che P ha letto $b^7 a^4$.

! d) Descrivete in termini informali $L(P)$.

Esercizio 6.2.6 Considerate il PDA P dell'Esercizio 6.1.1.

a) Convertite P in un altro PDA P_1 che accetta per stack vuoto lo stesso linguaggio che P accetta per stato finale; ossia $N(P_1) = L(P)$.

b) Trovate un PDA P_2 tale che $L(P_2) = N(P)$; tale cioè che P_2 accetta per stato finale ciò che P accetta per stack vuoto.

! **Esercizio 6.2.7** Mostrate che se P è un PDA, allora esiste un PDA P_2 con due soli simboli di stack tale che $L(P_2) = L(P)$. *Suggerimento*: date una codifica binaria dell'alfabeto di stack di P.

*! **Esercizio 6.2.8** Un PDA si dice *vincolato* se a ogni transizione può aumentare l'altezza dello stack di non più di un simbolo. In altre parole, se $\delta(q, a, Z)$ contiene (p, γ), deve essere $|\gamma| \leq 2$. Dimostrate che se P è un PDA, allora esiste un PDA vincolato P_3 tale che $L(P) = L(P_3)$.

6.3 Equivalenza di PDA e CFG

Dimostriamo ora che i linguaggi definiti dai PDA sono proprio i linguaggi liberi dal contesto. La via da seguire è indicata nella Figura 6.8. Lo scopo è quello di provare che le tre classi di linguaggi che seguono coincidono.

1. I linguaggi liberi dal contesto, cioè quelli definiti dalle CFG.
2. I linguaggi accettati per stato finale da un PDA.
3. I linguaggi accettati per stack vuoto da un PDA.

Abbiamo già dimostrato che (2) e (3) coincidono. È più semplice procedere dimostrando che (1) e (3) coincidono, da cui si deduce l'equivalenza delle tre classi.

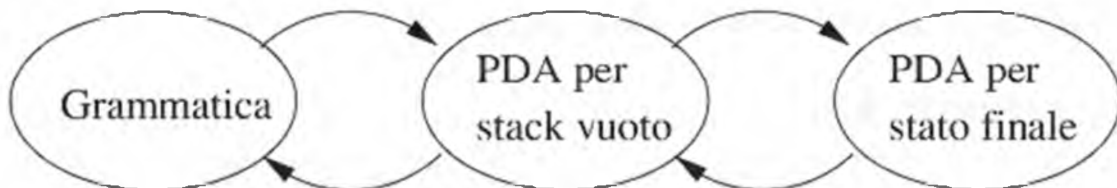


Figura 6.8 Struttura di una prova dell'equivalenza di tre modi di definire i CFL.

6.3.1 Dalle grammatiche agli automi a pila

Data una CFG G, costruiamo un PDA che ne simula le derivazioni a sinistra. Ogni forma sentenziale sinistra che non sia una stringa terminale si può scrivere nella forma $xA\alpha$, dove A è la variabile più a sinistra, x è la stringa di terminali alla sua sinistra, e α la stringa di terminali e variabili alla destra di A. Diremo che $A\alpha$ è la *coda* di questa forma. La coda di una forma sentenziale sinistra fatta di soli terminali è ϵ.

La costruzione di un PDA da una grammatica poggia sull'idea di far simulare al PDA la serie di forme sentenziali sinistre usate dalla grammatica per generare una stringa terminale data w. La coda di ogni forma $xA\alpha$ compare sullo stack con A in cima. In quel momento x è "rappresentata" dall'aver consumato x in input, lasciando ciò che in w segue x. Quindi, se $w = xy$, in input rimane y.

Supponiamo che il PDA si trovi nella ID $(q, y, A\alpha)$, che rappresenta la forma sentenziale sinistra $xA\alpha$. L'automa sceglie arbitrariamente una produzione per espandere A, poniamo $A \to \beta$. La sua mossa consiste nel sostituire A con β in cima allo stack, entrando nella ID $(q, y, \beta\alpha)$. Notiamo che il PDA ha un solo stato, q.

La ID $(q, y, \beta\alpha)$ potrebbe non rappresentare la successiva forma sentenziale sinistra, perché β può avere un prefisso formato da terminali. Di fatto può darsi che β non contenga

affatto variabili e che α abbia un prefisso composto di terminali. Per esporre in cima allo stack la prossima variabile, dobbiamo eliminare gli eventuali terminali all'inizio di $\beta\alpha$. Questi terminali vanno confrontati con i successivi simboli di input per verificare che le scelte per la derivazione a sinistra della stringa di input w siano corrette; in caso contrario la diramazione del PDA "muore".

Se in questo modo si determina una derivazione a sinistra di w, alla fine si raggiunge la forma sentenziale sinistra w. A quel punto ogni simbolo sullo stack o è stato espanso (caso delle variabili) o abbinato a un simbolo di input (caso dei terminali). Lo stack è vuoto e l'automa accetta.

Possiamo definire la costruzione più precisamente. Sia $G = (V, T, Q, S)$ una CFG. Costruiamo il PDA P che accetta $L(G)$ per stack vuoto:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

La funzione di transizione δ è definita come segue.

1. Per ogni variabile A,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \text{ è una produzione di } G\}$$

2. Per ogni terminale a, $\delta(q, a, a) = \{(q, \epsilon)\}$.

Esempio 6.12 Convertiamo in PDA la grammatica delle espressioni della Figura 5.2, riportata qui.

$$\begin{array}{lcl} I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ E & \to & I \mid E * E \mid E + E \mid (E) \end{array}$$

L'insieme dei terminali del PDA è $\{a, b, 0, 1, (,), +, *\}$. Questi otto simboli, con I ed E, formano l'alfabeto di stack. Definiamo la funzione di transizione.

a) $\delta(q, \epsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$.

b) $\delta(q, \epsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}$.

c) $\delta(q, a, a) = \{(q, \epsilon)\}$; $\delta(q, b, b) = \{(q, \epsilon)\}$; $\delta(q, 0, 0) = \{(q, \epsilon)\}$; $\delta(q, 1, 1) = \{(q, \epsilon)\}$; $\delta(q, (, () = \{(q, \epsilon)\}$; $\delta(q,),)) = \{(q, \epsilon)\}$; $\delta(q, +, +) = \{(q, \epsilon)\}$; $\delta(q, *, *) = \{(q, \epsilon)\}$.

Notiamo che (a) e (b) derivano dalla regola (1), le otto transizioni di (c) dalla regola (2). Inoltre δ è l'insieme vuoto, tranne nei casi da (a) a (c). □

Teorema 6.13 Se P è il PDA costruito dalla CFG G nel modo descritto sopra, allora $N(P) = L(G)$.

DIMOSTRAZIONE Dimostriamo che w è in $N(P)$ se e solo se è in $L(G)$.

(Se) Supponiamo che w sia in $L(G)$. Allora w ha una derivazione a sinistra

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$$

Per induzione su i proviamo che $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, dove y_i e α_i sono una rappresentazione della forma sentenziale sinistra γ_i. Cioè, sia α_i la coda di γ_i e $\gamma_i = x_i\alpha_i$. Allora y_i è la stringa per cui $x_iy_i = w$, ossia quello che resta dopo aver tolto x_i dall'input.

BASE Per $i = 1$, $\gamma_1 = S$. Quindi $x_1 = \epsilon$ e $y_1 = w$. Poiché $(q, w, S) \overset{*}{\vdash} (q, w, S)$ in 0 mosse, la base è dimostrata.

INDUZIONE Trattiamo ora il caso della seconda forma sentenziale sinistra e delle successive. Assumiamo

$$(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$$

e dimostriamo $(q, w, S) \overset{*}{\vdash} (q, y_{i+1}, \alpha_{i+1})$. Essendo una coda, α_i comincia con una variabile A. Inoltre il passo della derivazione $\gamma_i \Rightarrow \gamma_{i+1}$ comporta la sostituzione di A con il corpo di una delle sue produzioni, poniamo β. Per la regola (1) della costruzione di P possiamo sostituire A con β in cima allo stack; per la regola (2) possiamo ora abbinare i terminali in cima allo stack ai successivi simboli di input. Raggiungiamo così la ID $(q, y_{i+1}, \alpha_{i+1})$, che rappresenta la successiva forma sentenziale sinistra γ_{i+1}.

Per completare la dimostrazione notiamo che $\alpha_n = \epsilon$ perché la coda di γ_n (cioè w) è vuota. Quindi $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$, il che dimostra che P accetta w per stack vuoto.

(Solo se) Dobbiamo dimostrare un fatto più generale: che se P esegue una serie di mosse il cui effetto finale è quello di togliere una variabile A dalla cima dello stack senza mai scendere sotto A nello stack, allora A genera in G la stringa di input consumata nel corso di questo processo. Formalmente:

- se $(q, x, A) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon)$, allora $A \overset{*}{\underset{G}{\Rightarrow}} x$.

La dimostrazione procede per induzione sul numero di mosse fatte da P.

BASE Una mossa. Il solo caso possibile è che $A \to \epsilon$ sia una produzione di G, e che sia usata in una regola di tipo (1) dal PDA P. In tal caso $x = \epsilon$, e sappiamo che $A \Rightarrow \epsilon$.

INDUZIONE Supponiamo che P faccia n mosse, con $n > 1$. La prima mossa dev'essere di tipo (1), con A sostituito dal corpo di una delle sue produzioni alla sommità dello stack. Infatti si può applicare una regola di tipo (2) solo quando in cima allo stack c'è un terminale. Supponiamo che la produzione sia $A \to Y_1Y_2 \cdots Y_k$, in cui ogni Y_i è un terminale o una variabile.

Le successive $n-1$ mosse di P devono consumare x dall'input e produrre l'effetto di levare Y_1, Y_2, e così via, dallo stack, uno per volta. Possiamo scomporre x in $x_1x_2\cdots x_k$, dove x_1 è la parte di input consumata fino a quando si elimina dallo stack Y_1 (quindi lo stack all'inizio contiene $k-1$ simboli). A sua volta x_2 è la parte di input consumata nel corso della eliminazione di Y_2, e così via.

La Figura 6.9 illustra come scomporre x e gli effetti corrispondenti sullo stack. La figura suggerisce che β è BaC, quindi x è diviso in tre parti, $x_1x_2x_3$, con $x_2 = a$. Notiamo che in generale, se Y_i è un terminale, x_i dev'essere lo stesso terminale.

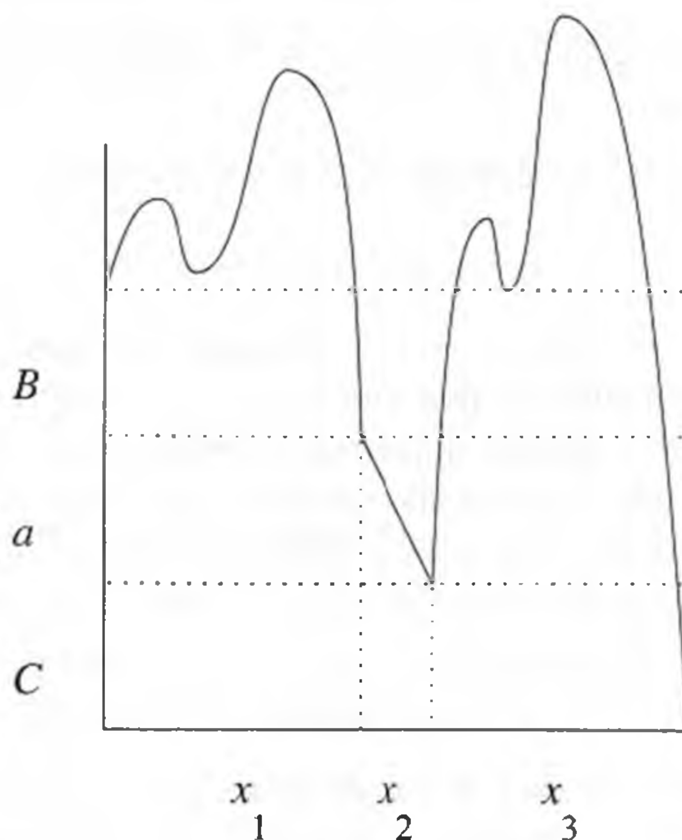


Figura 6.9 Il PDA P consuma x ed elimina BaC dallo stack.

Deduciamo che $(q, x_ix_{i+1}\cdots x_k, Y_i) \overset{*}{\vdash} (q, x_{i+1}\cdots x_k, \epsilon)$ per ogni $i = 1, 2, \ldots, k$. Inoltre, se Y_i è una variabile, si applica l'ipotesi di induzione, perché nessuna sequenza può fare più di $n-1$ mosse. Concludiamo allora che $Y_i \overset{*}{\Rightarrow} x_i$.

Se Y_i è un terminale, ci può essere solo una mossa, che abbina l'unico simbolo di x_i a Y_i, che gli è uguale. Anche in questo caso concludiamo che $Y_i \overset{*}{\Rightarrow} x_i$ senza usare alcun passo. Abbiamo ora la derivazione

$$A \Rightarrow Y_1Y_2\cdots Y_k \overset{*}{\Rightarrow} x_1Y_2\cdots Y_k \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} x_1x_2\cdots x_k$$

cioè $A \overset{*}{\Rightarrow} x$.

Per completare la dimostrazione, siano $A = S$ e $x = w$. Poiché sappiamo che w è in $N(P)$, sappiamo che $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$. Da quanto abbiamo dimostrato per induzione ricaviamo $S \overset{*}{\Rightarrow} w$, ossia w appartiene a $L(G)$. □

6.3.2 Dai PDA alle grammatiche

Completiamo ora le dimostrazioni di equivalenza provando che per ogni PDA P possiamo definire una CFG G il cui linguaggio coincide con quello accettato da P per stack vuoto. La dimostrazione si basa sul prendere atto che l'evento cruciale nel processo di elaborazione di un input da parte di un PDA è l'eliminazione netta di un simbolo dallo stack, conseguente alla lettura di una parte di input. Nell'eliminare un simbolo dallo stack, un PDA può cambiare stato; dobbiamo quindi tenere traccia dello stato in cui entra quando scende di un livello nello stack.

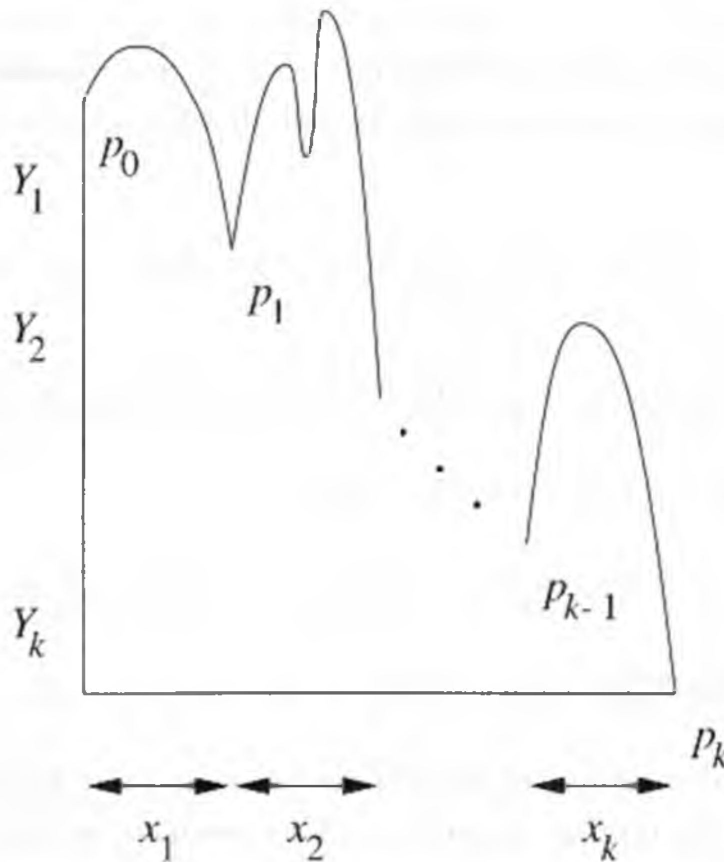


Figura 6.10 Sequenza di mosse di un PDA con l'effetto di eliminare un simbolo dallo stack.

La Figura 6.10 illustra come eliminare una serie di simboli $Y_1, Y_2, \ldots Y_k$ dallo stack. L'eliminazione di Y_1 coincide con la lettura di x_1 dall'input. Sottolineiamo che l'eliminazione è l'effetto finale di una o più mosse. Per esempio la prima mossa può sostituire Y_1 con un altro simbolo Z. La successiva può sostituire Z con UV; altre mosse possono

avere l'effetto di eliminare U, e altre ancora di eliminare V. L'effetto finale è che Y_1 è stata sostituita dal "nulla", cioè è stata eliminata; i simboli di input letti fino a quel punto formano x_1.

La Figura 6.10 illustra anche il mutamento di stato. Supponiamo che il PDA parta dallo stato p_0, con Y_1 in cima allo stack. Dopo le mosse il cui effetto è l'eliminazione di Y_1, il PDA è nello stato p_1. Da lì procede a eliminare Y_2 leggendo la stringa di input x_2, per trovarsi, dopo un certo numero di mosse, nello stato p_2, avendo eliminato Y_2 dallo stack. L'elaborazione prosegue fino a eliminare dallo stack tutti i simboli.

La costruzione di una grammatica equivalente impiega variabili che rappresentano ciascuna un "evento", con due componenti:

1. l'eliminazione effettiva di un simbolo X dallo stack

2. il passaggio dallo stato p allo stato q, dopo la sostituzione di X con ϵ sullo stack.

Rappresentiamo questa variabile con il simbolo $[pXq]$. Questa sequenza di caratteri descrive *una* variabile; non si tratta di cinque simboli distinti. La costruzione formale è data nel teorema seguente.

Teorema 6.14 Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ un PDA. Allora esiste una grammatica libera dal contesto, G, tale che $L(G) = N(P)$.

DIMOSTRAZIONE Costruiamo $G = (V, \Sigma, R, S)$, il cui insieme di variabili V contiene:

1. il simbolo speciale S come simbolo iniziale

2. tutti i simboli della forma $[pXq]$, con p e q stati in Q, e X simbolo di stack in Γ.

Le produzioni di G sono definite come segue.

a) Per ogni stato p, G contiene la produzione $S \to [q_0 Z_0 p]$. Il simbolo $[q_0 Z_0 p]$ serve a generare tutte le stringhe w che provocano l'eliminazione di Z_0 dallo stack di P nel passare dallo stato q_0 a p. Formalmente $(q_0, w, Z_0) \overset{*}{\vdash} (p, \epsilon, \epsilon)$. Queste produzioni indicano allora che lo stato iniziale S genera tutte le stringhe w che fanno svuotare lo stack in P a partire dalla ID iniziale.

b) Supponiamo che $\delta(q, a, X)$ contenga la coppia $(r, Y_1 Y_2 \cdots Y_k)$, dove:

   1. a è un simbolo in Σ oppure $a = \epsilon$

   2. k è un intero positivo arbitrario, eventualmente 0, nel qual caso la coppia è (r, ϵ).

Allora, per ogni sequenza di stati $r_1, r_2, \ldots, r_k$, G contiene la produzione

$$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$$

Questa produzione indica un modo di eliminare X e passare dallo stato q a r_k: leggere a (che può essere ϵ), usare parte dell'input per eliminare Y_1 dallo stack, passando dallo stato r a r_1, quindi leggere un'altra parte dell'input per eliminare Y_2 dallo stack e passare da r_1 a r_2, e così via.

Dimostriamo ora che l'interpretazione informale delle variabili $[qXp]$ è corretta.

- $[qXp] \overset{*}{\Rightarrow} w$ se e solo se $(q, w, X) \overset{*}{\vdash} (p, \epsilon, \epsilon)$.

(Se) Supponiamo $(q, w, X) \overset{*}{\vdash} (p, \epsilon, \epsilon)$. Proviamo che $[qXp] \overset{*}{\Rightarrow} w$ per induzione sul numero di mosse fatte dal PDA.

BASE Un passo. In questo caso (p, ϵ) dev'essere in $\delta(q, w, X)$, e w è un simbolo singolo oppure ϵ. Per come è fatta G, $[qXp] \rightarrow w$ è una produzione, quindi $[qXp] \Rightarrow w$.

INDUZIONE Supponiamo che la sequenza $(q, w, X) \overset{*}{\vdash} (p, \epsilon, \epsilon)$ richieda n passi, con $n > 1$. La prima mossa deve avere la forma

$$(q, w, X) \vdash (r_0, x, Y_1Y_2\cdots Y_k) \overset{*}{\vdash} (p, \epsilon, \epsilon)$$

dove $w = ax$ per un a che è ϵ oppure un simbolo in Σ. Ne deriva che la coppia $(r_0, Y_1Y_2\cdots Y_k)$ dev'essere in $\delta(q, a, X)$. Inoltre, per la costruzione di G, c'è una produzione $[qXr_k] \rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$, in cui:

1. $r_k = p$
2. $r_1, r_2, \ldots, r_{k-1}$ sono stati arbitrari in Q.

Come suggerito dalla Figura 6.10, osserviamo in particolare che i simboli $Y_1, Y_2, \ldots, Y_k$ vengono eliminati dallo stack uno per volta, e che possiamo scegliere p_i uguale allo stato del PDA quando si elimina Y_i, per $i = 1, 2, \ldots, k-1$. Sia $x = w_1w_2\cdots w_k$, dove w_i è l'input consumato nell'eliminare Y_i dallo stack. Vale allora $(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \epsilon, \epsilon)$.

Poiché ognuna di queste sequenze richiede meno di n mosse, possiamo applicare l'ipotesi di induzione e concludere che $[r_{i-1}Y_ir_i] \overset{*}{\Rightarrow} w_i$. Possiamo unire queste derivazioni alla prima produzione applicata e dedurre la formula

$$\begin{aligned}
&[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_1r_2]\cdots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} \\
&aw_1[r_1Y_2r_2][r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} \\
&aw_1w_2[r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} \\
&\cdots \\
&aw_1w_2\cdots w_k = w
\end{aligned}$$

dove $r_k = p$.

(Solo se) Per induzione sul numero di passi nella derivazione.

BASE Un passo. In questo caso $[qXp] \rightarrow w$ dev'essere una produzione. Questa produzione esiste solo se c'è una transizione di P in cui X viene eliminato e dallo stato q si passa in p. Perciò (p, ϵ) dev'essere in $\delta(q, a, X)$ e $a = w$. Ciò comporta che $(q, w, X) \vdash (p, \epsilon, \epsilon)$.

INDUZIONE Supponiamo che $[qXp] \overset{*}{\Rightarrow} w$ in n passi, con $n > 1$. Trattiamo esplicitamente la prima forma sentenziale, che deve avere la forma

$$[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} w$$

con $r_k = p$. Questa produzione deriva dal fatto che $(r_0, Y_1Y_2\cdots Y_k)$ è in $\delta(q, a, X)$.

Possiamo scomporre w in $w = aw_1w_2\cdots w_k$, in modo tale che $[r_{i-1}Y_ir_i] \overset{*}{\Rightarrow} w_i$ per $i = 1, 2, \ldots, k$. Dall'ipotesi di induzione sappiamo che per ogni i

$$(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \epsilon, \epsilon)$$

Se applichiamo il Teorema 6.5 per collocare le stringhe opportune dopo w_i sull'input e sotto Y_i nello stack, abbiamo anche

$$(r_{i-1}, w_iw_{i+1}\cdots w_k, Y_iY_{i+1}\cdots Y_k) \overset{*}{\vdash} (r_i, w_{i+1}\cdots w_k, Y_{i+1}\cdots Y_k)$$

Riunendo tutte le sequenze vediamo che

$$(q, aw_1w_2\cdots w_k, X) \vdash (r_0, w_1w_2\cdots w_k, Y_1Y_2\cdots Y_k) \overset{*}{\vdash}$$
$$(r_1, w_2w_3\cdots w_k, Y_2Y_3\cdots Y_k) \overset{*}{\vdash} (r_2, w_3\cdots w_k, Y_3\cdots Y_k) \overset{*}{\vdash} \cdots \overset{*}{\vdash} (r_k, \epsilon, \epsilon)$$

Poiché $r_k = p$ abbiamo dimostrato che $(q, w, X) \overset{*}{\vdash} (p, \epsilon, \epsilon)$.

Completiamo ora la dimostrazione. Per come sono formate le regole relative al simbolo iniziale S, $S \overset{*}{\Rightarrow} w$ se e solo se $[q_0Z_0p] \overset{*}{\Rightarrow} w$ per qualche p. Abbiamo così dimostrato che $[q_0Z_0p] \overset{*}{\Rightarrow} w$ se e solo se $(q, w, Z_0) \overset{*}{\vdash} (p, \epsilon, \epsilon)$, cioè se e solo se P accetta x per stack vuoto. Quindi $L(G) = N(P)$. □

Esempio 6.15 Convertiamo in una grammatica il PDA $P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ dell'Esempio 6.10. L'automa P_N accetta tutte le stringhe che violano per la prima volta la regola secondo cui ogni e (else) deve corrispondere a un i (if) precedente. Poiché P_N ha un solo stato e un solo simbolo di stack, la costruzione è molto semplice. La grammatica G comprende due sole variabili:

a) S, il simbolo iniziale, che si trova in ogni grammatica formata secondo il metodo del Teorema 6.14

b) $[qZq]$, l'unica tripla che si può costruire con gli stati e i simboli di stack di P_N.

Le produzioni della grammatica G sono le seguenti.

1. L'unica produzione per S è $S \rightarrow [qZq]$. Se il PDA avesse n stati, ci sarebbero n produzioni di questo tipo perché lo stato finale potrebbe essere uno qualsiasi di quelli. Il primo stato sarebbe lo stato iniziale e il simbolo di stack il simbolo iniziale, come nella produzione indicata.

2. Poiché $\delta_N(q, i, Z)$ contiene (q, ZZ), abbiamo la produzione $[qZq] \rightarrow i[qZq][qZq]$. In un esempio così semplice c'è di nuovo una sola produzione. Se ci fossero n stati, questa sola regola darebbe luogo a n^2 produzioni, perché i due stati centrali del corpo possono essere qualsiasi, come anche gli ultimi della testa e del corpo. In altre parole, se p ed r fossero due stati del PDA avremmo la produzione $[qZp] \rightarrow i[qZr][rZp]$.

3. Dal fatto che $\delta_N(q, e, Z)$ contiene (q, ϵ) ricaviamo la produzione

$$[qZq] \rightarrow e$$

Si noti che in questo caso l'elenco dei simboli di stack che sostituiscono Z è vuoto; l'unico simbolo nel corpo è quello, di input, che ha provocato la mossa.

Per comodità possiamo sostituire la tripla $[qZq]$ con un simbolo semplice, per esempio A. In questo modo l'intera grammatica comprende le seguenti produzioni:

$$S \rightarrow A$$
$$A \rightarrow iAA \mid e$$

Osservando che da A ed S derivano esattamente le stesse stringhe, possiamo identificarle e scrivere l'intera grammatica così:

$$G = (\{S\}, \{i, e\}, \{S \rightarrow iSS \mid e\}, S)$$

□

6.3.3 Esercizi

* **Esercizio 6.3.1** Convertite la grammatica

$$\begin{array}{lcl} S & \rightarrow & 0S1 \mid A \\ A & \rightarrow & 1A0 \mid S \mid \epsilon \end{array}$$

in un PDA che accetta per stack vuoto lo stesso linguaggio.

Esercizio 6.3.2 Convertite la grammatica

$$\begin{array}{lcl} S & \rightarrow & aAA \\ A & \rightarrow & aS \mid bS \mid a \end{array}$$

in un PDA che accetta per stack vuoto lo stesso linguaggio.

* **Esercizio 6.3.3** Convertite in una CFG il PDA $P = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$, con δ definita da:

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$
2. $\delta(q, 1, X) = \{(q, XX)\}$
3. $\delta(q, 0, X) = \{(p, X)\}$
4. $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
5. $\delta(p, 1, X) = \{(p, \epsilon)\}$
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$.

Esercizio 6.3.4 Convertite in una grammatica libera dal contesto il PDA dell'Esercizio 6.1.1.

Esercizio 6.3.5 Per ognuno dei seguenti linguaggi liberi dal contesto definite un PDA che lo accetti per stack vuoto. Potete definire prima una grammatica per il linguaggio e poi convertirla in PDA.

a) $\{a^n b^m c^{2(n+m)} \mid n \geq 0, \ m \geq 0\}$.

b) $\{a^i b^j c^k \mid i = 2j \text{ oppure } j = 2k\}$.

! c) $\{0^n 1^m \mid n \leq m \leq 2n\}$.

*! **Esercizio 6.3.6** Dimostrate che se P è un PDA, esiste un PDA P_1 con un solo stato tale che $N(P_1) = N(P)$.

! **Esercizio 6.3.7** Considerate un PDA con s stati e t simboli di stack in cui le stringhe che sostituiscono le regole di transizione hanno lunghezza minore o uguale a u. Determinate un limite superiore stretto sul numero di variabili della CFG associata all'automa secondo il metodo del Paragrafo 6.3.2.

6.4 Automi a pila deterministici

Per definizione i PDA possono essere non deterministici; il sottocaso di automa deterministico è però importante. In particolare i parser si comportano generalmente come PDA deterministici; la classe dei linguaggi accettati da questi automi è quindi interessante perché ci aiuta a capire quali costrutti sono adatti ai linguaggi di programmazione. In questo paragrafo definiamo i PDA deterministici e studiamo le loro capacità.

6.4.1 Definizione di PDA deterministico

In modo intuitivo possiamo dire che un PDA è deterministico se in ogni situazione non c'è scelta fra mosse alternative. Le scelte sono di due tipi. Se $\delta(q, a, X)$ contiene più di una coppia, allora il PDA è sicuramente non deterministico perché si può scegliere tra queste coppie nel decidere la mossa successiva. D'altro canto, anche se $\delta(q, a, X)$ è sempre un singoletto, potremmo comunque avere la scelta tra l'uso di un vero simbolo di input oppure una mossa su ϵ. Definiamo quindi un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ come *deterministico* (*DPDA, Deterministic PDA*) se e solo vengono soddisfatte le seguenti condizioni.

1. $\delta(q, a, X)$ ha al massimo un elemento per ogni q in Q, a in Σ (o $a = \epsilon$), e X in Γ.

2. Se $\delta(q, a, X)$ è non vuoto per un a in Σ, allora $\delta(q, \epsilon, X)$ dev'essere vuoto.

Esempio 6.16 Si può dimostrare che il linguaggio L_{wwr} dell'Esempio 6.2 è un CFL per il quale non esiste alcun DPDA. Tuttavia, collocando un "segnale di mezzo" c, il linguaggio diventa riconoscibile da un DPDA. In altre parole il linguaggio $L_{wcwr} = \{wcw^R \mid w \text{ è in } (\mathbf{0} + \mathbf{1})^*\}$ può essere riconosciuto da un PDA deterministico.

La strategia del DPDA consiste nel memorizzare gli 0 e gli 1 nello stack finché vede il segnale di mezzo c. A questo punto passa in un altro stato, in cui confronta i simboli di input con i simboli di stack, eliminandoli dallo stack se corrispondono. Se due simboli non corrispondono, il DPDA "muore": l'input non può essere della forma wcw^R. Se riesce a svuotare lo stack fino al simbolo iniziale, che ne contrassegna il fondo, allora accetta l'input.

L'idea è molto simile a quella del PDA nella Figura 6.2, che però è non deterministico perché nello stato q_0 ha sempre la possibilità di inserire il simbolo di input nello stack oppure di compiere una transizione su ϵ verso lo stato q_1. Deve cioè scommettere quando ha raggiunto la prima metà. Il DPDA per L_{wcwr} è rappresentato come diagramma di transizione nella Figura 6.11.

Si tratta chiaramente di un PDA deterministico, che non ha mai la scelta fra mosse diverse nello stesso stato, con lo stesso input e lo stesso simbolo di stack. Per quanto riguarda la scelta fra un simbolo di input reale ed ϵ, l'unica ϵ-transizione che compie

è quella da q_1 a q_2 con Z_0 alla sommità dello stack, ma in q_1 non ci sono altre mosse possibili quando Z_0 è alla sommità dello stack. □

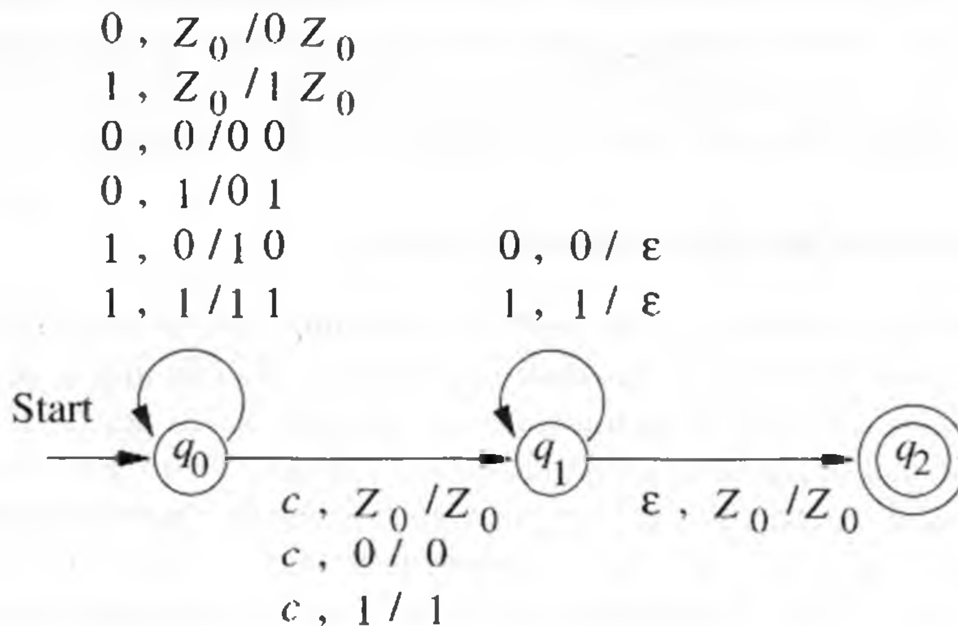


Figura 6.11 Un PDA deterministico che accetta L_{wcwr}.

6.4.2 Linguaggi regolari e PDA deterministici

I DPDA accettano una classe di linguaggi che si pone tra i linguaggi regolari e i CFL. Dimostriamo anzitutto che i linguaggi dei DPDA includono tutti i linguaggi regolari.

Teorema 6.17 Se L è un linguaggio regolare, allora $L = L(P)$ per un DPDA P.

DIMOSTRAZIONE In sostanza un DPDA può simulare un automa a stati finiti deterministico. Poiché deve avere uno stack, il PDA vi mantiene un simbolo Z_0, ma in effetti lo ignora, limitandosi a tener conto dello stato. In termini formali sia $A = (Q, \Sigma, \delta_A, q_0, F)$ un DFA. Costruiamo il DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

definendo $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ per tutti gli stati p e q in Q, tali che $\delta_A(q, a) = p$.

Affermiamo che $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, Z_0)$ se e solo se $\hat{\delta}_A(q_0, w) = p$. In altre parole P simula A servendosi degli stati. Le dimostrazioni in entrambi i sensi sono facili induzioni su $|w|$, e le lasciamo al lettore. Poiché sia A sia P accettano entrando in uno degli stati di F, concludiamo che i loro linguaggi coincidono. □

Quando consideriamo l'accettazione per stack vuoto, scopriamo che la capacità di riconoscere linguaggi in questo modo è limitata. Diciamo che un linguaggio L ha la *proprietà di prefisso* se in esso non esistono due stringhe x e y diverse, tali che x è un prefisso di y.

Esempio 6.18 Il linguaggio L_{wcwr} dell'Esempio 6.16 ha la proprietà di prefisso. Infatti non possono esserci due stringhe wcw^R e xcx^R, una delle quali è il prefisso dell'altra, salvo che non si tratti della stessa stringa. Per capirne la ragione supponiamo che wcw^R sia un prefisso di xcx^R e che $w \neq x$. Allora w dev'essere più breve di x. Perciò la c in wcw^R giunge in una posizione in cui xcx^R ha uno 0 oppure un 1. È una posizione nella prima x. Ciò contraddice l'assunto che wcw^R sia un prefisso di xcx^R.

D'altra parte esistono linguaggi molto semplici che non hanno la proprietà di prefisso. Consideriamo $\{0\}^*$, ossia l'insieme di tutte le stringhe di 0. Chiaramente in questo linguaggio esistono coppie di stringhe delle quali una è prefisso dell'altra, e dunque il linguaggio non gode della proprietà di prefisso. In effetti, prese due stringhe qualsiasi, una è prefisso dell'altra, condizione questa più forte di quella necessaria a stabilire che la proprietà di prefisso non vale. □

Notiamo che il linguaggio $\{0\}^*$ è un linguaggio regolare. Perciò non è vero che ogni linguaggio regolare è $N(P)$ per un DPDA P. La relazione che segue può essere dimostrata per esercizio.

Teorema 6.19 Un linguaggio L è $N(P)$ per un DPDA P se e solo se L gode della proprietà di prefisso ed L è $L(P')$ per un DPDA P'. □

6.4.3 DPDA e linguaggi liberi dal contesto

Come abbiamo visto, un DPDA può accettare linguaggi non regolari, come L_{wcwr}. Per provare che questo linguaggio non è regolare supponiamo che lo sia e ricorriamo al *pumping lemma*. Se n è la costante del lemma, consideriamo la stringa $w = 0^n c 0^n$, che è in L_{wcwr}. Se ora "replichiamo" questa stringa, la lunghezza del primo gruppo di 0 deve cambiare, producendo stringhe in cui il segnale di mezzo non si trova al centro. Dato che tali stringhe non sono in L_{wcwr}, abbiamo una contraddizione, e la conclusione è che L_{wcwr} non è regolare.

D'altra parte esistono CFL, come L_{wwr}, che non possono essere $L(P)$ per nessun DPDA P. La dimostrazione formale è complessa, ma l'intuizione è evidente. Se P è un DPDA che accetta L_{wwr}, allora, data una sequenza di 0, deve memorizzarli nello stack oppure compiere un'azione equivalente per conteggiare un numero arbitrario di 0. Per esempio potrebbe memorizzare un X ogni due 0 visti e usare lo stato per ricordare se il numero è pari o dispari.

Supponiamo che P abbia visto n volte 0 e poi veda 110^n. Deve verificare che ci sono

n 0 dopo 11, e a questo scopo deve togliere simboli dallo stack.[6] Ora P ha visto 0^n110^n. Se vede una stringa identica nel seguito, deve accettare perché l'input sarebbe della forma ww^R, con $w = 0^n110^n$. Se invece vede 0^m110^m, per $m \neq n$, P *non* deve accettare. Ma, essendo lo stack vuoto, non può avere memoria dell'intero n e non riesce a riconoscere correttamente L_{wwr}. La nostra conclusione è che

- i linguaggi accettati dai DPDA per stato finale includono propriamente i linguaggi regolari, ma sono inclusi propriamente nei CFL.

6.4.4 DPDA e grammatiche ambigue

Possiamo definire più precisamente la portata dei DPDA osservando che tutti i linguaggi accettati da questi automi hanno grammatiche non ambigue. Purtroppo i linguaggi dei DPDA non coincidono esattamente con il sottoinsieme dei CFL che non sono inerentemente ambigui. Per esempio L_{wwr} ha una grammatica non ambigua

$$S \to 0S0 \mid 1S1 \mid \epsilon$$

sebbene non sia il linguaggio di un DPDA. I teoremi che seguono precisano l'enunciato che chiude il paragrafo precedente.

Teorema 6.20 Se $L = N(P)$ per un DPDA P, allora L ha una grammatica libera dal contesto non ambigua.

DIMOSTRAZIONE Affermiamo che la costruzione del Teorema 6.14 dà come prodotto una CFG non ambigua G quando il PDA cui è applicata è deterministico. In primo luogo ricordiamo dal Teorema 5.29 che per dimostrare che G è non ambigua basta provare che ha derivazioni a sinistra uniche.

Supponiamo che P accetti la stringa w per stack vuoto. Poiché è deterministico, lo fa tramite una sequenza di mosse unica, e non può fare altre mosse dopo aver vuotato lo stack. Da questa sequenza di mosse possiamo determinare l'unica scelta di produzione in una derivazione a sinistra mediante la quale G deriva w. La regola di P che determina la produzione è sempre fissata univocamente. Però una regola di P, poniamo $\delta(q, a, X) = \{(r, Y_1Y_2 \cdots Y_k)\}$, potrebbe dar luogo a molte produzioni di G, con stati diversi nelle posizioni che riflettono gli stati di P dopo aver eliminato ognuno degli $Y_1, Y_2, \ldots, Y_{k-1}$. Dal momento che P è deterministico, solo una delle sequenze di scelte sarà coerente con le operazioni di P, e dunque solo una di queste produzioni condurrà alla derivazione di w. □

[6] Questo enunciato è la parte intuitiva che richiede una dimostrazione formale (difficile, per inciso). C'è un altro modo in cui P può confrontare blocchi uguali di 0?

Possiamo dimostrare qualcosa di più: anche i linguaggi che i DPDA accettano per stato finale hanno grammatiche non ambigue. Poiché sappiamo costruire direttamente grammatiche solo a partire dai PDA che accettano per stack vuoto, occorre cambiare il linguaggio in questione in modo da avere la proprietà di prefisso, e poi modificare la grammatica che ne risulta in modo da generare il linguaggio originale. Tutto ciò è possibile ricorrendo a un simbolo di "segnale di fine".

Teorema 6.21 Se $L = L(P)$ per un DPDA P, allora L ha una CFG non ambigua.

DIMOSTRAZIONE Sia \$ un simbolo ("segnale di fine") che non compare nelle stringhe di L, e sia $L' = L\$$. In altri termini le stringhe di L' sono le stringhe di L, ciascuna seguita dal simbolo \$. Allora L' ha sicuramente la proprietà di prefisso, e per il Teorema 6.19 $L' = N(P')$ per un DPDA P'.[7] Per il Teorema 6.20 esiste una grammatica non ambigua G' che genera il linguaggio $N(P')$, che è L'.

A questo punto costruiamo da G' una grammatica G tale che $L(G) = L$. Dobbiamo solo togliere dalle stringhe il segnale di fine \$. Quindi trattiamo \$ come una variabile di G e introduciamo la produzione $\$ \rightarrow \epsilon$; per il resto le produzioni di G' e G coincidono. Poiché $L(G') = L'$, ne consegue che $L(G) = L$.

Affermiamo che G è non ambigua. Nella dimostrazione le derivazioni a sinistra in G sono uguali alle derivazioni a sinistra in G', salvo il fatto che le derivazioni in G hanno un passo finale in cui la variabile \$ è sostituita da ϵ. Dunque, se una stringa di terminali w avesse due derivazioni a sinistra in G, anche la stringa $w\$$ ne avrebbe due in G'. Poiché sappiamo che G' è non ambigua, anche G lo è. □

6.4.5 Esercizi

Esercizio 6.4.1 Per ognuno dei seguenti PDA verificate se è deterministico o no mostrando che soddisfa la definizione di DPDA oppure trovando una o più regole che la violano.

a) Il PDA dell'Esempio 6.2.

* b) Il PDA dell'Esercizio 6.1.1.

c) Il PDA dell'Esercizio 6.3.3.

Esercizio 6.4.2 Definite un automa a pila deterministico per ognuno dei seguenti linguaggi:

---

[7] La dimostrazione del Teorema 6.19 compare nell'Esercizio 6.4.3, ma possiamo vedere facilmente come si costruisce P' da P. Aggiungiamo un nuovo stato q, in cui P' entra ogni volta che P si trova in uno stato accettante e l'input successivo è \$. Nello stato q, P' elimina tutti i simboli dallo stack. Inoltre P' ha bisogno dell'indicatore di fondo dello stack per evitare uno svuotamento accidentale mentre simula P.

a) $\{0^n 1^m \mid n \leq m\}$

b) $\{0^n 1^m \mid n \geq m\}$

c) $\{0^n 1^m 0^n \mid n \text{ ed } m \text{ sono arbitrari}\}$.

Esercizio 6.4.3 Dimostrate il Teorema 6.19 in tre passi:

* a) mostrate che se $L = N(P)$ per un DPDA P, allora L ha la proprietà di prefisso

! b) mostrate che se $L = N(P)$ per un DPDA P, esiste un DPDA P' tale che $L = L(P')$

*! c) mostrate che se L ha la proprietà di prefisso ed è $L(P')$ per un DPDA P', esiste un DPDA P tale che $L = N(P)$.

!! **Esercizio 6.4.4** Mostrate che il linguaggio

$$L = \{0^n 1^n \mid n \geq 1\} \cup \{0^n 1^{2n} \mid n \geq 1\}$$

è un linguaggio libero dal contesto che non è accettato da alcun DPDA. *Suggerimento*: mostrate che devono esistere due stringhe della forma $0^n 1^n$ per valori differenti di n, poniamo n_1 ed n_2, tali che un ipotetico DPDA per L entra nella stessa ID dopo aver letto le due stringhe. In termini intuitivi il DPDA deve cancellare dal suo stack quasi tutto ciò che vi colloca durante la lettura degli 0, allo scopo di controllare che sia stato visto lo stesso numero di 1. Dunque il DPDA non può decidere se accettare dopo aver letto n_1 o n_2 simboli 1.

6.5 Riepilogo

- *Automi a pila*: un PDA è un automa a stati finiti non deterministico dotato di uno stack che può essere usato per memorizzare una stringa di lunghezza arbitraria. Lo stack può essere letto e modificato solo alla sommità.

- *Mosse di un automa a pila*: un PDA sceglie la sua mossa successiva basandosi sullo stato corrente, sul simbolo di input a seguire e sul simbolo alla sommità dello stack. Inoltre può scegliere di fare una mossa a prescindere dal simbolo di input e senza consumarlo. Essendo non deterministico, il PDA può avere un numero finito di scelte per ogni mossa; ogni scelta comprende il nuovo stato e la stringa che deve sostituire il simbolo in cima allo stack.

- *Accettazione negli automi a pila*: un PDA segnala l'accettazione in due modi alternativi: entrando in uno stato accettante oppure vuotando lo stack. I due metodi sono equivalenti, nel senso che qualunque linguaggio venga accettato tramite un metodo è accettato anche tramite l'altro (da un altro PDA).

- *Descrizioni istantanee*: per descrivere la "condizione corrente" di un PDA ci serviamo di ID composte da uno stato, dall'input residuo e dal contenuto dello stack. Una funzione di transizione $\vdash$ tra le ID rappresenta le singole mosse di un PDA.

- *Automi a pila e grammatiche*: i linguaggi accettati dai PDA per stato finale o stack vuoto sono esattamente i linguaggi liberi dal contesto.

- *Automi a pila deterministici*: un PDA è deterministico se non ha mai la scelta fra mosse alternative per un dato stato, un simbolo di input (incluso ϵ) e un simbolo di stack. Inoltre non ha mai scelta tra una mossa che consuma input effettivo e una su input ϵ.

- *Accettazione negli automi a pila deterministici*: i due modi di accettazione, per stato finale e per stack vuoto, nei DPDA non hanno la stessa portata. I linguaggi accettati per stack vuoto coincidono con quelli accettati per stato finale che hanno la proprietà di prefisso: nessuna stringa nel linguaggio è il prefisso di un'altra parola nel linguaggio stesso.

- *I linguaggi accettati dai DPDA*: i DPDA accettano (per stato finale) tutti i linguaggi regolari e alcuni linguaggi non regolari. I linguaggi dei DPDA sono liberi dal contesto e hanno tutti CFG non ambigue. Perciò si collocano, in senso stretto, tra i linguaggi regolari e i linguaggi liberi dal contesto.

6.6 Bibliografia

La nozione di automa a pila viene attribuita a studi indipendenti di Oettinger [4] e Schutzenberger [5]. Anche l'equivalenza tra gli automi a pila e i linguaggi liberi dal contesto è il risultato di scoperte autonome. Compare infatti in un rapporto tecnico del MIT scritto da N. Chomsky nel 1961, ma la sua prima pubblicazione si deve a Evey [1].

I PDA deterministici sono stati introdotti per la prima volta da Fischer [2] e Schutzenberger [5], e sono diventati poi importanti come modello per i parser. In particolare [3] introduce le "grammatiche LR(k)", una sottoclasse delle CFG che generano esattamente i linguaggi dei DPDA. Le grammatiche LR(k) sono a loro volta il fondamento di YACC, il generatore di parser trattato nel Paragrafo 5.3.2.

1. J. Evey, "Application of pushdown store machines," *Proc. Fall Joint Computer Conference* (1963), AFIPS Press, Montvale, NJ, pp. 215–227.

2. P. C. Fischer, "On computability by certain classes of restricted Turing machines," *Proc. Fourth Annl. Symposium on Switching Circuit Theory and Logical Design* (1963), pp. 23–32.

3. D. E. Knuth, "On the translation of languages from left to right," *Information and Control* **8**:6 (1965), pp. 607–639.

4. A. G. Oettinger, "Automatic syntactic analysis and the pushdown store," *Proc. Symposia on Applied Math.* **12** (1961), American Mathematical Society, Providence, RI.

5. M. P. Schutzenberger, "On context-free languages and pushdown automata," *Information and Control* **6**:3 (1963), pp. 246–264.

Capitolo 7

Proprietà dei linguaggi liberi dal contesto

Completiamo lo studio dei linguaggi liberi dal contesto trattando alcune loro proprietà. Il nostro primo compito è semplificare le grammatiche libere dal contesto in modo da dimostrare più facilmente proprietà dei CFL; se un linguaggio è un CFL, possiamo infatti provare che ha una grammatica di forma speciale.

Dimostreremo poi un *pumping lemma* per i CFL. Si tratta di un teorema analogo al Teorema 4.1 per i linguaggi regolari, ma che può essere usato per dimostrare che un linguaggio non è libero dal contesto. Successivamente considereremo proprietà dei tipi già studiati per i linguaggi regolari nel Capitolo 4: proprietà di chiusura e di decisione. Vedremo che alcune proprietà di chiusura (ma non tutte) di cui godono i linguaggi regolari valgono anche per i CFL. Analogamente certe questioni riguardanti i CFL sono risolte da algoritmi che generalizzano quelli sviluppati per i linguaggi regolari, ma esistono anche questioni sui CFL cui non siamo in grado di dare risposta.

7.1 Forme normali per grammatiche libere dal contesto

In questo paragrafo intendiamo mostrare che ogni CFL (senza ϵ) è generato da una CFG le cui produzioni sono tutte della forma $A \to BC$ o $A \to a$, dove A, B e C sono variabili e a è un terminale. Questa forma è nota come *forma normale di Chomsky*. Per arrivarci dobbiamo compiere alcune semplificazioni preliminari, utili di per sé in diverse situazioni.

1. Dobbiamo eliminare i "simboli inutili": variabili o terminali che non compaiono in nessuna derivazione di una stringa di terminali dal simbolo iniziale.

2. Dobbiamo eliminare le "ϵ-produzioni", cioè quelle della forma $A \to \epsilon$ per una variabile A.

3. Dobbiamo eliminare le "produzioni unitarie", cioè quelle della forma $A \to B$, con A e B variabili.

7.1.1 Eliminazione di simboli inutili

Diremo che un simbolo X è *utile* per una grammatica $G = (V, T, P, S)$ se esiste una derivazione della forma $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$, dove w appartiene a T^*. Notiamo che X può provenire da V o da T, e che la forma sentenziale $\alpha X \beta$ può essere la prima o l'ultima della derivazione. Se X non è utile, diremo che è *inutile*. Ovviamente il linguaggio generato da una grammatica non cambia se omettiamo un simbolo inutile; possiamo quindi scoprirli tutti ed eliminarli.

Per eliminarli, stabiliamo anzitutto due cose che un simbolo deve poter fare per essere utile.

1. Diciamo che X è un *generatore* se esiste una stringa terminale w tale che $X \overset{*}{\Rightarrow} w$. Ogni terminale è un generatore perché genera se stesso tramite una derivazione di zero passi.

2. Diciamo che X è *raggiungibile* se esiste una derivazione $S \overset{*}{\Rightarrow} \alpha X \beta$ per qualche α e β.

Un simbolo utile dev'essere senz'altro sia generatore sia raggiungibile. Se eliminiamo prima i simboli non generatori e poi, dalla grammatica che ne risulta, i simboli non raggiungibili, rimarranno solo, come dimostreremo, i simboli utili.

Esempio 7.1 Consideriamo la grammatica:

$$\begin{aligned} S &\to AB \mid a \\ A &\to b \end{aligned}$$

Tutti i simboli, tranne B, sono generatori; a e b generano se stessi; S genera a e A genera b. Se eliminiamo B, dobbiamo eliminare anche la produzione $S \to AB$; ne risulta la grammatica:

$$\begin{aligned} S &\to a \\ A &\to b \end{aligned}$$

Ora soltanto S e a sono raggiungibili da S. Eliminando A e b rimane soltanto la produzione $S \to a$. Questa produzione forma da sola una grammatica il cui linguaggio è $\{a\}$, che coincide con quello della grammatica di partenza.

Se verifichiamo prima la raggiungibilità, scopriamo che tutti i simboli della grammatica

$$S \rightarrow AB \mid a$$
$$A \rightarrow b$$

sono raggiungibili. Se ora eliminiamo il simbolo B, che non è un generatore, otteniamo una grammatica con simboli inutili, in particolare A e b. □

Teorema 7.2 Sia $G = (V, T, P, S)$ una CFG, e poniamo $L(G) \neq \emptyset$, cioè G genera almeno una stringa. Sia $G_1 = (V_1, T_1, P_1, S)$ la grammatica costruita come segue.

1. Per prima cosa eliminiamo i simboli che non sono generatori e le produzioni in cui figurano. Sia $G_2 = (V_2, T_2, P_2, S)$ la grammatica risultante. Notiamo che S è un generatore, per l'ipotesi che $L(G)$ contiene almeno una stringa; dunque S non è stato eliminato.

2. In un secondo tempo eliminiamo i simboli non raggiungibili nella grammatica G_2.

Allora G_1 non contiene simboli inutili e $L(G_1) = L(G)$.

DIMOSTRAZIONE Sia X un simbolo superstite, cioè contenuto in $V_1 \cup T_1$. Sappiamo che $X \overset{*}{\underset{G}{\Rightarrow}} w$ per una stringa w in T^*. Inoltre ogni simbolo utilizzato nella derivazione di w da X è un generatore. Quindi $X \overset{*}{\underset{G_2}{\Rightarrow}} w$.

Poiché X non è stato eliminato al secondo passo, sappiamo anche che esistono α e β tali che $S \overset{*}{\underset{G_2}{\Rightarrow}} \alpha X \beta$. Per di più, ogni simbolo che compare nella derivazione è raggiungibile, quindi $S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta$.

Sappiamo che ogni simbolo in $\alpha X \beta$ è raggiungibile e che tutti questi simboli appartengono a $V_2 \cup T_2$, dunque sono dei generatori in G_2. Nella derivazione di una stringa terminale, poniamo $\alpha X \beta \overset{*}{\underset{G_2}{\Rightarrow}} xwy$, compaiono solo simboli raggiungibili da S perché sono raggiunti da simboli in $\alpha X \beta$. Perciò questa è anche una derivazione di G_1:

$$S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta \overset{*}{\underset{G_1}{\Rightarrow}} xwy$$

Concludiamo che X è utile in G_1 e che, poiché X è un simbolo arbitrario di G_1, in G_1 non ci sono simboli inutili.

L'ultimo punto da dimostrare è $L(G_1) = L(G)$. Come di consueto, per provare che due insiemi coincidono dimostriamo che ciascuno è contenuto nell'altro.

$L(G_1) \subseteq L(G)$: dato che per ottenere G_1 abbiamo soltanto eliminato simboli e produzioni da G, abbiamo $L(G_1) \subseteq L(G)$.

$L(G) \subseteq L(G_1)$: dobbiamo dimostrare che se w è in $L(G)$, allora w è in $L(G_1)$. Se w è in $L(G)$, allora $S \overset{*}{\underset{G}{\Rightarrow}} w$. In questa derivazione ciascun simbolo è evidentemente sia raggiungibile sia generatore. Dunque si tratta anche di una derivazione di G_1. Ossia $S \overset{*}{\underset{G_1}{\Rightarrow}} w$, e pertanto w è in $L(G_1)$. □

7.1.2 Calcolo dei simboli generatori e raggiungibili

Restano da chiarire due punti: come si computa l'insieme dei simboli generatori in una grammatica e come si computa l'insieme dei simboli raggiungibili di una grammatica. Per ambedue i problemi l'algoritmo che utilizziamo "fa del suo meglio" per scoprire i simboli dei due tipi. Mostreremo che un simbolo non è né generatore né raggiungibile se le costruzioni induttive di questi insiemi non riescono a stabilire che lo sia.

Sia $G = (V, T, P, S)$ una grammatica. Per calcolare i simboli generatori di G svolgiamo la seguente induzione.

BASE Ogni simbolo di T è palesemente generatore in quanto genera se stesso.

INDUZIONE Supponiamo che esista una produzione $A \to \alpha$ e che si sappia già che ogni simbolo di α è generatore. Allora A è generatore. Notiamo che la regola include il caso in cui $\alpha = \epsilon$; tutte le variabili che hanno ϵ come corpo di una produzione sono senz'altro generatori.

Esempio 7.3 Consideriamo la grammatica dell'Esempio 7.1. Per la base, a e b sono generatori. Per l'induzione possiamo usare la produzione $A \to b$ per concludere che A è generatore. A questo punto l'induzione è conclusa. Dato che non è stato stabilito che B è generatore non è possibile usare la produzione $S \to AB$. L'insieme di simboli generatori è dunque $\{a, b, A, S\}$. □

Teorema 7.4 L'algoritmo enunciato sopra trova tutti e soli i simboli generatori di G.

DIMOSTRAZIONE In una direzione si prova facilmente, per induzione sull'ordine in cui i simboli vengono aggiunti all'insieme dei generatori, che ogni simbolo aggiunto è davvero un generatore. Lasciamo al lettore questa parte della dimostrazione.

Per l'altra direzione supponiamo che X sia un simbolo generatore, per esempio $X \overset{*}{\underset{G}{\Rightarrow}} w$. Dimostriamo per induzione sulla lunghezza di questa derivazione che X viene riconosciuto come generatore.

BASE Zero passi. Allora X è un terminale ed è rilevato nella base.

INDUZIONE Se la derivazione consiste in n passi, per $n > 0$, allora X è una variabile. Poniamo che la derivazione sia $X \Rightarrow \alpha \overset{*}{\Rightarrow} w$, ovvero la prima produzione usata sia $X \to \alpha$. Da ogni simbolo di α deriva una stringa terminale che è parte di w, e tale derivazione avviene in meno di n passi. Per l'ipotesi induttiva ogni simbolo di α risulta quindi generatore. La parte induttiva dell'algoritmo permette di usare la produzione $X \to \alpha$ per dedurre che X è generatore. □

Consideriamo ora l'algoritmo induttivo grazie al quale troviamo l'insieme dei simboli raggiungibili della grammatica $G = (V, T, P, S)$. Possiamo di nuovo mostrare che un simbolo non è raggiungibile se non viene aggiunto all'insieme purché la procedura "faccia del suo meglio" per trovare i simboli raggiungibili.

BASE S è senz'altro raggiungibile.

INDUZIONE Supponiamo di aver scoperto che una certa variabile A è raggiungibile. Allora, per ogni produzione con A in testa, tutti i simboli nel corpo sono raggiungibili.

Esempio 7.5 Partiamo ancora dalla grammatica dell'Esempio 7.1. Per la base, S è raggiungibile. Poiché S ha produzioni con corpo AB e a, concludiamo che A, B e a sono raggiungibili. B non ha produzioni, ma A ha $A \rightarrow b$. Concludiamo perciò che b è raggiungibile. Nessun altro simbolo può entrare nell'insieme dei simboli raggiungibili $\{S, A, B, a, b\}$. □

Teorema 7.6 L'algoritmo enunciato sopra trova tutti e soli i simboli raggiungibili di G.

DIMOSTRAZIONE La dimostrazione consta di due semplici induzioni, simili al Teorema 7.4. Le lasciamo come esercizio. □

7.1.3 Eliminazione di ϵ-produzioni

Mostriamo ora che le ϵ-produzioni, sebbene spesso comode per definire grammatiche, non sono essenziali. Senza una produzione che abbia ϵ come corpo, è ovviamente impossibile generare la stringa vuota come membro di un linguaggio. Perciò dimostriamo in realtà che se un linguaggio L ha una CFG, allora $L - \{\epsilon\}$ ha una CFG senza ϵ-produzioni. Se ϵ non è in L, allora L stesso è $L - \{\epsilon\}$, quindi L ha una CFG senza ϵ-produzioni.

La strategia che seguiamo parte dall'individuazione delle variabili "annullabili". Una variabile A è *annullabile* se $A \overset{*}{\Rightarrow} \epsilon$. Se A è annullabile, allora ogni volta che A compare nel corpo di una produzione, come $B \rightarrow CAD$, da A può derivare ϵ. Scriviamo quindi due versioni della produzione, la prima senza A nel corpo ($B \rightarrow CD$), che corrisponde al caso in cui A è stata usata per generare ϵ, e la seconda con A presente ($B \rightarrow CAD$). Se però usiamo la versione con A, non possiamo permettere che da A derivi ϵ. La cosa non è difficile perché possiamo semplicemente eliminare tutte le produzioni che hanno ϵ come corpo, impedendo così che qualsiasi variabile generi ϵ.

Sia $G = (V, T, P, S)$ una CFG. Possiamo trovare tutti i simboli annullabili di G tramite il seguente algoritmo iterativo. In seguito dimostreremo che non ci sono simboli annullabili, a eccezione di quelli trovati dall'algoritmo.

BASE Se $A \rightarrow \epsilon$ è una produzione di G, allora A è annullabile.

INDUZIONE Se esiste una produzione $B \rightarrow C_1C_2 \cdots C_k$ in cui ogni C_i è annullabile, allora B è annullabile. Notiamo che per essere annullabile ogni C_i deve essere una variabile: dunque dobbiamo considerare unicamente le produzioni con sole variabili nel corpo.

Teorema 7.7 In una grammatica G i soli simboli annullabili sono le variabili trovate mediante l'algoritmo descritto.

DIMOSTRAZIONE L'enunciato del teorema è di fatto "A è annullabile se e solo se l'algoritmo la identifica come annullabile". Per la parte "se", osserviamo che, grazie a una semplice induzione sull'ordine in cui vengono scoperti i simboli annullabili, da ognuno di essi deriva effettivamente ϵ. Per quanto riguarda la parte "solo se", svolgiamo un'induzione sulla lunghezza della più breve derivazione $A \stackrel{*}{\Rightarrow} \epsilon$.

BASE Un solo passo. Allora $A \to \epsilon$ dev'essere una produzione, e A viene scoperto nella base dell'algoritmo.

INDUZIONE Supponiamo $A \stackrel{*}{\Rightarrow} \epsilon$ in n passi, con $n > 1$. Il primo passo sarà $A \Rightarrow C_1C_2 \cdots C_k \stackrel{*}{\Rightarrow} \epsilon$, dove da ogni C_i deriva ϵ in meno di n passi. Per l'ipotesi induttiva ogni C_i è dichiarata annullabile dall'algoritmo. Dalla produzione $A \Rightarrow C_1C_2 \cdots C_k \stackrel{*}{\Rightarrow} \epsilon$, con il passo di induzione deduciamo che A è annullabile. □

Esemplifichiamo ora la costruzione di una grammatica priva di ϵ-produzioni. Sia $G = (V, T, P, S)$ una CFG. Determiniamo tutti i simboli annullabili di G. Costruiamo una nuova grammatica $G_1 = (V, T, P_1, S)$, il cui insieme di produzioni P_1 è definito come segue.

Per ogni produzione $A \to X_1X_2 \cdots X_k$ di P, con $k \geq 1$, supponiamo che m dei k X_i siano annullabili. La nuova grammatica G_1 avrà 2^m versioni della produzione, dove le X_i sono presenti o assenti in tutte le possibili combinazioni. C'è però un'eccezione: se $m = k$, ossia tutti i simboli sono annullabili (e quindi lo è A). Non consideriamo pertanto il caso in cui tutte le X_i sono assenti. Notiamo inoltre che se una produzione della forma $A \to \epsilon$ è in P, non la collochiamo in P_1.

Esempio 7.8 Consideriamo la grammatica

$$\begin{aligned} S &\to AB \\ A &\to aAA \mid \epsilon \\ B &\to bBB \mid \epsilon \end{aligned}$$

Anzitutto troviamo tutti i simboli annullabili. A e B lo sono immediatamente in quanto hanno produzioni il cui corpo è dato da ϵ. Poi, dal momento che il corpo della produzione $S \to AB$ consiste di soli simboli annullabili, ricaviamo che S è annullabile. Tutte e tre le variabili sono dunque annullabili.

Costruiamo ora le produzioni della grammatica G_1. Consideriamo dapprima $S \to AB$. Tutti i simboli del corpo sono annullabili, perciò ci sono quattro combinazioni di presenza e assenza per A e B. Non essendo però consentito cancellare tutti i simboli, ci sono solo tre produzioni:

$$S \to AB \mid A \mid B$$

Passiamo poi a considerare la produzione $A \to aAA$. In seconda e terza posizione ci sono simboli annullabili, per cui anche qui esistono quattro combinazioni presente/assente. In questo caso sono permesse tutte e quattro le scelte, dato che il simbolo non

annullabile a è presente comunque. Le quattro scelte danno le produzioni:

$$A \rightarrow aAA \mid aA \mid aA \mid a$$

Notiamo che le due opzioni mediane danno la stessa produzione perché non è rilevante quale A eliminiamo, se decidiamo di eliminarne una. Di conseguenza la grammatica finale G_1 avrà solo tre produzioni per A.

Analogamente la produzione per B dà in G_1:

$$B \rightarrow bBB \mid bB \mid b$$

Le due ϵ-produzioni di G non danno nulla in G_1. Dunque G_1 è costituita dalle seguenti produzioni:

$$\begin{aligned} S &\rightarrow AB \mid A \mid B \\ A &\rightarrow aAA \mid aA \mid a \\ B &\rightarrow bBB \mid bB \mid b \end{aligned}$$

□

Concludiamo lo studio relativo all'eliminazione delle ϵ-produzioni dimostrando che la costruzione esemplificata sopra non trasforma il linguaggio, salvo il fatto che ϵ non è più presente se lo era nel linguaggio di G. Poiché è chiaro che la costruzione elimina le ϵ-produzioni, avremo una dimostrazione completa dell'affermazione che per ogni CFG G esiste una grammatica G_1 priva di ϵ-produzioni tale che

$$L(G_1) = L(G) - \{\epsilon\}$$

Teorema 7.9 Se la grammatica G_1 è costruita da G secondo lo schema illustrato per eliminare le ϵ-produzioni, allora $L(G_1) = L(G) - \{\epsilon\}$.

DIMOSTRAZIONE Dobbiamo dimostrare che se $w \neq \epsilon$, allora w è in $L(G_1)$ se e solo se è in $L(G)$. Come capita spesso, è più facile dimostrare un enunciato più generale. In questo caso dobbiamo occuparci delle stringhe terminali generate da ciascuna variabile, anche se ci interessa solo quanto viene generato dal simbolo iniziale S. Dimostreremo dunque:

- $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ se e solo se $A \overset{*}{\underset{G}{\Rightarrow}} w$ e $w \neq \epsilon$.

In ciascun caso la dimostrazione è un'induzione sulla lunghezza della derivazione.

(Solo se) Supponiamo che $A \overset{*}{\underset{G_1}{\Rightarrow}} w$. Allora senz'altro $w \neq \epsilon$ perché G_1 non ha ϵ-produzioni. Dobbiamo mostrare per induzione sulla lunghezza della derivazione che $A \overset{*}{\underset{G}{\Rightarrow}} w$.

BASE Un passo. Allora esiste una produzione $A \rightarrow w$ in G_1. Per la costruzione di G_1 esiste in G una produzione $A \rightarrow \alpha$, tale che α è w, con zero o più variabili annullabili intercalate. Dunque $A \underset{G}{\Rightarrow} \alpha \overset{*}{\underset{G}{\Rightarrow}} w$ in G, dove i passi successivi al primo, se ce ne sono, generano ϵ da tutte le variabili presenti in α.

INDUZIONE Supponiamo che la derivazione compia $n > 1$ passi. Allora la derivazione sarà $A \underset{G_1}{\Rightarrow} X_1X_2 \cdots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$. La prima produzione applicata deve provenire da una produzione $A \rightarrow Y_1Y_2 \cdots Y_m$, dove le Y sono le X, nello stesso ordine, con l'aggiunta di zero o più variabili annullabili intercalate. Possiamo inoltre scomporre w in $w_1w_2 \cdots w_k$, dove $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ per $i = 1, 2, \ldots, k$. Se X_i è un terminale, allora $w_i = X_i$, e se X_i è una variabile, la derivazione $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ compie meno di n passi. Per l'ipotesi induttiva possiamo concludere che $X_i \overset{*}{\underset{G}{\Rightarrow}} w_i$.

Costruiamo ora una derivazione corrispondente in G:

$$A \underset{G}{\Rightarrow} Y_1Y_2 \cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} X_1X_2 \cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w_1w_2 \cdots w_k = w$$

Il primo passo è un'applicazione della produzione $A \rightarrow Y_1Y_2 \cdots Y_k$, che sappiamo esistere in G. Il successivo gruppo di passi rappresenta la derivazione di ϵ da ognuna delle Y_j che non sia una delle X_i. Il gruppo finale di passi rappresenta le derivazioni delle w_i dalle X_i, che sappiamo esistere per l'ipotesi induttiva.

(Se) Supponiamo che $A \overset{*}{\underset{G}{\Rightarrow}} w$ e $w \neq \epsilon$. Mostriamo per induzione sulla lunghezza n della derivazione che $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

BASE Un passo. In questo caso $A \rightarrow w$ è una produzione di G. Poiché $w \neq \epsilon$, questa produzione è anche una produzione di G_1, e $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

INDUZIONE Supponiamo che la derivazione compia $n > 1$ passi. Allora la derivazione sarà $A \underset{G}{\Rightarrow} Y_1Y_2 \cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} w$. Possiamo scomporre $w = w_1w_2 \cdots w_m$ in modo che $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ per $i = 1, 2, \ldots, m$. Siano $X_1, X_2, \ldots, X_k$ le Y_j, nell'ordine, tali che $w_j \neq \epsilon$. Dal momento che $w \neq \epsilon$, deve valere $k \geq 1$. Di conseguenza $A \rightarrow X_1X_2 \cdots X_k$ è una produzione di G_1.

Affermiamo che $X_1X_2 \cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w$, perché le sole Y_j che non sono presenti tra le X sono state usate per generare ϵ, e dunque non contribuiscono alla derivazione di w. Poiché ogni derivazione $Y_j \overset{*}{\underset{G}{\Rightarrow}} w_j$ compie meno di n passi, possiamo applicare l'ipotesi induttiva e concludere che se $w_j \neq \epsilon$, allora $Y_j \overset{*}{\underset{G_1}{\Rightarrow}} w_j$. Dunque $A \underset{G_1}{\Rightarrow} X_1X_2 \cdots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$.

Completiamo ora la dimostrazione. Sappiamo che w è in $L(G_1)$ se e solo se $S \overset{*}{\underset{G_1}{\Rightarrow}} w$.

Ponendo $A = S$ otteniamo che w è in $L(G_1)$ se e solo se $S \overset{*}{\underset{G}{\Rightarrow}} w$ e $w \neq \epsilon$. In altre parole w è in $L(G_1)$ se e solo se w è in $L(G)$ e $w \neq \epsilon$. □

7.1.4 Eliminazione delle produzioni unitarie

Una *produzione unitaria* è una produzione della forma $A \rightarrow B$, dove sia A sia B sono variabili. Queste produzioni possono essere utili. Nell'Esempio 5.27 abbiamo visto come, grazie alle produzioni unitarie $E \rightarrow T$ e $T \rightarrow F$, sia possibile creare una grammatica non ambigua per espressioni aritmetiche semplici:

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

D'altra parte le produzioni unitarie possono complicare certe dimostrazioni e introdurre nelle derivazioni dei passi aggiuntivi di cui tecnicamente non ci sarebbe bisogno. Per esempio possiamo espandere la T nella produzione $E \rightarrow T$ in ambedue i modi possibili, sostituendola con le due produzioni $E \rightarrow F \mid T * F$. Questo cambiamento non elimina comunque le produzioni unitarie, perché abbiamo introdotto la produzione unitaria $E \rightarrow F$ che prima non faceva parte della grammatica. Espandendo ulteriormente $E \rightarrow F$ con le due produzioni per F si ottiene $E \rightarrow I \mid (E) \mid T * F$. Abbiamo ancora una produzione unitaria, ossia $E \rightarrow I$. Ma se espandiamo ancora questa I nei sei modi possibili, ne ricaviamo

$$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F$$

La produzione unitaria per E è scomparsa. Notiamo che $E \rightarrow a$ *non* è una produzione unitaria perché l'unico simbolo nel corpo è un terminale, e non già una variabile, come richiesto per le produzioni unitarie.

La tecnica suggerita, ossia l'espansione delle produzioni unitarie fino alla loro scomparsa, funziona spesso, ma può fallire se esiste un ciclo di produzioni unitarie, come $A \rightarrow B$, $B \rightarrow C$ e $C \rightarrow A$. Una tecnica di sicura riuscita consiste nel trovare prima tutte le coppie di variabili A e B tali che $A \overset{*}{\Rightarrow} B$ con una sequenza di sole produzioni unitarie. Osserviamo che è possibile avere $A \overset{*}{\Rightarrow} B$ anche se non entra in gioco alcuna produzione unitaria. Per esempio potremmo avere le produzioni $A \rightarrow BC$ e $C \rightarrow \epsilon$.

Una volta determinate tutte queste coppie, possiamo sostituire qualunque sequenza di passi di derivazione in cui $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_n \Rightarrow \alpha$ per mezzo di una produzione che usa la produzione non unitaria $B_n \rightarrow \alpha$ direttamente da A; in altre parole $A \rightarrow \alpha$. Per cominciare, ecco la costruzione induttiva delle coppie (A, B) tali che $A \overset{*}{\Rightarrow} B$ impiegando solo produzioni unitarie. Chiameremo una tale coppia *coppia unitaria*.

BASE (A, A) è una coppia unitaria per ogni variabile A. Infatti $A \stackrel{*}{\Rightarrow} A$ in zero passi.

INDUZIONE Supponiamo di aver determinato che (A, B) è una coppia unitaria e $B \rightarrow C$ è una produzione, dove C è una variabile. Allora (A, C) è una coppia unitaria.

Esempio 7.10 Consideriamo la grammatica delle espressioni dell'Esempio 5.27 riprodotta sopra. La base produce le coppie unitarie (E, E), (T, T), (F, F) e (I, I). Per quanto riguarda il passo induttivo possiamo fare le seguenti deduzioni:

1. (E, E) e la produzione $E \rightarrow T$ danno la coppia unitaria (E, T)
2. (E, T) e la produzione $T \rightarrow F$ danno la coppia unitaria (E, F)
3. (E, F) e la produzione $F \rightarrow I$ danno la coppia unitaria (E, I)
4. (T, T) e la produzione $T \rightarrow F$ danno la coppia unitaria (T, F)
5. (T, F) e la produzione $F \rightarrow I$ danno la coppia unitaria (T, I)
6. (F, F) e la produzione $F \rightarrow I$ danno la coppia unitaria (F, I).

Non si possono dedurre altre coppie: queste dieci rappresentano effettivamente tutte le derivazioni che applicano solo produzioni unitarie. □

Lo schema dovrebbe essere ormai familiare. Il fatto che l'algoritmo proposto generi tutte le coppie attese si basa su una semplice dimostrazione. Sfruttiamo quindi queste coppie per eliminare le produzioni unitarie da una grammatica e provare che i linguaggi delle due grammatiche coincidono.

Teorema 7.11 L'algoritmo descritto determina le coppie unitarie di una CFG G.

DIMOSTRAZIONE In una direzione si tratta di una semplice induzione sull'ordine in cui vengono scoperte le coppie: se (A, B) risulta una coppia unitaria, allora la derivazione $A \underset{G}{\stackrel{*}{\Rightarrow}} B$ usa solo produzioni unitarie. Lasciamo al lettore questa parte della dimostrazione.

Nell'altra direzione supponiamo che $A \underset{G}{\stackrel{*}{\Rightarrow}} B$ faccia uso soltanto di produzioni unitarie. Possiamo provare che troveremo la coppia (A, B) per induzione sulla lunghezza della derivazione.

BASE Zero passi. In questo caso $A = B$ e si aggiunge la coppia (A, B) nella base.

INDUZIONE Supponiamo che $A \overset{*}{\Rightarrow} B$ operi in n passi, per $n > 0$, dove a ogni passo si applica una produzione unitaria. La derivazione è del tipo

$$A \overset{*}{\Rightarrow} C \Rightarrow B$$

La derivazione $A \overset{*}{\Rightarrow} C$ compie $n - 1$ passi; per l'ipotesi induttiva scopriamo dunque la coppia (A, C). La parte induttiva dell'algoritmo combina poi la coppia (A, C) con la produzione $C \to B$ per dedurre la coppia (A, B). □

La procedura per eliminare le produzioni unitarie costruisce la CFG $G_1 = (V, T, P_1, S)$ a partire da una CFG $G = (V, T, P, S)$ in due fasi.

1. Si determinano tutte le coppie unitarie di G.

2. Per ogni coppia unitaria (A, B) si aggiungono a P_1 tutte le produzioni $A \to \alpha$, dove $B \to \alpha$ è una produzione non unitaria in P. Si noti che può essere $A = B$; in tal modo P_1 contiene tutte le produzioni non unitarie di P.

Esempio 7.12 Riprendiamo l'Esempio 7.10, in cui abbiamo svolto il passo (1) della costruzione descritta sopra per la grammatica delle espressioni dell'Esempio 5.27. La Figura 7.1 illustra il passo (2) dell'algoritmo, dove formiamo il nuovo insieme di produzioni usando il primo membro di una coppia come testa e tutti i corpi non unitari per il secondo membro della coppia come corpi delle nuove produzioni.

Coppia	Produzioni
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \to T * F$
(T, F)	$T \to (E)$
(T, I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \to (E)$
(F, I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Figura 7.1 Grammatica costruita nel passo (2) dell'algoritmo di eliminazione delle produzioni unitarie.

Il passo finale consiste nell'eliminare le produzioni unitarie dalla grammatica della Figura 7.1. La grammatica risultante:

$$E \to E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$T \to T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

non ha produzioni unitarie, eppure genera lo stesso insieme di espressioni della grammatica della Figura 5.19. □

Teorema 7.13 Se la grammatica G_1 è costruita dalla grammatica G tramite l'algoritmo per eliminare le produzioni unitarie, allora $L(G_1) = L(G)$.

DIMOSTRAZIONE Dimostriamo che w è in $L(G)$ se e solo se è in $L(G_1)$.

(Se) Supponiamo che $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Poiché ogni produzione di G_1 equivale a una sequenza di zero o più produzioni unitarie di G seguite da una produzione non unitaria di G, sappiamo che $\alpha \underset{G_1}{\Rightarrow} \beta$ implica $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$. In altre parole ogni passo di una derivazione in G_1 può essere sostituito da uno o più passi di derivazione in G. Unendo queste sequenze di passi concludiamo che $S \overset{*}{\underset{G}{\Rightarrow}} w$.

(Solo se) Supponiamo ora che w sia in $L(G)$. Per le equivalenze del Paragrafo 5.2 sappiamo che w ha una derivazione a sinistra, ossia $S \underset{lm}{\Rightarrow} w$. Ogni volta che si usa una produzione unitaria in una derivazione a sinistra, la variabile del corpo diventa la variabile più a sinistra, e dunque viene immediatamente sostituita. Di conseguenza la derivazione a sinistra nella grammatica G può essere scomposta in una sequenza di passi in cui zero o più produzioni unitarie sono seguite da una produzione non unitaria. Osserviamo che ogni produzione non unitaria che non è preceduta da una produzione unitaria è un "passo" di per sé. Ognuno di questi passi può essere compiuto da un'unica produzione di G_1 perché la costruzione di G_1 ha creato esattamente le produzioni che riflettono zero o più produzioni unitarie seguite da una produzione non unitaria. Perciò $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. □

Riassumiamo le semplificazioni descritte fino a questo punto. Vogliamo convertire una CFG arbitraria G in una CFG equivalente che non abbia simboli inutili, ϵ-produzioni o produzioni unitarie. Nel farlo bisogna prestare attenzione all'ordine di applicazione delle costruzioni. Un ordine sicuro è:

1. eliminare le ϵ-produzioni
2. eliminare le produzioni unitarie
3. eliminare i simboli inutili.

Come nel Paragrafo 7.1.1 bisognava ordinare opportunamente i due passi per evitare che il risultato avesse simboli inutili, anche qui occorre ordinare i tre passi secondo le indicazioni date, altrimenti il risultato potrebbe conservare alcune caratteristiche che intendevamo eliminare.

Teorema 7.14 Se G è una CFG che genera un linguaggio contenente almeno una stringa diversa da ϵ, allora esiste un'altra CFG G_1 tale che $L(G_1) = L(G) - \{\epsilon\}$, e G_1 non ha ϵ-produzioni, produzioni unitarie o simboli inutili.

DIMOSTRAZIONE Cominciamo eliminando le ϵ-produzioni con il metodo illustrato nel Paragrafo 7.1.3. Se poi eliminiamo le produzioni unitarie con il metodo del Paragrafo 7.1.4, non introduciamo alcuna ϵ-produzione, perché i corpi delle nuove produzioni sono identici a quelli delle produzioni originarie. Infine eliminiamo i simboli inutili con il metodo del Paragrafo 7.1.1. Dato che questa trasformazione elimina solo produzioni e simboli, e non introduce mai nuove produzioni, la grammatica risultante sarà ancora priva di ϵ-produzioni e produzioni unitarie. □

7.1.5 Forma normale di Chomsky

Completiamo l'esame delle semplificazioni grammaticali dimostrando che ogni CFL non vuoto, privo di ϵ, ha una grammatica G le cui produzioni sono di due semplici forme:

1. $A \to BC$, dove A, B e C sono variabili
2. $A \to a$, dove A è una variabile e a un terminale.

Inoltre G non ha simboli inutili. Una tale grammatica si dice in CNF (*Chomsky Normal Form*, forma normale di Chomsky).[1]

Per porre una grammatica in CNF si parte da una grammatica che soddisfa i vincoli del Teorema 7.14, cioè priva di ϵ-produzioni, produzioni unitarie e simboli inutili. Ogni produzione è quindi della forma $A \to a$, una forma ammessa nella CNF, oppure ha un corpo di lunghezza 2 o più. Il nostro compito è duplice:

a) far sì che i corpi di lunghezza 2 o più consistano soltanto di variabili

b) scomporre i corpi di lunghezza 3 o più in una cascata di produzioni, ciascuna con un corpo fatto di due variabili.

[1] Noam Chomsky è il linguista che per primo propose le grammatiche libere dal contesto per descrivere il linguaggio naturale, e che dimostrò che ogni CFG può essere posta in questa forma. È interessante notare che, mentre la CNF non sembra avere impieghi interessanti in linguistica, ne vedremo le applicazioni in molti altri ambiti, per esempio come modo efficiente per stabilire se una stringa appartiene a un CFL (Paragrafo 7.4.4).

Cominciamo da (a). Per ogni terminale a che compare in un corpo di lunghezza 2 o più creiamo una nuova variabile, A. Questa variabile ha una sola produzione, $A \to a$. D'ora in poi adoperiamo A anziché a in tutti i corpi di lunghezza 2 o più. A questo punto il corpo di ogni produzione o è un terminale o una sequenza di almeno due variabili senza terminali.

Per il punto (b) dobbiamo scomporre le produzioni $A \to B_1 B_2 \cdots B_k$, con $k \geq 3$, in una serie di produzioni con due variabili in ogni corpo. Definiamo $k-2$ nuove variabili, $C_1, C_2, \ldots, C_{k-2}$. La produzione originale va sostituita con le $k-1$ produzioni

$$A \to B_1C_1, \quad C_1 \to B_2C_2, \ldots, C_{k-3} \to B_{k-2}C_{k-2}, \quad C_{k-2} \to B_{k-1}B_k$$

Esempio 7.15 Convertiamo in CNF la grammatica dell'Esempio 7.12. Per il punto (a) notiamo che ci sono otto terminali, a, b, 0, 1, +, $*$, (e), ognuno dei quali compare in un corpo insieme ad altri simboli. Dobbiamo perciò introdurre otto nuove variabili corrispondenti a questi terminali, e otto produzioni in cui le nuove variabili sono sostituite dai rispettivi terminali:

$$\begin{array}{llll} A \to a & B \to b & Z \to 0 & O \to 1 \\ P \to + & M \to * & L \to (& R \to) \end{array}$$

Inserendo queste produzioni, e sostituendo ogni terminale nei corpi diversi da un singolo terminale con la variabile corrispondente, otteniamo la grammatica riprodotta nella Figura 7.2.

$$\begin{array}{lcl} E & \to & EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ T & \to & TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ F & \to & LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ I & \to & a \mid b \mid IA \mid IB \mid IZ \mid IO \\ A & \to & a \\ B & \to & b \\ Z & \to & 0 \\ O & \to & 1 \\ P & \to & + \\ M & \to & * \\ L & \to & (\\ R & \to &) \end{array}$$

Figura 7.2 Ogni corpo diventa un solo terminale o una sequenza di variabili.

Tutte le produzioni sono ora nella forma normale di Chomsky, tranne quelle con corpo di lunghezza 3: EPT, TMF e LER. Alcuni di questi corpi compaiono in più di una

produzione, ma possiamo trattarli una sola volta, definendo una nuova variabile per ciascuno. Per EPT definiamo la nuova variabile C_1 e sostituiamo l'unica produzione in cui compare, $E \to EPT$, con $E \to EC_1$ e $C_1 \to PT$.

Per TMF definiamo la nuova variabile C_2. Le due produzioni per questo corpo, $E \to TMF$ e $T \to TMF$, sono sostituite da $E \to TC_2$, $T \to TC_2$ e $C_2 \to MF$. Per LER definiamo C_3 e sostituiamo le sue tre produzioni, $E \to LER$, $T \to LER$ e $F \to LER$, con $E \to LC_3$, $T \to LC_3$, $F \to LC_3$ e $C_3 \to ER$. La grammatica finale, che è in CNF, è riprodotta nella Figura 7.3. □

$$
\begin{array}{lcl}
E & \to & EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
T & \to & TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
F & \to & LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
I & \to & a \mid b \mid IA \mid IB \mid IZ \mid IO \\
A & \to & a \\
B & \to & b \\
Z & \to & 0 \\
O & \to & 1 \\
P & \to & + \\
M & \to & * \\
L & \to & (\\
R & \to &) \\
C_1 & \to & PT \\
C_2 & \to & MF \\
C_3 & \to & ER
\end{array}
$$

Figura 7.3 Corpi costituiti da un terminale o da due variabili.

Teorema 7.16 Se G è una CFG il cui linguaggio contiene almeno una stringa diversa da ϵ, esiste una grammatica G_1 in forma normale di Chomsky tale che $L(G_1) = L(G) - \{\epsilon\}$.

DIMOSTRAZIONE Per il Teorema 7.14 possiamo trovare una CFG G_2 priva di simboli inutili, ϵ-produzioni e produzioni unitarie, tale che $L(G_2) = L(G) - \{\epsilon\}$. La trasformazione di G_2 in una grammatica CNF G_1 modifica le produzioni in modo che ogni produzione di G_2 sia simulata da una o più produzioni di G_1. Inoltre le nuove variabili di G_1 hanno ciascuna una sola produzione, così che possono essere usate unicamente come previsto.

Dimostriamo formalmente che w è in $L(G_2)$ se e solo se è in $L(G_1)$.

(Solo se) Se w ha una derivazione in G_2 è facile sostituire ogni produzione impiegata, poniamo $A \to X_1X_2 \cdots X_k$, con una sequenza di produzioni di G_1. Un passo nella

derivazione in G_2 diventa quindi uno o più passi nella derivazione di w mediante le produzioni di G_1. Se un X_i è terminale, sappiamo che G_1 ha una variabile corrispondente B_i e una produzione $B_i \to X_i$. Se $k > 2$, G_1 ha le produzioni $A \to B_1C_1$, $C_1 \to B_2C_2$, e così via, dove B_i è la variabile introdotta per il terminale X_i o è X_i stessa, nel caso questa sia una variabile. Queste produzioni simulano in G_1 un passo di una derivazione di G_2 che usa $A \to X_1X_2 \cdots X_k$. Concludiamo che in G_1 c'è una derivazione di w, e quindi w è in $L(G_1)$.

(Se) Sia w in $L(G_1)$. Allora esiste un albero sintattico in G_1 con S alla radice e w come prodotto. Lo convertiamo in albero sintattico di G_2, ancora con radice S e prodotto w.

Per prima cosa dobbiamo "disfare" la parte (b) della costruzione CNF. Supponiamo che ci sia un nodo etichettato A, con due figli etichettati B_1 e C_1, dove C_1 è una delle variabili introdotte al punto (b). Questa parte dell'albero sintattico ha quindi la forma della Figura 7.4(a). Infatti, poiché le variabili introdotte hanno ciascuna una sola produzione, esse possono comparire in un solo modo e, come abbiamo visto, tutte le variabili introdotte per la produzione $A \to B_1B_2 \cdots B_k$ devono comparire insieme.

Ognuno di questi grappoli di nodi dev'essere sostituito dalla produzione che rappresenta. La trasformazione dell'albero è illustrata nella Figura 7.4(b).

Non è ancora detto che l'albero risultante sia un albero sintattico di G_2. Infatti il passo (a) della costruzione di CNF introduce altre variabili da cui derivano singoli terminali. D'altra parte possiamo rilevarle nell'albero corrente e sostituire un nodo etichettato da una tale variabile A e il suo unico figlio, di etichetta a, con un solo nodo di etichetta a. Ogni nodo interno dell'albero forma ora una produzione di G_2. Poiché w è il prodotto di un albero sintattico in G_2, concludiamo che w è in $L(G_2)$. □

7.1.6 Esercizi

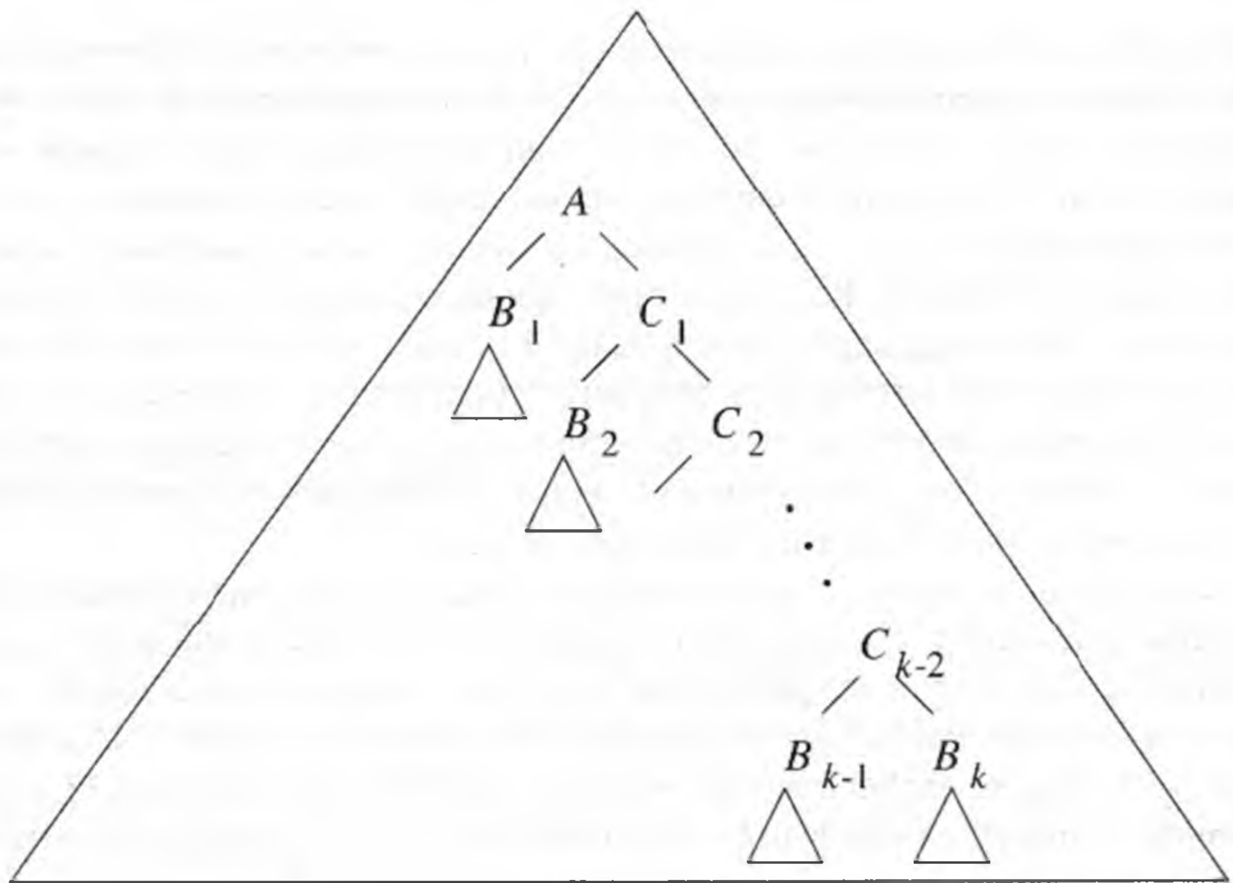
* **Esercizio 7.1.1** Trovate una grammatica equivalente a

$$\begin{array}{lcl} S & \to & AB \mid CA \\ A & \to & a \\ B & \to & BC \mid AB \\ C & \to & aB \mid b \end{array}$$

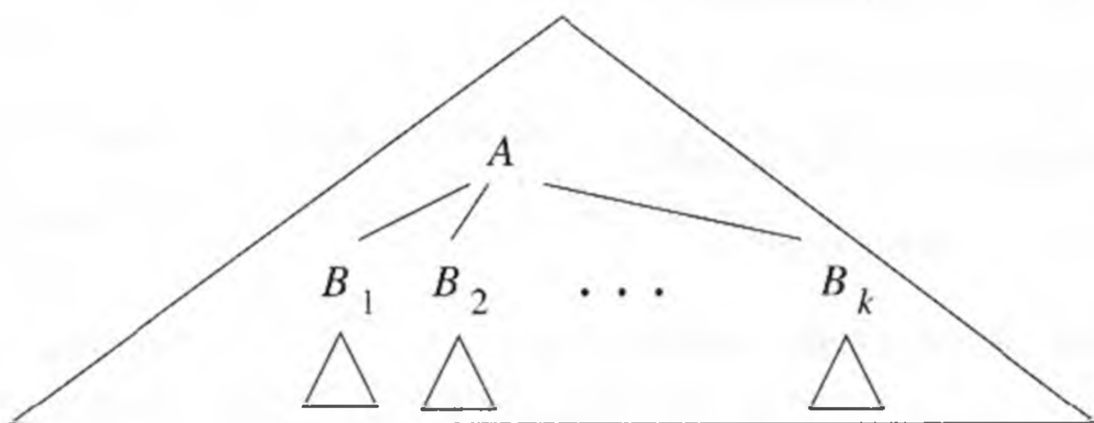
senza simboli inutili.

* **Esercizio 7.1.2** Considerate la grammatica:

$$\begin{array}{lcl} S & \to & ASB \mid \epsilon \\ A & \to & aAS \mid a \\ B & \to & SbS \mid A \mid bb \end{array}$$



(a)



(b)

Figura 7.4 Un albero sintattico in G_1 deve usare le nuove variabili in un modo specifico.

Forma normale di Greibach

C'è un'altra forma normale interessante, per la quale non daremo dimostrazioni. Ogni linguaggio non vuoto privo di ϵ è $L(G)$ per una grammatica G le cui produzioni sono della forma $A \rightarrow a\alpha$, dove a è un terminale e α una stringa di zero o più variabili. Convertire in questa forma una grammatica è complesso, anche se il compito risulta più facile se si parte, per esempio, da una grammatica in forma normale di Chomsky. Per sommi capi, espandiamo la prima variabile di ogni produzione fino a raggiungere un terminale. Per via di possibili cicli in cui non si raggiunge mai un terminale, si deve però "cortocircuitare" la procedura definendo una produzione che inserisca nel corpo un terminale come primo simbolo, seguito da variabili che generano tutte le sequenze di variabili che sarebbero state generate nel corso della derivazione di quel terminale.

Questa forma, detta *forma normale di Greibach* perché Sheila Greibach fu la prima a descrivere un modo per costruirla, ha diverse proprietà interessanti. Poiché l'applicazione di una produzione introduce esattamente un terminale nella forma sentenziale, una stringa di lunghezza n ha una derivazione di esattamente n passi. Inoltre, se applichiamo la costruzione dei PDA del Teorema 6.13 a una grammatica in forma normale di Greibach, otteniamo un PDA senza ϵ-transizioni; si dimostra così che è sempre possibile eliminarle da un PDA.

ed eseguite nell'ordine i passi seguenti.

a) Eliminate le ϵ-produzioni.

b) Eliminate le produzioni unitarie.

c) Eliminate i simboli inutili.

d) Costruite la forma normale di Chomsky.

Esercizio 7.1.3 Ripetete l'Esercizio 7.1.2 per questa grammatica:

$$\begin{array}{lcl} S & \rightarrow & 0A0 \mid 1B1 \mid BB \\ A & \rightarrow & C \\ B & \rightarrow & S \mid A \\ C & \rightarrow & S \mid \epsilon \end{array}$$

Esercizio 7.1.4 Ripetete l'Esercizio 7.1.2 per la seguente grammatica:

$$\begin{array}{lcl} S & \rightarrow & AAA \mid B \\ A & \rightarrow & aA \mid B \\ B & \rightarrow & \epsilon \end{array}$$

Esercizio 7.1.5 Ripetete l'Esercizio 7.1.2 per la seguente grammatica:

$$\begin{array}{lcl} S & \rightarrow & aAa \mid bBb \mid \epsilon \\ A & \rightarrow & C \mid a \\ B & \rightarrow & C \mid b \\ C & \rightarrow & CDE \mid \epsilon \\ D & \rightarrow & A \mid B \mid ab \end{array}$$

Esercizio 7.1.6 Definite una grammatica CNF per l'insieme delle stringhe di parentesi bilanciate. Non è necessario partire da una grammatica non CNF.

!! **Esercizio 7.1.7** Sia G una CFG con p produzioni e nessun corpo di produzione più lungo di n. Dimostrate che se $A \underset{G}{\overset{*}{\Rightarrow}} \epsilon$, esiste una derivazione di ϵ da A in non più di $(n^p - 1)/(n - 1)$ passi. Quanto ci si può effettivamente avvicinare al limite?

! **Esercizio 7.1.8** Sia G una grammatica con n produzioni, priva di ϵ-produzioni; supponiamo di averla convertita in CNF.

a) Dimostrate che la grammatica CNF ha $O(n^2)$ produzioni.

b) Dimostrate che la grammatica CNF può avere un numero di produzioni proporzionale a n^2. *Suggerimento*: esaminate la procedura che elimina le produzioni unitarie.

Esercizio 7.1.9 Scrivete le dimostrazioni induttive che completano i seguenti teoremi.

a) La parte del Teorema 7.4 in cui si prova che i simboli rilevati sono effettivamente generatori.

b) Le due direzioni del Teorema 7.6, in cui si dimostra la correttezza dell'algoritmo del Paragrafo 7.1.2 per determinare i simboli raggiungibili.

c) La parte del Teorema 7.11 in cui si dimostra che tutte le coppie rilevate sono effettivamente unitarie.

*! **Esercizio 7.1.10** Per ogni linguaggio libero dal contesto e privo di ϵ, è possibile trovare una grammatica le cui produzioni siano tutte della forma $A \rightarrow BCD$ (corpo fatto di tre variabili) oppure della forma $A \rightarrow a$ (corpo fatto di un solo terminale)? Dimostrate questo enunciato o fornite un controesempio.

Esercizio 7.1.11 In questo esercizio dimostriamo che per ogni linguaggio L libero dal contesto che contenga almeno una stringa diversa da ϵ, c'è una CFG in forma normale di Greibach che genera $L - \{\epsilon\}$. Ricordiamo che in una grammatica in forma normale di Greibach (GNF) il corpo di ogni produzione comincia con un terminale. La procedura si fonda su una serie di lemmi e costruzioni.

a) Supponiamo che una CFG G abbia una produzione $A \to \alpha B \beta$ e che tutte le produzioni per B siano $B \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$. Sostituendo $A \to \alpha B \beta$ con tutte le produzioni ottenute sostituendo a B il corpo di una produzione di B, cioè $A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \cdots \mid \alpha\gamma_n\beta$, otteniamo una grammatica che genera lo stesso linguaggio di G.

D'ora in poi assumiamo che la grammatica G di L sia in forma normale di Chomsky, con variabili $A_1, A_2, \ldots, A_k$.

*! b) Dimostrate che, applicando ripetutamente la trasformazione della parte (a), possiamo convertire G in una grammatica equivalente, in cui ogni corpo di produzione per A_i comincia o con un terminale o con A_j, per un $j \geq i$. In ambedue i casi tutti i simboli dopo il primo in ogni corpo sono variabili.

! c) Sia G_1 la grammatica ottenuta compiendo il passo (b) su G. Sia A_i una variabile e $A_i \to A_i\alpha_1 \mid \cdots \mid A_i\alpha_m$ tutte le sue produzioni i cui corpi cominciano con A_i. Siano

$$A_i \to \beta_1 \mid \cdots \mid \beta_p$$

tutte le altre produzioni di A_i. Notiamo che ogni β_j deve cominciare o con un terminale o con una variabile di indice maggiore di i. Introduciamo una nuova variabile, B_i, e sostituiamo le prime m produzioni con

$$\begin{aligned} A_i &\to \beta_1 B_i \mid \cdots \mid \beta_p B_i \\ B_i &\to \alpha_1 B_i \mid \alpha_1 \mid \cdots \mid \alpha_m B_i \mid \alpha_m \end{aligned}$$

Provate che la grammatica risultante genera lo stesso linguaggio di G e di G_1.

*! d) Sia G_2 la grammatica ottenuta al passo (c). Notiamo che i corpi di tutte le produzioni di A_i cominciano con un terminale o con A_j, per $j > i$. Anche i corpi di tutte le produzioni di B_i cominciano con un terminale o con A_j. Dimostrate che G_2 ha una grammatica equivalente in GNF. *Suggerimento*: sistemate anzitutto le produzioni per A_k, poi per A_{k-1}, e così via, fino ad A_1, applicando il punto (a). Quindi, ancora applicando (a), sistemate le produzioni per B_i, in qualsiasi ordine.

Esercizio 7.1.12 Applicate la costruzione dell'Esercizio 7.1.11 per convertire in GNF la grammatica:

$$S \rightarrow AA \mid 0$$
$$A \rightarrow SS \mid 1$$

7.2 Il pumping lemma per i linguaggi liberi dal contesto

Sviluppiamo ora uno strumento per dimostrare che certi linguaggi non sono liberi dal contesto. Il teorema, detto "pumping lemma per i linguaggi liberi dal contesto", afferma che in ogni stringa sufficientemente lunga di un CFL si possono trovare due sottostringhe, brevi e vicine, che è possibile iterare in tandem. Possiamo cioè ripetere per i volte le due stringhe, per ogni intero i, e generare stringhe che appartengono al linguaggio.

Confrontiamo questo teorema con l'analogo lemma per i linguaggi regolari (Teorema 4.1), secondo il quale c'è sempre una stringa breve da iterare. La differenza si chiarisce esaminando un linguaggio come $L = \{0^n1^n \mid n \geq 1\}$. Possiamo provare che non è regolare fissando n e ripetendo una sottostringa di 0 fino a ottenere una stringa con più 0 che 1. Il lemma per i CFL dice invece che ci sono due sottostringhe: potremmo allora dover scegliere una stringa di 0 e una di 1, generando così solo stringhe in L quando le replichiamo. È una circostanza fortunata perché L è un CFL, e non saremmo quindi in grado di usare il *pumping lemma* dei CFL per costruire stringhe al di fuori di L.

7.2.1 Dimensione degli alberi sintattici

Il primo passo verso un *pumping lemma* per CFL è l'esame della forma e dimensione degli alberi sintattici. La CNF si usa, tra l'altro, per trasformare alberi sintattici in alberi binari. Questi alberi hanno alcune proprietà utili; ne sfrutteremo subito una.

Teorema 7.17 Consideriamo un albero sintattico di una grammatica in forma normale di Chomsky $G = (V, T, P, S)$; sia w, una stringa terminale, il suo prodotto. Se la lunghezza del cammino più lungo è n, abbiamo $|w| \leq 2^{n-1}$.

DIMOSTRAZIONE Per induzione su n.

BASE $n = 1$. Ricordiamo che in un albero la lunghezza di un cammino è il numero di archi, cioè il numero dei nodi meno uno. Perciò un albero con una lunghezza massima dei cammini pari a 1 consiste nella sola radice e in una foglia, etichettata da un terminale. La stringa w è questo terminale, quindi $|w| = 1$. Poiché in questo caso $2^{n-1} = 2^0 = 1$, la base è provata.

INDUZIONE Supponiamo che il cammino più lungo abbia lunghezza n, con $n > 1$. La radice dell'albero usa una produzione, che deve avere la forma $A \rightarrow BC$ perché $n > 1$; non possiamo cioè svolgere l'albero da una produzione con un terminale. Nessun cammino nei sottoalberi radicati in B e C può essere lungo più di $n - 1$ perché questi cammini escludono gli archi dalla radice ai figli etichettati B e C. Quindi, per l'ipotesi

di induzione, i due sottoalberi hanno ciascuno un prodotto di lunghezza al più 2^{n-2}. Il prodotto dell'intero albero è la concatenazione dei loro prodotti, e ha quindi lunghezza non superiore a $2^{n-2} + 2^{n-2} = 2^{n-1}$. Abbiamo così provato il passo induttivo. □

7.2.2 Enunciato del pumping lemma

Il *pumping lemma* per i CFL è molto simile a quello per i linguaggi regolari, anche se ogni stringa z di un CFL L viene scomposta in cinque parti e si replicano la seconda e la quarta in tandem.

Teorema 7.18 (Il *pumping lemma* per linguaggi liberi dal contesto) Sia L un CFL. Esiste una costante n tale che, se z è una stringa in L tale che $|z|$ è almeno n, possiamo scrivere $z = uvwxy$, con le seguenti condizioni.

1. $|vwx| \leq n$. La parte mediana ha lunghezza limitata.

2. $vx \neq \epsilon$. Poiché v e x sono i pezzi da replicare, questa condizione afferma che almeno una di queste stringhe dev'essere non vuota.

3. Per ogni $i \geq 0$, uv^iwx^iy è in L. Le due stringhe v e x possono essere replicate per un numero arbitrario, anche zero, di volte, e le stringhe ottenute sono ancora in L.

DIMOSTRAZIONE Per prima cosa troviamo una grammatica G per L che sia in forma normale di Chomsky. Questo non è tecnicamente possibile se L è il CFL $\emptyset$ o $\{\epsilon\}$. D'altra parte se $L = \emptyset$, l'enunciato del teorema, che si riferisce a una stringa z in L, non può essere violato perché in $\emptyset$ non c'è nessuna stringa. Inoltre la grammatica CNF G genera in realtà $L - \{\epsilon\}$, ma questo è irrilevante perché sceglieremo $n > 0$, nel qual caso z non può comunque essere ϵ.

Fissata ora una grammatica CNF $G = (V, T, P, S)$ tale che $L(G) = L - \{\epsilon\}$, supponiamo che G abbia m variabili. Scegliamo $n = 2^m$. Supponiamo poi che z in L sia lunga almeno n. Per il Teorema 7.17 un albero sintattico il cui cammino più lungo sia lungo m, o meno, ha un prodotto di lunghezza $2^{m-1} = n/2$, o meno. Un tale albero sintattico non può avere z come prodotto perché z è troppo lunga. Ogni albero sintattico con prodotto z ha perciò un cammino di lunghezza almeno pari a $m + 1$.

La Figura 7.5 illustra il cammino più lungo nell'albero per z, dove k è almeno m e il cammino è lungo $k + 1$. Poiché $k \geq m$, lungo il cammino ci sono almeno $m + 1$ occorrenze di variabili $A_0, A_1, \ldots, A_k$. Ma in V ci sono solo m variabili distinte, quindi almeno due delle ultime $m + 1$ variabili del cammino (da A_{k-m} ad A_k incluse) devono essere uguali. Supponiamo $A_i = A_j$, con $k - m \leq i < j \leq k$.

Possiamo allora dividere l'albero come illustrato nella Figura 7.6. La stringa w è il prodotto del sottoalbero radicato in A_j. Le stringhe v e x sono quelle a sinistra e a destra di w nel prodotto del sottoalbero più grande, radicato in A_i. Si osservi che, non essendoci

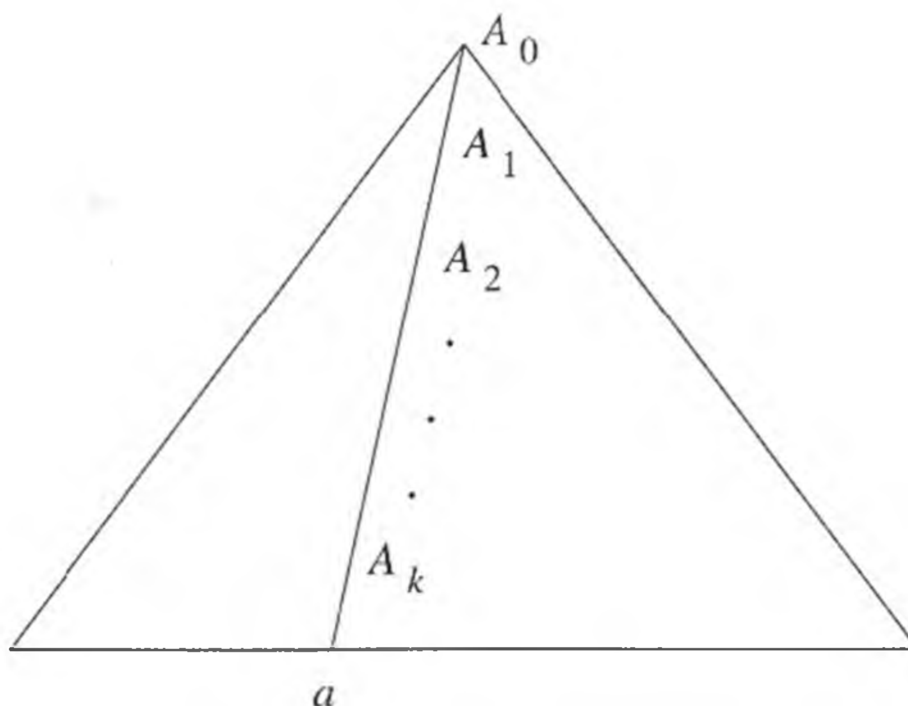


Figura 7.5 Una stringa sufficientemente lunga in L deve avere un cammino "lungo" nel suo albero sintattico.

produzioni unitarie, v e x non possono essere entrambe ϵ. Infine u e y sono le parti di z a sinistra e a destra del sottoalbero radicato in A_i.

Se $A_i = A_j = A$, possiamo costruire nuovi alberi sintattici a partire da quello originale, come illustrato nella Figura 7.7(a). Possiamo anzitutto sostituire il sottoalbero radicato in A_i, che ha prodotto vwx, con quello radicato in A_j, che ha prodotto w. Possiamo farlo perché ambedue gli alberi hanno la radice etichettata A. L'albero risultante, illustrato nella Figura 7.7(b), ha come prodotto la stringa uwy e corrisponde al caso $i = 0$ nello schema uv^iwx^iy.

La Figura 7.7(c) suggerisce un'altra possibilità. Qui il sottoalbero radicato in A_j è stato sostituito dall'intero sottoalbero radicato in A_i. L'operazione è valida per lo stesso motivo di prima: sostituiamo un albero con radice etichettata A con un altro avente la stessa caratteristica. Il prodotto di questo albero è uv^2wx^2y. Se ora sostituissimo il sottoalbero della Figura 7.7(c) che ha come prodotto w con il sottoalbero più grande, di prodotto vwx, otterremmo un albero di prodotto uv^3wx^3y. Potremmo continuare così per ogni esponente i. In G ci sono quindi alberi sintattici per tutte le stringhe di forma uv^iwx^iy, e il lemma è quasi dimostrato.

L'ultimo dettaglio è la condizione (1), che dice $|vwx| \leq n$. Poiché abbiamo scelto A_i vicino alla base dell'albero ($k - i \leq m$), il cammino più lungo nel sottoalbero radicato in A_i non può essere più lungo di $m + 1$. Per il Teorema 7.17 la lunghezza del prodotto del sottoalbero radicato in A_i non può superare $2^m = n$. $\square$

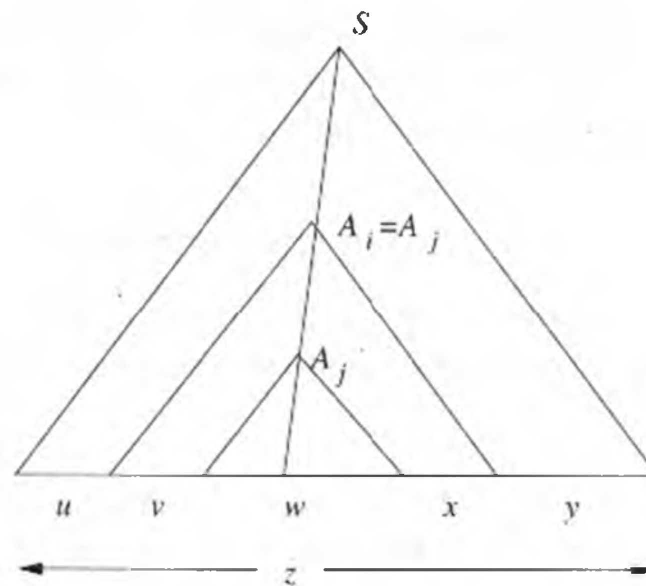


Figura 7.6 Scomposizione della stringa w.

7.2.3 Applicazioni del pumping lemma per i CFL

Come abbiamo fatto per i linguaggi regolari, ci serviamo del *pumping lemma* per i CFL come di un gioco a due.

1. Scegliamo un linguaggio L che vogliamo dimostrare non essere un CFL.

2. L'avversario sceglie n senza comunicarcelo: dovremo agire per ogni possibile n.

3. Scegliamo z; per farlo possiamo usare n come parametro.

4. L'avversario scompone z in $uvwxy$, con il solo vincolo che $|vwx| \leq n$ e $vx \neq \epsilon$.

5. Vinciamo la partita se possiamo scegliere i in modo che uv^iwx^iy non sia in L.

Vediamo ora alcuni esempi di linguaggi che dimostriamo non liberi dal contesto per mezzo del *pumping lemma*. Il primo esempio mostra che i linguaggi liberi dal contesto, pur potendo confrontare due gruppi di simboli secondo criteri di uguaglianza, non possono farlo per tre gruppi.

Esempio 7.19 Sia L il linguaggio $\{0^n1^n2^n \mid n \geq 1\}$. L consiste di tutte le stringhe in $\mathbf{0^+1^+2^+}$ con lo stesso numero di occorrenze di ciascun simbolo, per esempio 012, 001122, e così via. Supponiamo che L sia libero dal contesto. Il *pumping lemma* fornisce allora un intero n.[2] Scegliamo $z = 0^n1^n2^n$.

[2]Precisiamo che n è la costante fornita dal lemma e non ha nulla a che vedere con la variabile n che compare nella definizione di L.

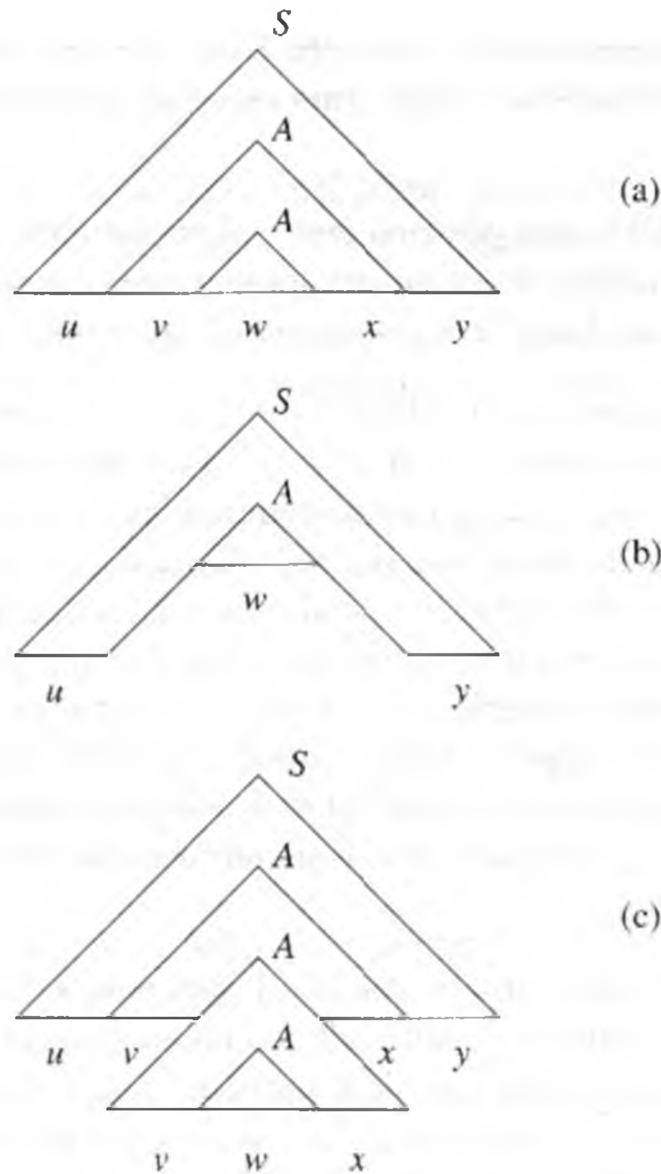


Figura 7.7 Le stringhe v e x, replicate prima zero, poi due volte.

Supponiamo che l'avversario spezzi z in $z = uvwxy$, dove $|vwx| \leq n$, e v e x non sono entrambe ϵ. Sappiamo che vwx non può contenere sia 0 sia 2 perché l'ultimo 0 e il primo 2 sono separati da $n + 1$ posizioni. Dimostreremo che L contiene stringhe al di fuori di L, contraddicendo l'assunto che L sia un CFL. Ci sono due casi.

1. vwx non contiene 2. In questo caso vx consiste solo di 0 e 1, e conta almeno un simbolo. Quindi uwy, che dovrebbe essere in L per il *pumping lemma*, ha n 2, ma ha meno di n 0 o meno di n 1, o entrambi. Non appartiene dunque a L. Ne deduciamo che in questo caso L non è un CFL.

2. vwx non contiene 0. Analogamente al primo caso, uwy ha n 0 e meno di n 1 o meno di n 2. Quindi non è in L.

In ambedue i casi concludiamo che L dovrebbe contenere una stringa che sappiamo non appartenergli. Dalla contraddizione ricaviamo che l'assunto era scorretto: L *non* è un CFL. □

Un'altra cosa che i CFL non possono fare è abbinare coppie con lo stesso numero di simboli, se le coppie sono intrecciate fra loro. L'idea viene precisata nell'esempio seguente, in cui si prova mediante il *pumping lemma* che un linguaggio non è un CFL.

Esempio 7.20 Sia L il linguaggio $\{0^i1^j2^i3^j \mid i \geq 1 \text{ e } j \geq 1\}$. Se L è libero dal contesto, sia n la costante per L; scegliamo $z = 0^n1^n2^n3^n$. Possiamo scrivere $z = uvwxy$ con i vincoli abituali: $|vwx| \leq n$ e $vx \neq \epsilon$. Di conseguenza vwx è contenuta nella sottostringa di un solo simbolo o sta a cavallo di due simboli adiacenti.

Se vwx consiste di un solo simbolo ripetuto, uwy ha n occorrenze di ognuno dei tre simboli distinti e meno di n del quarto simbolo. Perciò non può essere in L. Se vwx è a cavallo di due simboli, per esempio 1 e 2, a uwy mancano degli 1 o dei 2, o entrambi. Supponiamo che manchino degli 1. Poiché ci sono n 3 la stringa non può essere in L. Analogamente, se mancano dei 2, essendoci n 0, uwy non può essere in L. Abbiamo allora una contraddizione con l'ipotesi che L sia un CFL, da cui deduciamo che non lo è. □

Come ultimo esempio dimostriamo che i CFL non possono abbinare due stringhe di lunghezza arbitraria, se le stringhe sono scelte da un alfabeto di più di un simbolo. Per inciso, questa osservazione significa che le grammatiche non sono adatte a imporre in un linguaggio di programmazione certi vincoli "semantici", come il consueto vincolo che un identificatore dev'essere dichiarato prima dell'uso. Per registrare gli identificatori dichiarati si impiega nella pratica un altro meccanismo, per esempio una *tabella dei simboli*; non cercheremo quindi di ideare un parser che di per sé verifichi l'aderenza alla regola citata.

Esempio 7.21 Sia $L = \{ww \mid w \text{ è in } \{0,1\}^*\}$. L consiste di stringhe ripetute, come ϵ, 0101, 00100010 o 110110. Supponiamo L libero dal contesto, e sia n la costante del *pumping lemma*. Consideriamo la stringa $z = 0^n1^n0^n1^n$. La stringa è la ripetizione di 0^n1^n, ed è quindi in L.

Prendendo spunto dagli esempi precedenti scomponiamo $z = uvwxy$ in modo che $|vwx| \leq n$ e $vx \neq \epsilon$. Proveremo che uwy non è in L, dimostrando così per assurdo che L non è libero dal contesto.

Osserviamo per prima cosa che da $|vwx| \leq n$ segue $|uwy| \geq 3n$. Perciò se uwy è una stringa ripetuta, come tt, la lunghezza di t è almeno $3n/2$. A seconda della posizione di vwx in z dobbiamo considerare diversi casi.

1. Supponiamo che vwx sia nei primi n 0. In particolare sia vx fatto di k 0, con $k > 0$. Allora uwy comincia con $0^{n-k}1^n$. Poiché $|uwy| = 4n - k$, sappiamo che

se $uwy = tt$ si ha $|t| = 2n - k/2$. Pertanto t termina dopo il primo blocco di 1, e quindi termina con uno 0. Ma uwy termina in 1, e non può essere uguale a tt.

2. Supponiamo che vwx sia a cavallo del primo blocco di 0 e il primo di 1. Può darsi che vx consista solo di 0, se $x = \epsilon$. Allora il ragionamento secondo cui uwy non è della forma tt è lo stesso del caso (1). Se vx ha almeno un 1, notiamo che t, la cui lunghezza è almeno $3n/2$, deve terminare per 1^n perché così termina uwy. In ogni caso non ci sono altri blocchi di n 1 oltre al blocco finale, quindi t non può ripetersi in uwy.

3. Se vwx è contenuta nel primo blocco di 1, la prova che uwy non è in L è la stessa della seconda parte del caso (2).

4. Supponiamo che vwx sia a cavallo del primo blocco di 1 e del secondo di 0. Se vx non contiene 0, si ragiona come nel caso in cui vwx è contenuta nel primo blocco di 1. Se vx ha almeno uno 0, uwy comincia con un blocco di n 0, come t, se $uwy = tt$. In ogni caso non ci sono altri blocchi di n 0 in uwy per la seconda copia di t. Ancora una volta concludiamo che uwy non è in L.

5. Negli altri casi, in cui vwx è nella seconda metà di z, il ragionamento è simmetrico ai casi in cui si trova nella prima metà.

Perciò in nessun caso uwy è in L; concludiamo che L non è libero dal contesto. □

7.2.4 Esercizi

Esercizio 7.2.1 Applicando il *pumping lemma* per i CFL dimostrate che i seguenti linguaggi non sono liberi dal contesto.

* a) $\{a^i b^j c^k \mid i < j < k\}$.

b) $\{a^n b^n c^i \mid i \leq n\}$.

c) $\{0^p \mid p$ è un numero primo$\}$. *Suggerimento*: adattate le idee impiegate nell'Esempio 4.3, dove si dimostra che questo linguaggio non è regolare.

*! d) $\{0^i 1^j \mid j = i^2\}$.

! e) $\{a^n b^n c^i \mid n \leq i \leq 2n\}$.

! f) $\{ww^R w \mid w$ è una stringa di 0 e 1$\}$, cioè l'insieme delle stringhe formate da una stringa w seguita dalla stessa stringa rovesciata, e ancora da w, come 001100001.

! **Esercizio 7.2.2** Quando si cerca di applicare il *pumping lemma* a un CFL, "vince l'avversario" e non si riesce a completare la prova. Spiegate dove la dimostrazione fallisce per i seguenti linguaggi.

a) $\{00, 11\}$.

* b) $\{0^n 1^n \mid n \geq 1\}$.

* c) L'insieme delle palindrome sull'alfabeto $\{0, 1\}$.

! **Esercizio 7.2.3** Una versione più forte del *pumping lemma* per i CFL, nota come *lemma di Ogden*, differisce dal primo perché si concentra su n posizioni "privilegiate" di una stringa z e garantisce che le stringhe da replicare hanno tra 1 ed n posizioni privilegiate. Il vantaggio è che un linguaggio può avere stringhe fatte di due parti, di cui una può essere replicata senza generare stringhe estranee al linguaggio, mentre l'altra, se replicata, ne genera. Se non possiamo imporre che la replicazione avvenga nella seconda parte, non siamo in grado di portare a termine una dimostrazione di non libertà dal contesto. Ecco l'enunciato formale del lemma di Ogden: sia L un CFL; esiste una costante n tale che se z è una stringa in L, di lunghezza non inferiore a n, in cui scegliamo almeno n posizioni *privilegiate*, possiamo scrivere $z = uvwxy$ con le seguenti condizioni:

1. vwx ha al massimo n posizioni privilegiate
2. vx ha almeno una posizione privilegiata
3. per ogni i, uv^iwx^iy è in L.

Dimostrate il lemma di Ogden. *Suggerimento*: la dimostrazione coincide con quella del Teorema 7.18, se fingiamo che le posizioni non privilegiate di z siano assenti quando scegliamo un cammino "lungo" nell'albero sintattico di z.

* **Esercizio 7.2.4** Applicate il lemma di Ogden (Esercizio 7.2.3) per semplificare la dimostrazione dell'Esempio 7.21 che $L = \{ww \mid w \text{ è in } \{0, 1\}^*\}$ non è un CFL. *Suggerimento*: con $z = 0^n 1^n 0^n 1^n$ dichiarate privilegiati i due blocchi mediani.

Esercizio 7.2.5 Per mezzo del lemma di Ogden (Esercizio 7.2.3) dimostrate che i seguenti linguaggi non sono CFL.

! a) $\{0^i 1^j 0^k \mid j = \max(i, k)\}$.

!! b) $\{a^n b^n c^i \mid i \neq n\}$. *Suggerimento*: se n è la costante del lemma di Ogden, considerate la stringa $z = a^n b^n c^{n!}$.

7.3 Proprietà di chiusura dei linguaggi liberi dal contesto

Consideriamo ora alcune operazioni sui linguaggi liberi dal contesto che producono nuovi CFL. Molte di queste proprietà di chiusura sono analoghe ai teoremi sui linguaggi regolari del Paragrafo 4.2, con alcune differenze.

Presentiamo per prima un'operazione in cui sostituiamo ogni simbolo nelle stringhe di un linguaggio con un intero linguaggio. Questa operazione, che generalizza l'omomorfismo studiato nel Paragrafo 4.2.3, è utile per dimostrare altre proprietà di chiusura dei CFL, tra cui quelle relative alle operazioni su espressioni regolari: unione, concatenazione e chiusura. Dimostriamo che i CFL sono chiusi per omomorfismo e omomorfismo inverso. Diversamente dai linguaggi regolari, i CFL non sono chiusi per intersezione e differenza, ma l'intersezione e la differenza di un CFL e di un linguaggio regolare sono sempre CFL.

7.3.1 Sostituzioni

Sia Σ un alfabeto; per ogni simbolo a in Σ scegliamo un linguaggio L_a. Gli alfabeti per questi linguaggi sono arbitrari, ovvero possono essere diversi tra di loro così come distinti da Σ. La scelta definisce una funzione s (una *sostituzione*) su Σ; per ogni simbolo a denoteremo L_a con $s(a)$.

Se $w = a_1a_2\cdots a_n$ è una stringa in Σ^*, $s(w)$ è il linguaggio di tutte le stringhe $x_1x_2\cdots x_n$ tali che la stringa x_i appartiene al linguaggio $s(a_i)$, per $i = 1, 2, \ldots, n$. In altri termini $s(w)$ è la concatenazione dei linguaggi $s(a_1)s(a_2)\cdots s(a_n)$. Possiamo estendere ai linguaggi la definizione di s: $s(L)$ è l'unione degli $s(w)$ per tutte le stringhe w in L.

Esempio 7.22 Siano $s(0) = \{a^nb^n \mid n \geq 1\}$ e $s(1) = \{aa, bb\}$. Quindi s è una sostituzione sull'alfabeto $\Sigma = \{0, 1\}$. Il linguaggio $s(0)$ è l'insieme delle stringhe con uno o più a seguiti da un uguale numero di b, mentre $s(1)$ è il linguaggio, finito, formato dalle due stringhe aa e bb.

Sia $w = 01$. Allora $s(w)$ è la concatenazione dei linguaggi $s(0)s(1)$. Per la precisione $s(w)$ consiste nelle stringhe della forma a^nb^naa e a^nb^{n+2}, con $n \geq 1$.

Poniamo ora $L = L(\mathbf{0}^*)$, cioè l'insieme di tutte le stringhe di 0. Allora $s(L) = \left(s(0)\right)^*$. Questo linguaggio è l'insieme di tutte le stringhe della forma

$$a^{n_1}b^{n_1}a^{n_2}b^{n_2}\cdots a^{n_k}b^{n_k}$$

per qualche $k \geq 0$ e per qualsiasi sequenza di interi positivi $n_1, n_2, \ldots, n_k$. Il linguaggio comprende fra le sue stringhe ϵ, $aabbaaabbb$ e $abaabbabab$. □

Teorema 7.23 Se L è un linguaggio libero dal contesto sull'alfabeto Σ, ed s è una sostituzione su Σ tale che $s(a)$ è un CFL per ogni a in Σ, allora $s(L)$ è un CFL.

DIMOSTRAZIONE L'idea di fondo è questa: prendiamo una CFG per L e sostituiamo ogni terminale a con il simbolo iniziale di una CFG per il linguaggio $s(a)$. Otteniamo una CFG che genera $s(L)$. Dobbiamo ora precisare alcuni dettagli.

In termini formali partiamo da una grammatica per ognuno dei linguaggi in questione: poniamo quindi $G = (V, \Sigma, P, S)$ per L e $G_a = (V_a, T_a, P_a, S_a)$ per ogni a in Σ. Poiché la scelta dei nomi delle variabili è libera, facciamo in modo che i vari insiemi di variabili siano disgiunti, cioè che nessun A compaia in più di uno degli insiemi V e V_a. Questa scelta ha lo scopo di garantire che, unendo le produzioni delle diverse grammatiche in un insieme, non vengano accidentalmente mescolate produzioni da grammatiche distinte, con l'effetto di generare derivazioni che non appartengono a nessuna delle grammatiche assegnate.

Costruiamo ora una nuova grammatica $G' = (V', T', P', S)$ per $s(L)$.

- V' è l'unione di V e di tutti i V_a, per a in Σ.
- T' è l'unione di tutti i T_a, per a in Σ.
- P' contiene:
    1. tutte le produzioni in ogni P_a, per a in Σ
    2. le produzioni di P, nei cui corpi ogni terminale a è sostituito da S_a.

In questo modo gli alberi sintattici di G' si sviluppano come quelli di G, ma anziché avere un prodotto in Σ^* hanno una frontiera i cui nodi sono etichettati da S_a per un a in Σ. Da ognuno di questi nodi pende un albero sintattico di G_a, il cui prodotto è una stringa terminale nel linguaggio $s(a)$. Un tipico albero sintattico è illustrato nella Figura 7.8.

Dobbiamo ora dimostrare formalmente che la costruzione è corretta, cioè che G' genera il linguaggio $s(L)$.

- Una stringa w è in $L(G')$ se e solo se w è in $s(L)$.

(Se) Sia w in $s(L)$. Allora esistono una stringa $x = a_1a_2 \cdots a_n$ in L e stringhe x_i in $s(a_i)$, per $i = 1, 2, \ldots, n$, tali che $w = x_1x_2 \cdots x_n$. Quindi la porzione di G' proveniente dalle produzioni di G, con S_a al posto di ogni a, genera una stringa pari a x con S_a al posto di ogni a. Questa stringa è $S_{a_1}S_{a_2} \cdots S_{a_n}$. Questa parte della derivazione di w corrisponde al triangolo superiore nella Figura 7.8.

Poiché le produzioni di ogni G_a sono anche produzioni di G', la derivazione di x_i da S_{a_i} è anche una derivazione in G'. Gli alberi sintattici di queste derivazioni corrispondono ai triangoli inferiori della figura. Dato che il prodotto dell'albero sintattico di G' è $x_1x_2 \cdots x_n = w$, concludiamo che w è in $L(G')$.

(Solo se) Sia ora w in $L(G')$. Affermiamo che l'albero sintattico di w deve avere la forma dell'albero della Figura 7.8. Il motivo è che le variabili delle grammatiche G e G_a, per a

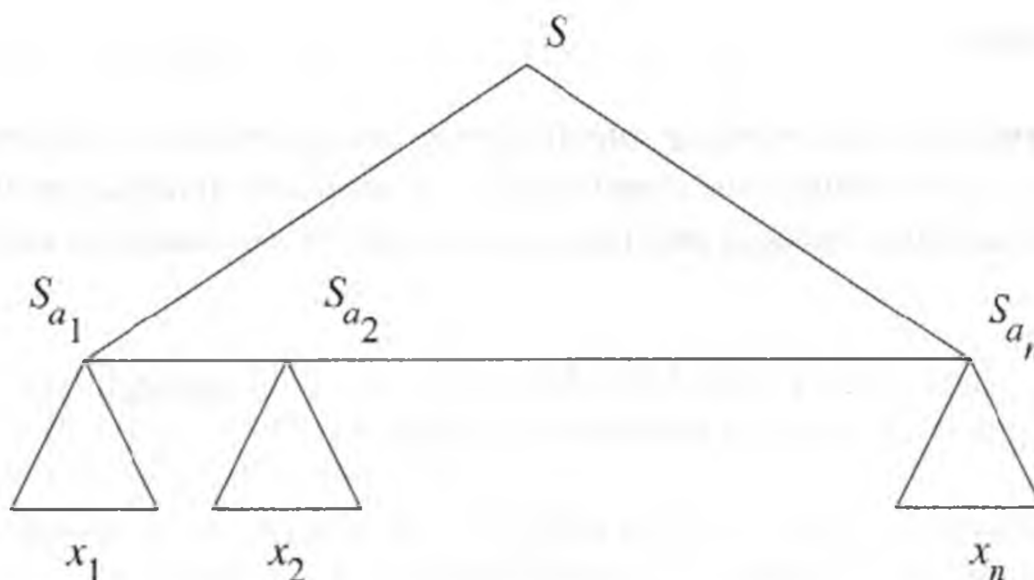


Figura 7.8 Un albero sintattico di G' si sviluppa come uno di G, ma termina in più alberi, ciascuno appartenente a una delle grammatiche G_a.

in Σ, sono disgiunte. Perciò in vetta all'albero, a partire da S, si applicano solo produzioni di G fino a derivare simboli S_a, al di sotto dei quali si applicano solo produzioni di G_a. Di conseguenza, se w ha T come albero sintattico, possiamo determinare una stringa $a_1a_2 \cdots a_n$ in $L(G)$ e stringhe x_i nel linguaggio $s(a_i)$ tali che:

1. $w = x_1x_2 \cdots x_n$

2. la stringa $S_{a_1}S_{a_2} \cdots S_{a_n}$ è il prodotto di un albero formato da T eliminando alcuni sottoalberi (come suggerito nella Figura 7.8).

Ma la stringa $x_1x_2 \cdots x_n$ è in $s(L)$, essendo formata dalla sostituzione di ogni a_i con x_i. Concludiamo che w è in $s(L)$. □

7.3.2 Applicazioni del teorema di sostituzione

Grazie al Teorema 7.23 possiamo dimostrare anche per i CFL numerose proprietà di chiusura note che abbiamo studiato nel caso dei linguaggi regolari.

Teorema 7.24 I linguaggi liberi dal contesto sono chiusi rispetto alle seguenti operazioni:

1. unione

2. concatenazione

3. chiusura (*) e chiusura positiva ($^+$)

4. omomorfismo.

DIMOSTRAZIONE Ciascuna richiede solo di definire un'opportuna sostituzione. Le dimostrazioni che seguono comportano l'operazione di sostituzione di linguaggi liberi dal contesto applicata ad altri linguaggi liberi dal contesto; per il Teorema 7.23 esse producono CFL.

1. *Unione*: siano L_1 ed L_2 due CFL. Allora $L_1 \cup L_2$ è il linguaggio $s(L)$, dove L è il linguaggio $\{1, 2\}$ ed s è la sostituzione definita da $s(1) = L_1$ e $s(2) = L_2$.

2. *Concatenazione*: siano L_1 ed L_2 due CFL. Allora L_1L_2 è il linguaggio $s(L)$, dove L è il linguaggio $\{12\}$ ed s è la stessa sostituzione del caso (1).

3. *Chiusura e chiusura positiva*: siano L_1 un CFL, L il linguaggio $\{1\}^*$ ed s la sostituzione $s(1) = L_1$; allora $L_1^* = s(L)$. Analogamente, se L è invece il linguaggio $\{1\}^+$, allora $L_1^+ = s(L)$.

4. Sia L un CFL sull'alfabeto Σ e sia h un omomorfismo su Σ. Sia s la sostituzione che rimpiazza ogni simbolo a in Σ con il linguaggio formato dalla sola stringa $h(a)$. Cioè $s(a) = \{h(a)\}$ per ogni a in Σ. Allora $h(L) = s(L)$.

□

7.3.3 Inversione

I CFL sono chiusi anche rispetto all'operazione di inversione. Non possiamo servirci del teorema di sostituzione, ma c'è una semplice costruzione basata su grammatiche.

Teorema 7.25 Se L è un CFL, allora lo è anche L^R.

DIMOSTRAZIONE Sia $L = L(G)$ per un CFL $G = (V, T, P, S)$. Costruiamo $G^R = (V, T, P^R, S)$, dove P^R è l'insieme degli "inversi" delle produzioni in P: se $A \to \alpha$ è una produzione di G, $A \to \alpha^R$ è una produzione di G^R. Una semplice induzione sulla lunghezza delle derivazioni in G e G^R dimostra che $L(G^R) = L^R$. In sostanza tutte le forme sentenziali di G^R sono inversi di forme sentenziali di G, e viceversa. Ne lasciamo la dimostrazione formale come esercizio. □

7.3.4 Intersezione con un linguaggio regolare

I CFL non sono chiusi rispetto all'operazione di intersezione. Lo prova un semplice esempio.

Esempio 7.26 Nell'Esempio 7.19 abbiamo visto che il linguaggio

$$L = \{0^n 1^n 2^n \mid n \geq 1\}$$

non è libero dal contesto. D'altra parte i due linguaggi che seguono lo sono.

$$L_1 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$$
$$L_2 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$$

Una grammatica per L_1 è ad esempio la seguente:

$$S \to AB$$
$$A \to 0A1 \mid 01$$
$$B \to 2B \mid 2$$

In questa grammatica, A genera tutte le stringhe della forma $0^n 1^n$, B tutte le stringhe di 2. Una grammatica per L_2 è data da:

$$S \to AB$$
$$A \to 0A \mid 0$$
$$B \to 1B2 \mid 12$$

Il meccanismo è simile, ma A genera tutte le stringhe di 0, mentre B le stringhe della forma $1^n 2^n$.

Si nota immediatamente che $L = L_1 \cap L_2$. Infatti L_1 impone che ci sia lo stesso numero di 0 e di 1 ed L_2 che ci sia lo stesso numero di 1 e di 2. Una stringa in ambedue i linguaggi deve avere lo stesso numero di ripetizioni dei tre simboli, e quindi appartenere a L.

Se i CFL fossero chiusi rispetto all'intersezione, potremmo dimostrare l'enunciato, falso, che L è libero dal contesto. Abbiamo così dimostrato per assurdo che i CFL non sono chiusi rispetto all'intersezione. □

Per quanto riguarda l'intersezione, possiamo affermare una proprietà più debole: i linguaggi liberi dal contesto sono chiusi rispetto all'intersezione con un linguaggio regolare. L'enunciato formale e la dimostrazione costituiscono il seguente teorema.

Teorema 7.27 Se L è un CFL ed R è un linguaggio regolare, $L \cap R$ è un CFL.

DIMOSTRAZIONE Ci serviamo delle rappresentazioni dei CFL mediante automi a pila e dei linguaggi regolari mediante automi a stati finiti. La dimostrazione generalizza quella del Teorema 4.8, dove si eseguono "in parallelo" due automi per avere l'intersezione dei loro linguaggi. Qui si esegue un automa a stati finiti in parallelo con un PDA; il risultato è un nuovo PDA, come illustrato nella Figura 7.9.

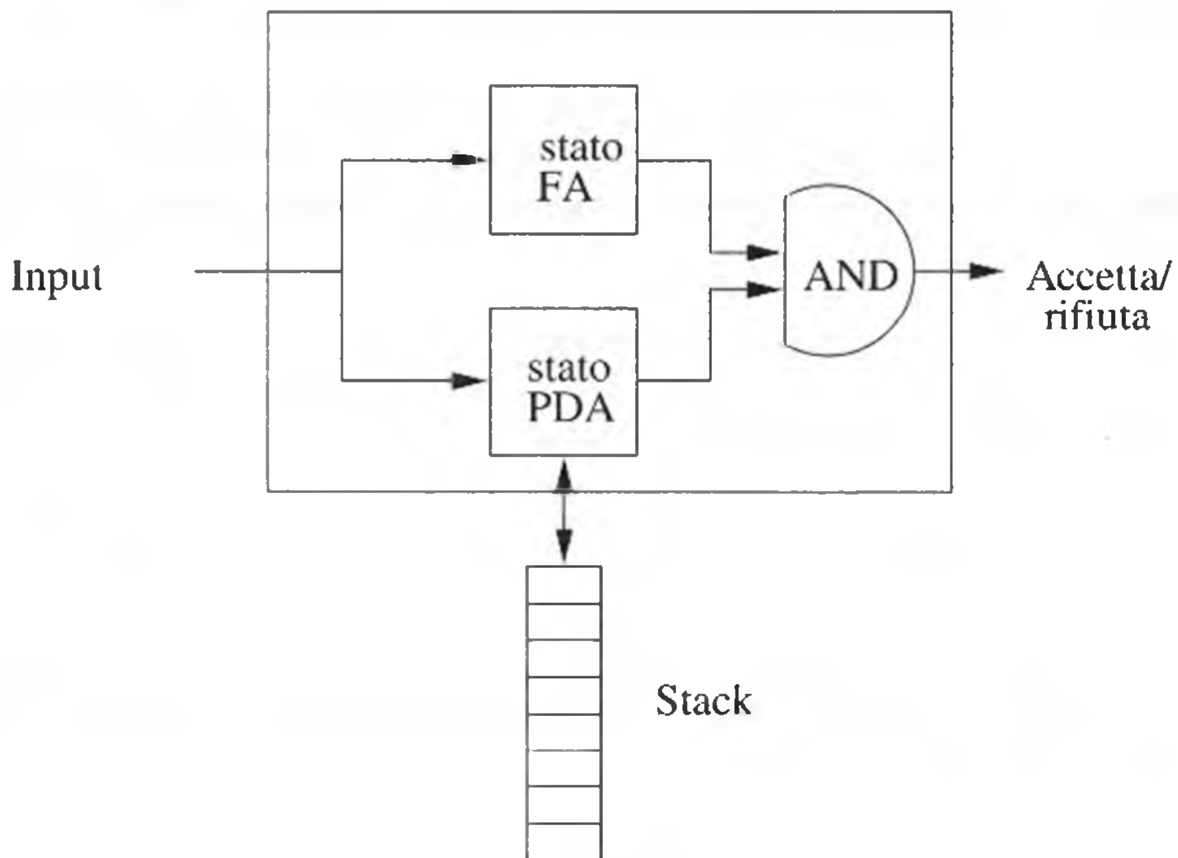


Figura 7.9 Un PDA e un FA eseguiti in parallelo formano un nuovo PDA.

Formalmente sia

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

un PDA che accetta L per stato finale, e sia

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

un DFA per R. Definiamo il PDA

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$$

dove $\delta\big((q, p), a, X\big)$ è per definizione l'insieme delle coppie $\big((r, s), \gamma\big)$ tali che:

1. $s = \hat{\delta}_A(p, a)$
2. la coppia (r, γ) appartiene a $\delta_P(q, a, X)$.

Ogni mossa del PDA P può essere compiuta anche in P'; inoltre un secondo componente dello stato di P' tiene traccia dello stato del DFA A. Notiamo che a può essere un simbolo di Σ o ϵ. Nel primo caso $\hat{\delta}(p, a) = \delta_A(p)$, mentre se $a = \epsilon$, allora $\hat{\delta}(p, a) = p$; dunque A non cambia stato mentre P muove su input ϵ.

Una semplice induzione sul numero di mosse fatte dal PDA dimostra che $(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \gamma)$ se e solo se $((q_P, q_A), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((q, p), \epsilon, \gamma)$, dove $p = \hat{\delta}(p_A, w)$. Lasciamo le induzioni per esercizio. Poiché (q, p) è uno stato accettante di P' se e solo se q è uno stato accettante di P e p è accettante in A, concludiamo che P' accetta w se e solo se sia P sia A accettano, cioè se w è in $L \cap R$. □

Esempio 7.28 Il PDA F rappresentato nella Figura 6.6 accetta per stato finale l'insieme delle stringhe di i ed e che corrispondono a violazioni minimali della regola sull'ordine di **if** ed **else** nei programmi C. Chiamiamo L questo linguaggio. Il PDA F è definito da

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

dove δ_F è così specificata:

1. $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$
3. $\delta_F(q, e, Z) = \{(q, \epsilon)\}$
4. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon, \epsilon)\}$.

Definiamo ora un automa a stati finiti

$$A = (\{s, t\}, \{i, e\}, \delta_A, s, \{s, t\})$$

che accetta le stringhe del linguaggio $\mathbf{i}^*\mathbf{e}^*$, cioè tutte le stringhe fatte di i seguiti da e, in numero arbitrario. Chiamiamo R questo linguaggio. La funzione di transizione δ_A è così definita:

a) $\delta_A(s, i) = s$

b) $\delta_A(s, e) = t$

c) $\delta_A(t, e) = t$.

A rigor di termini, A non è un DFA, come richiesto dal Teorema 7.27, perché gli manca uno stato trappola di destinazione per l'input i nello stato t. La stessa costruzione funziona però anche per un NFA perché il PDA da costruire può essere non deterministico. In questo caso il PDA costruito è di fatto deterministico, anche se "muore" su certe sequenze di input.

Definiamo un PDA

$$P = (\{p, q, r\} \times \{s, t\}, \{i, e\}, \{Z, X_0\}, \delta, (p, s), X_0, \{r\} \times \{s, t\})$$

Le transizioni di δ sono elencate sotto, contrassegnate dalla regola del PDA F (un numero da 1 a 4) e dalla regola del DFA A (una lettera, a, b o c) da cui derivano. Le ϵ-transizioni del PDA F non derivano da regole di A. Notiamo che le regole sono formate secondo necessità a partire dallo stato di P composto dagli stati iniziali di F e A, e costruendo le regole per gli altri stati solo quando si scopre che P li può raggiungere.

1: $\delta\big((p,s),\epsilon,X_0\big) = \big\{\big((q,s),ZX_0\big)\big\}$.

2a: $\delta\big((q,s),i,Z\big) = \big\{\big((q,s),ZZ\big)\big\}$.

3b: $\delta\big((q,s),e,Z\big) = \big\{\big((q,t),\epsilon\big)\big\}$.

4: $\delta\big((q,s),\epsilon,X_0\big) = \big\{\big((r,s),\epsilon\big)\big\}$. Nota: possiamo provare che questa regola non si applica mai perché è impossibile togliere dallo stack senza aver visto un e, e quando P vede un e, il secondo componente del suo stato diventa t.

3c: $\delta\big((q,t),e,Z\big) = \big\{\big((q,t),\epsilon\big)\big\}$.

4: $\delta\big((q,t),\epsilon,X_0\big) = \big\{\big((r,t),\epsilon\big)\big\}$.

Il linguaggio $L \cap R$ è l'insieme delle stringhe composte da un certo numero di i seguiti da un e in più, cioè $\{i^n e^{n+1} \mid n \geq 0\}$. Si tratta proprio dell'insieme di violazioni fatte di un blocco di **if** seguito da un blocco di **else**. Il linguaggio è evidentemente un CFL, generato dalla grammatica con le produzioni $S \rightarrow iSe \mid e$.

Osserviamo che il PDA P accetta il linguaggio $L \cap R$. Dopo aver inserito uno Z nello stack, ne inserisce altri in risposta agli input i, rimanendo nello stato (q,s). Quando vede un e, va nello stato (q,t) e comincia a svuotare lo stack. L'automa muore se vede un i; quando X_0 compare in cima allo stack compie una transizione spontanea allo stato (r,t) e accetta. □

Dal fatto che i CFL non sono chiusi rispetto all'intersezione, ma lo sono rispetto all'intersezione con un linguaggio regolare, ricaviamo le proprietà relative alle operazioni di differenza e complemento, compendiate nel teorema seguente.

Teorema 7.29 Siano L, L_1 ed L_2 tre CFL, ed R un linguaggio regolare.

1. $L - R$ è un linguaggio libero dal contesto.

2. $\overline{L}$ può non essere libero dal contesto.

3. $L_1 - L_2$ può non essere libero dal contesto.

DIMOSTRAZIONE Per (1) notiamo che $L - R = L \cap \overline{R}$. Se R è regolare, per il Teorema 4.5 anche $\overline{R}$ è regolare. Quindi, per il Teorema 7.27, $L - R$ è un CFL.

Per (2) supponiamo che $\overline{L}$ sia libero dal contesto quando lo è L. Poiché

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

e i CFL sono chiusi rispetto all'unione, ne seguirebbe che sono chiusi anche rispetto all'intersezione. Dall'Esempio 7.26 sappiamo però che non è vero.

Infine dimostriamo (3). Sappiamo che Σ^* è un CFL per ogni alfabeto Σ: è facile definire una grammatica o un PDA per questo linguaggio regolare. Perciò se $L_1 - L_2$ fosse un CFL quando lo sono L_1 ed L_2, ne seguirebbe che $\Sigma^* - L$ è un CFL se lo è L. Ma, per un'opportuna scelta dell'alfabeto Σ, $\Sigma^* - L$ è $\overline{L}$. Ciò sarebbe in contraddizione con (2): abbiamo quindi dimostrato per assurdo che $L_1 - L_2$ non è necessariamente un CFL. □

7.3.5 Omomorfismo inverso

Riprendiamo dal Paragrafo 4.2.4 l'operazione detta "omomorfismo inverso". Se h è un omomorfismo ed L un linguaggio, $h^{-1}(L)$ è l'insieme delle stringhe w tali che $h(w)$ è in L. La dimostrazione che i linguaggi regolari sono chiusi per omomorfismo inverso è indicata nella Figura 4.6, dove si illustra come definire un automa a stati finiti che elabora un simbolo di input a applicandogli un omomorfismo h e simulando un altro automa a stati finiti sulla sequenza $h(a)$.

La stessa proprietà di chiusura, ma per i CFL, si può dimostrare in modo molto simile impiegando i PDA in luogo degli automi a stati finiti. I PDA presentano però un problema che gli automi a stati finiti non hanno. L'azione di un automa a stati finiti su una sequenza di input è una transizione di stato, e quindi, rispetto al nuovo automa, è affine a una mossa singola.

Se l'automa è un PDA, invece, una sequenza di mosse può non equivalere a una mossa su un solo simbolo di input. In particolare il PDA può, in n mosse, eliminare n simboli dallo stack, mentre in una mossa ne può eliminare uno solo. La costruzione per i PDA analoga alla Figura 4.6 è perciò un po' più complessa ed è illustrata nella Figura 7.10. L'adattamento consiste nel collocare $h(a)$ in un "buffer" dopo aver letto l'input a. I simboli di $h(a)$ vengono passati, uno alla volta, al PDA da simulare. Il nuovo PDA legge un altro simbolo di input e gli applica l'omomorfismo solo quando il buffer è vuoto. Nel prossimo teorema formalizziamo questa costruzione.

Teorema 7.30 Sia L un CFL e h un omomorfismo. Allora $h^{-1}(L)$ è un CFL.

DIMOSTRAZIONE Supponiamo che h si applichi ai simboli dell'alfabeto Σ e produca stringhe in T^*. Assumiamo anche che L sia un linguaggio sull'alfabeto T. Come suggerito

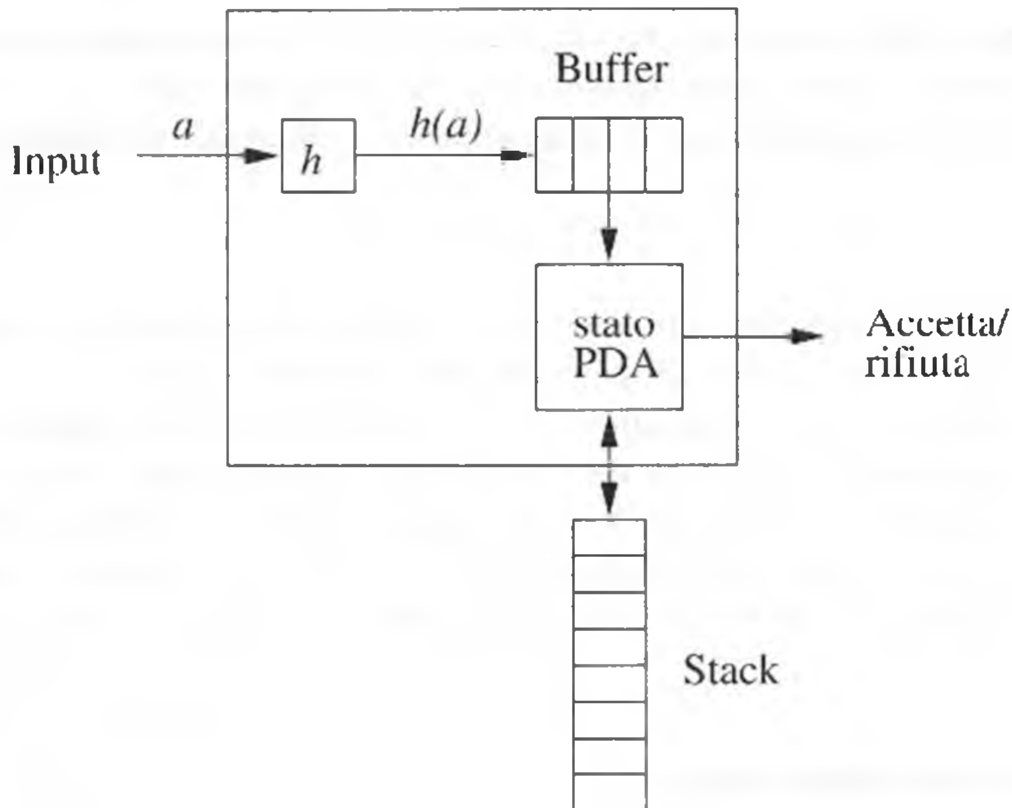


Figura 7.10 Costruzione di un PDA che accetta l'omomorfismo inverso del linguaggio di un PDA assegnato.

in precedenza partiamo da un PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ che accetta L per stato finale. Costruiamo un nuovo PDA

$$P' = (Q', \Sigma, \Gamma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\}) \tag{7.1}$$

dove

1. Q' è l'insieme delle coppie (q, x) tali che:

   (a) q è uno stato in Q

   (b) x è un suffisso (non necessariamente proprio) di una stringa $h(a)$ per un simbolo di input a in Σ.

   Il primo componente dello stato di P' è quindi lo stato di P; il secondo è il buffer. Ipotizziamo che il buffer sia periodicamente riempito con una stringa $h(a)$, e poi perda elementi dalla fronte a mano a mano che i suoi simboli vengono passati al PDA simulato P. Poiché Σ è finito e $h(a)$ è finito per ogni a, P' ha un numero finito di stati.

2. δ' è definita dalle seguenti regole.

(a) $\delta'\big((q,\epsilon),a,X\big) = \big\{\big((q,h(a)),X\big)\big\}$ per ogni simbolo a in Σ, ogni stato q in Q e ogni simbolo X di stack in Γ. Notiamo che qui a non può essere ϵ. Quando il buffer è vuoto, P' può consumare il simbolo di input successivo a e collocare $h(a)$ nel buffer.

(b) Se $\delta(q,b,X)$ contiene (p,γ), dove b è in T o $b = \epsilon$, allora

$$\delta'\big((q,bx),\epsilon,X\big)$$

contiene $\big((p,x),\gamma\big)$. Quindi P' può sempre simulare una mossa di P usando il primo simbolo nel buffer. Se b è un simbolo in T, il buffer non dev'essere vuoto; se $b = \epsilon$, il buffer può essere vuoto.

3. Per la definizione (7.1), lo stato iniziale di P' è (q_0,ϵ); quindi P' parte dallo stato iniziale di P con il buffer vuoto.

4. Similmente, secondo la (7.1), gli stati accettanti di P' sono le coppie (q,ϵ), dove q è uno stato accettante di P.

L'enunciato che segue caratterizza la relazione fra P' e P:

- $\big(q_0,h(w),Z_0\big) \overset{*}{\underset{P}{\vdash}} (p,\epsilon,\gamma)$ se e solo se $\big((q_0,\epsilon),w,Z_0\big) \overset{*}{\underset{P'}{\vdash}} \big((p,\epsilon),\epsilon,\gamma\big)$.

Per dimostrarlo si procede, nelle due direzioni, per induzione sul numero di mosse fatte dai due automi. Nella parte "se" si deve osservare che, quando il buffer non è vuoto, P' non può leggere altri simboli di input e deve simulare P fino a svuotare il buffer (ma può continuare a simulare P anche con il buffer vuoto). Lasciamo i dettagli per esercizio.

Se accettiamo questa relazione fra P' e P, ne deduciamo che P accetta $h(w)$ se e solo se P' accetta w per come sono definiti gli stati accettanti di P'. Perciò $L(P') = h^{-1}\big(L(P)\big)$. □

7.3.6 Esercizi

Esercizio 7.3.1 Mostrate che i CFL sono chiusi rispetto alle seguenti operazioni.

* a) *init*, definita nell'Esercizio 4.2.6(c). *Suggerimento*: partite da una grammatica CNF per il linguaggio L.

*! b) L'operazione L/a, definita nell'Esercizio 4.2.2. *Suggerimento*: anche in questo caso partite da una grammatica CNF per L.

!! c) *cycle*, definita nell'Esercizio 4.2.11. *Suggerimento*: tentate una costruzione basata sui PDA.

Esercizio 7.3.2 Considerate i due linguaggi seguenti:

$$L_1 = \{a^n b^{2n} c^m \mid n, m \geq 0\}$$
$$L_2 = \{a^n b^m c^{2m} \mid n, m \geq 0\}$$

a) Dimostrate che sono entrambi liberi dal contesto definendo una grammatica per ognuno.

! b) $L_1 \cap L_2$ è un CFL? Giustificate la risposta.

!! **Esercizio 7.3.3** Dimostrate che i CFL *non* sono chiusi rispetto alle seguenti operazioni.

* a) *min*, definita nell'Esercizio 4.2.6(a).

b) *max*, definita nell'Esercizio 4.2.6(b).

c) *half*, definita nell'Esercizio 4.2.8.

d) *alt*, definita nell'Esercizio 4.2.7.

Esercizio 7.3.4 Definiamo *shuffle* di due stringhe w e x l'insieme di tutte le stringhe ottenute intercalando arbitrariamente le posizioni di w e x. Più esattamente, *shuffle*(w, x) è l'insieme delle stringhe z tali che:

1. ogni posizione di z si possa attribuire a w o a x, ma non a entrambi
2. le posizioni di z attribuite a w formano w se lette da sinistra a destra
3. le posizioni di z attribuite a x formano x se lette da sinistra a destra.

Per esempio, se $w = 01$ e $x = 110$, *shuffle*$(01, 110)$ è l'insieme di stringhe $\{01110, 01101, 10110, 10101, 11010, 11001\}$. Per illustrare il ragionamento: la terza stringa, 10110, si ottiene attribuendo la seconda e la quinta posizione a 01 e le altre a 110; la prima stringa, 01110, si può ottenere in tre modi attribuendo la prima posizione e una fra la seconda, la terza e la quarta a 01, le altre tre a 110. Possiamo estendere l'operazione ai linguaggi definendo *shuffle*(L_1, L_2) come l'unione su tutte le coppie di stringhe, w in L_1 e x in L_2, di *shuffle*(w, x).

a) Che cos'è *shuffle*$(00, 111)$?

* b) Che cos'è *shuffle*(L_1, L_2) se $L_1 = L(\mathbf{0}^*)$ e $L_2 = \{0^n 1^n \mid n \geq 0\}$?

*! c) Dimostrate che se L_1 ed L_2 sono linguaggi regolari, lo è anche

$$\textit{shuffle}(L_1, L_2)$$

Suggerimento: partite da un DFA per L_1 e da un altro per L_2.

! d) Dimostrate che, se L è un CFL ed R un linguaggio regolare, *shuffle*(L, R) è un CFL. *Suggerimento*: partite da un PDA per L e da un DFA per R.

!! e) Dimostrate con un controesempio che se L_1 ed L_2 sono CFL, *shuffle*(L_1, L_2) può non essere un CFL.

*!! **Esercizio 7.3.5** Una stringa y si dice una *permutazione* della stringa x se possiamo ottenere x riordinando i simboli di y. Per esempio le permutazioni della stringa $x = 011$ sono 110, 101 e 011. Dato un linguaggio L, *perm*(L) è l'insieme delle permutazioni delle stringhe in L. Per esempio, se $L = \{0^n 1^n \mid n \geq 0\}$, *perm*$(L)$ è l'insieme delle stringhe con lo stesso numero di 0 e di 1.

a) Fornite un esempio di linguaggio regolare L sull'alfabeto $\{0, 1\}$ tale che *perm*(L) non sia regolare. Giustificate la risposta. *Suggerimento*: cercate un linguaggio regolare le cui permutazioni siano tutte le stringhe con lo stesso numero di 0 e di 1.

b) Fornite un esempio di linguaggio regolare L sull'alfabeto $\{0, 1, 2\}$ tale che *perm*(L) non sia libero dal contesto.

c) Dimostrate che, per ogni linguaggio regolare L su un alfabeto di due simboli, *perm*(L) è libero dal contesto.

Esercizio 7.3.6 Dimostrate formalmente che i CFL sono chiusi rispetto all'inversione (Teorema 7.25).

Esercizio 7.3.7 Completate la dimostrazione del Teorema 7.27 provando che

$$(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \gamma)$$

se e solo se $\big((q_P, q_A), w, Z_0\big) \overset{*}{\underset{P'}{\vdash}} \big((q, p), \epsilon, \gamma\big)$ e $p = \hat{\delta}(p_A, w)$.

7.4 Proprietà di decisione dei CFL

Esaminiamo ora quali questioni inerenti ai linguaggi liberi dal contesto siamo in grado di risolvere. Come nel Paragrafo 4.3 sulle proprietà di decisione dei linguaggi regolari, partiremo sempre da una rappresentazione di un CFL: o una grammatica o un PDA. Dal Paragrafo 6.3 sappiamo di poter trasformare una grammatica in un PDA, e viceversa; possiamo quindi scegliere la rappresentazione più comoda a seconda dei casi.

Per i CFL scopriremo che ben poco è decidibile: le verifiche più importanti che possiamo fare sono volte a stabilire se un linguaggio sia vuoto o no e se una data stringa

gli appartenga. Il paragrafo termina con una breve discussione su certi problemi che nel Capitolo 9 dimostreremo essere "indecidibili", cioè privi di algoritmi per risolverli. Cominciamo il paragrafo con alcune note sulla complessità della conversione fra grammatiche e PDA. Queste valutazioni riguardano l'efficienza con cui possiamo decidere una proprietà dei CFL per una particolare rappresentazione.

7.4.1 Complessità delle conversioni fra CFG e PDA

Prima di trattare gli algoritmi di decisione per i CFL, esaminiamo la complessità di conversione da una rappresentazione all'altra. Se un linguaggio è specificato in una forma diversa da quella richiesta dall'algoritmo, il tempo necessario per la conversione fa parte del costo di un algoritmo di decisione.

Di qui in avanti denoteremo con n la lunghezza dell'intera rappresentazione di un PDA o di una CFG. Questo modo di rappresentare la dimensione di una grammatica o di un automa è approssimativo perché il tempo di esecuzione di alcuni algoritmi si descrive più esattamente rispetto a parametri più specifici, come il numero delle variabili di una grammatica o la somma delle lunghezze delle stringhe di stack presenti nella funzione di transizione di un PDA.

La lunghezza totale è comunque sufficiente a distinguere i tratti più significativi: se l'algoritmo sia lineare nella lunghezza (cioè impieghi un tempo poco superiore a quello necessario per leggere l'input), se sia esponenziale nella lunghezza (e quindi consenta la conversione solo per esempi piccoli), oppure se sia polinomiale ma non lineare (cioè possa essere eseguito anche per esempi grandi, ma in tempo che può essere rilevante).

Alcune conversioni presentate fin qui sono lineari nella dimensione dell'input. Poiché impiegano tempo lineare, la rappresentazione che producono è non solo generata rapidamente, ma anche di dimensione simile a quella dell'input. Elenchiamole.

1. Conversione di una CFG in PDA tramite l'algoritmo del Teorema 6.13.

2. Conversione di un PDA che accetta per stato finale in PDA che accetta per stack vuoto, tramite la costruzione del Teorema 6.11.

3. Conversione di un PDA che accetta per stack vuoto in PDA che accetta per stato finale, tramite la costruzione del Teorema 6.9.

Valutare il tempo di esecuzione della conversione da PDA a grammatica (Teorema 6.14) è molto più complesso. Notiamo anzitutto che n, la lunghezza totale dell'input, è un limite superiore al numero di stati e di simboli di stack; quindi non possiamo avere nella grammatica più di n^3 variabili della forma $[pXq]$. Il tempo di esecuzione della conversione può essere invece esponenziale, se qualche transizione del PDA inserisce un numero elevato di simboli nello stack. Una regola può collocare sullo stack fino a n simboli.

Riesaminando la costruzione di produzioni da una regola del tipo "$\delta(q, a, X)$ contiene $(r_0, Y_1Y_2 \cdots Y_k)$" notiamo che essa dà luogo a una serie di produzioni della forma $[qXr_k] \to [r_0Y_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$ per tutte le sequenze di stati $r_1, r_2, \ldots, r_k$. Poiché k può essere vicino a n, e il numero di stati può essere vicino a n, il numero totale di produzioni cresce come n^n. La costruzione è quindi impraticabile per PDA di dimensione media se c'è anche una sola stringa di stack lunga.

Per fortuna si può evitare il caso peggiore. Come suggerito nell'Esercizio 6.2.8, possiamo dividere l'inserimento di una stringa di simboli di stack in una sequenza di non più di n passi, ognuno dei quali inserisce un simbolo. Se $\delta(q, a, X)$ contiene $(r_0, Y_1Y_2 \cdots Y_k)$, definiamo i nuovi stati $p_2, p_3, \ldots, p_{k-1}$. A questo punto sostituiamo $(r_0, Y_1Y_2 \cdots Y_k)$ in $\delta(q, a, X)$ con $(p_{k-1}, Y_{k-1}Y_k)$ e introduciamo le nuove transizioni

$$\delta(p_{k-1}, Y_{k-1}) = \{(p_{k-2}, Y_{k-2}Y_{k-1})\}, \; \delta(p_{k-2}, Y_{k-2}) = \{(p_{k-3}, Y_{k-3}Y_{k-2})\}$$

e così via, fino a $\delta(p_2, Y_2) = \{(r_0, Y_1Y_2)\}$.

Nessuna transizione ha ora più di due simboli di stack. Abbiamo aggiunto al massimo n nuovi stati, e la lunghezza totale di tutte le regole di transizione di δ è cresciuta al massimo di un fattore costante, rimanendo quindi $O(n)$. Ci sono $O(n)$ regole di transizione, ognuna delle quali genera $O(n^2)$ produzioni perché nelle produzioni derivate da ogni regola si devono scegliere solo due stati. La grammatica così formata ha quindi lunghezza $O(n^3)$ e si costruisce in tempo cubico. Il prossimo teorema compendia l'analisi informale.

Teorema 7.31 C'è un algoritmo di costo $O(n^3)$ che, a partire da un PDA P la cui rappresentazione è lunga n, produce una CFG di lunghezza $O(n^3)$. La CFG genera il linguaggio che P accetta per stack vuoto. Eventualmente possiamo far sì che G generi il linguaggio accettato da P per stato finale. □

7.4.2 Tempo di esecuzione della conversione in forma normale di Chomsky

Poiché un algoritmo di decisione può richiedere che una CFG sia in forma normale di Chomsky, dobbiamo esaminare il tempo di esecuzione degli algoritmi definiti per convertire una grammatica qualsiasi in grammatica CNF. I passi preservano in genere, a meno di un fattore costante, la lunghezza della descrizione: partendo da una grammatica di lunghezza n ne producono una di lunghezza $O(n)$. Le osservazioni che seguono precisano questa affermazione.

1. Con un algoritmo opportuno (vedi Paragrafo 7.4.3) si possono rilevare i simboli raggiungibili e generatori di una grammatica in tempo $O(n)$. L'eliminazione dei simboli inutili richiede tempo $O(n)$ e non aumenta la dimensione della grammatica.

2. Determinare le coppie unitarie ed eliminare le produzioni unitarie come nel Paragrafo 7.1.4 richiede tempo $O(n^2)$; la grammatica risultante ha lunghezza $O(n^2)$.

3. La sostituzione di terminali con variabili nel corpo delle produzioni come nel Paragrafo 7.1.5 (forma normale di Chomsky) richiede tempo $O(n)$ e determina una grammatica di lunghezza $O(n)$.

4. Anche la scomposizione dei corpi di lunghezza 3, o più, in corpi di lunghezza 2, svolta nel Paragrafo 7.1.5, richiede tempo $O(n)$ e dà luogo a una grammatica di lunghezza $O(n)$.

Il punto critico riguarda la costruzione del Paragrafo 7.1.3, in cui si eliminano le ϵ-produzioni. Da una sola produzione il cui corpo è lungo k possiamo generare $2^k - 1$ produzioni per la nuova grammatica. Poiché k può essere proporzionale a n, questa parte della costruzione può richiedere tempo $O(2^n)$ e produrre una grammatica di lunghezza $O(2^n)$.

Per evitare questa crescita esponenziale basta limitare la lunghezza dei corpi delle produzioni. L'espediente del Paragrafo 7.1.5 si può applicare a qualsiasi corpo, non soltanto a quelli privi di terminali. Suggeriamo perciò, come passo preliminare all'eliminazione delle ϵ-produzioni, di spezzare i corpi più lunghi in sequenze di produzioni con corpi di lunghezza 2. Questo passo richiede tempo $O(n)$ e allunga la grammatica di un fattore solo lineare. La costruzione del Paragrafo 7.1.3 per eliminare le ϵ-produzioni opera allora su corpi di lunghezza al più 2, con un tempo di esecuzione $O(n)$, e genera una grammatica di lunghezza $O(n)$.

Modificando così la costruzione CNF, l'unico passo non lineare è l'eliminazione delle produzioni unitarie. Poiché questo passo è $O(n^2)$, otteniamo il seguente teorema.

Teorema 7.32 Data una grammatica G di lunghezza n, possiamo costruire in tempo $O(n^2)$ una grammatica equivalente in forma normale di Chomsky di lunghezza $O(n^2)$. □

7.4.3 Verificare se un CFL è vuoto

Abbiamo già studiato l'algoritmo per verificare se un CFL L è vuoto. Data una grammatica G per L, si applica l'algoritmo del Paragrafo 7.1.2 per stabilire se S, il simbolo iniziale di G, è generatore, cioè se da S derivi almeno una stringa. L è vuoto se e solo se S non è generatore.

Data la sua importanza, tratteremo in dettaglio il tempo che questo metodo richiede per trovare tutti i simboli generatori di una grammatica G. Supponiamo che la lunghezza di G sia n. Il numero di variabili è allora dell'ordine di n, e ogni passo della ricerca induttiva di variabili generatrici può richiedere tempo $O(n)$ per esaminare tutte le produzioni di G. Se a ogni passo si scopre solo una nuova variabile generatrice, possono

essere necessari $O(n)$ passi. Dunque un'implementazione ingenua del metodo ha un costo $O(n^2)$.

Esiste però un algoritmo più scaltro, che prepara una struttura dati appropriata e riduce il tempo di ricerca dei generatori a $O(n)$. La struttura, illustrata nella Figura 7.11, comincia con un array (a sinistra nella Figura) i cui indici sono le variabili, per ognuna delle quali dice se ha già stabilito che è generatrice. Nella Figura 7.11, secondo quanto indica l'array, abbiamo già scoperto che B è generatrice, ma non sappiamo se lo sia A. Dal momento che una variabile non è generatrice se l'algoritmo non la dichiara tale, al termine dell'esecuzione i punti interrogativi diventano dei "no".

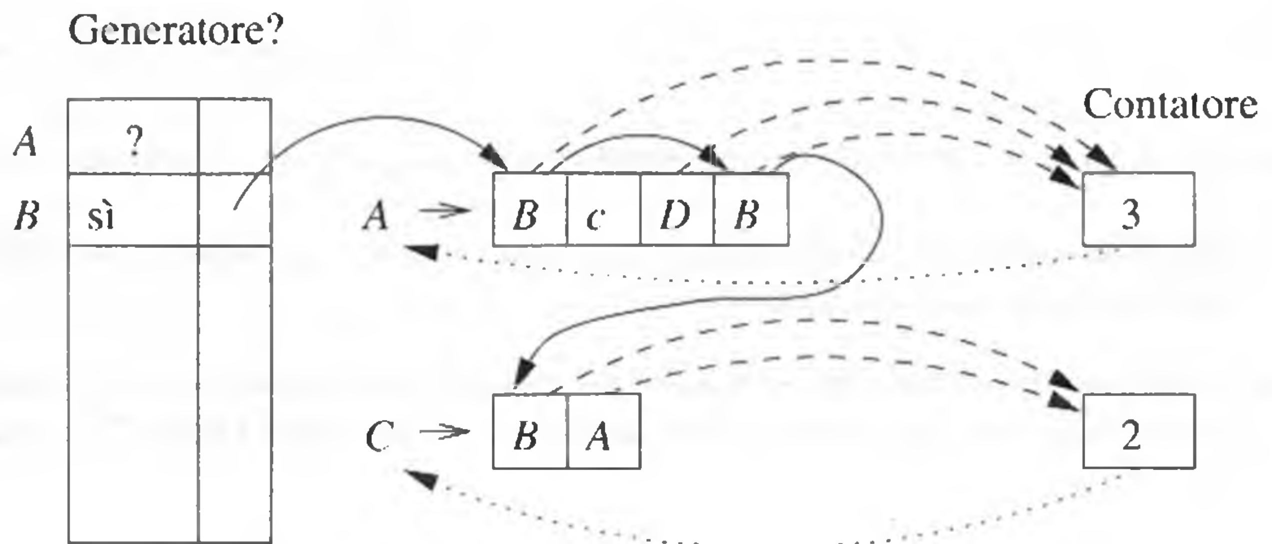


Figura 7.11 Struttura dati per la verifica in tempo lineare di linguaggio vuoto.

Si elaborano le produzioni allestendo liste di diverso tipo. Innanzitutto a ogni variabile si associa una lista delle posizioni in cui compare. Per esempio la lista per la variabile B è indicata dagli archi a tratto continuo. Per ogni produzione si conserva il conto delle posizioni occupate da variabili di cui non si è ancora stabilito se generano stringhe terminali. Gli archi tratteggiati collegano ogni produzione al rispettivo contatore. I contatori nella Figura 7.11 indicano che non abbiamo ancora esaminato alcuna variabile, anche se abbiamo appena stabilito che B è generatrice.

Supponiamo di avere scoperto che B è generatrice. Scorriamo l'elenco delle posizioni dei corpi che contengono B. Per ogni posizione diminuiamo di 1 il contatore della produzione; per stabilire se la variabile di testa è generatrice dobbiamo ora esaminare una posizione in meno.

Quando un contatore raggiunge lo 0 sappiamo che la variabile di testa è generatrice. Il puntatore indicato da una linea punteggiata conduce alla variabile, che possiamo ora inserire nella fila delle variabili generatrici di cui dobbiamo esaminare le conseguenze (come abbiamo fatto per B). La fila non è illustrata.

Altre applicazioni della verifica lineare di linguaggio vuoto

La stessa struttura dati e la stessa tecnica impiegata nel Paragrafo 7.4.3 per stabilire se una variabile è generatrice consentono di eseguire in tempo lineare anche altre verifiche citate nel Paragrafo 7.1. Due esempi importanti:

1. quali simboli sono raggiungibili?
2. quali simboli sono annullabili?

Dobbiamo dimostrare che l'algoritmo richiede tempo $O(n)$. Ci sono tre punti cruciali.

- Poiché una grammatica di dimensione n ha al massimo n variabili, la creazione e la preparazione dell'array richiedono tempo $O(n)$.
- Poiché ci sono al massimo n produzioni, e la loro lunghezza totale è al massimo n, la preparazione dei puntatori e dei contatori illustrata nella Figura 7.11 si compie in tempo $O(n)$.
- Il lavoro fatto quando si scopre che il contatore per una produzione è 0 (tutti i simboli nel suo corpo sono generatori) si divide in due categorie.
    1. Lavoro svolto per la produzione: rilevare che il contatore è 0; stabilire quale variabile, poniamo A, è in testa; verificare se è già stata dichiarata generatrice e se necessario metterla in fila. Tutti questi passi costano $O(1)$ per ogni produzione. Il costo totale in questa categoria è $O(n)$.
    2. Lavoro svolto nell'esaminare le posizioni nei corpi delle produzioni aventi A in testa. Questo lavoro è proporzionale al numero di posizioni per A. Il lavoro complessivo per trattare tutti i simboli generatori è quindi proporzionale alla somma delle lunghezze dei corpi, che è $O(n)$.

Concludiamo che il lavoro totale svolto dall'algoritmo è $O(n)$.

7.4.4 Appartenenza a un CFL

Possiamo anche stabilire se una stringa w appartiene a un CFL L. Supponendo assegnata una grammatica o un PDA per L, la cui dimensione si possa considerare costante e indipendente da w, esistono diversi modi per farlo in tempo esponenziale rispetto a $|w|$, cioè inefficienti. Per esempio si può cominciare convertendo la rappresentazione di L in una

grammatica CNF equivalente. Poiché gli alberi sintattici di una grammatica CNF sono binari, se w è lunga n, nell'albero ci sono esattamente $2n - 1$ nodi etichettati da variabili (la dimostrazione, induttiva, è semplice e la lasciamo al lettore). Il numero di alberi ed etichettature possibili è quindi "solo" esponenziale in n; in linea di principio possiamo scriverli tutti e osservare se uno di essi ha per prodotto w.

Una tecnica molto più efficiente si fonda sulla "programmazione dinamica". L'algoritmo, noto come *algoritmo CYK*[3], parte da una grammatica CNF $G = (V, T, P, S)$ per un linguaggio L. L'input dell'algoritmo è una stringa $w = a_1a_2 \cdots a_n$ in T^*. In tempo $O(n^3)$ l'algoritmo costruisce una tabella che dice se w è in L. Si noti che, nel calcolare il tempo d'esecuzione, si considera fissata la grammatica; la sua dimensione influisce solo per un fattore costante sul tempo, misurato rispetto alla lunghezza della stringa w di cui si verifica l'appartenenza a L.

Nell'algoritmo CYK si costruisce una tabella triangolare, come illustrato nella Figura 7.12. L'asse orizzontale corrisponde alle posizioni della stringa $w = a_1a_2 \cdots a_n$, che supponiamo lunga 5. La voce X_{ij} della tabella è l'insieme delle variabili A tali che $A \overset{*}{\Rightarrow} a_ia_{i+1} \cdots a_j$. In particolare ci interessa sapere se S è nell'insieme X_{1n} perché ciò equivale a dire $S \overset{*}{\Rightarrow} w$, ossia w è in L.

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

Figura 7.12 La tabella costruita dall'algoritmo CYK.

Per riempire la tabella procediamo verso l'alto, riga per riga. Ogni riga corrisponde a un valore di lunghezza delle sottostringhe; la più bassa è per le stringhe di lunghezza

[3]Dal nome dei tre ricercatori che nella sostanza hanno concepito, indipendentemente l'uno dagli altri, la stessa idea: J. Cocke, D. Younger e T. Kasami.

1, la seconda dal basso per quelle di lunghezza 2, e così via, fino alla riga in alto, che corrisponde all'unica sottostringa lunga n, cioè w. L'elaborazione di ogni voce della tabella, secondo il metodo che descriveremo ora, richiede tempo $O(n)$. Poiché la tabella ospita $n(n+1)/2$ voci, l'intera procedura di riempimento richiede tempo $O(n^3)$. Ecco l'algoritmo per determinare gli X_{ij}.

BASE La prima riga si calcola così. La stringa che comincia e finisce nella posizione i è il terminale a_i; poiché la grammatica è in CNF, c'è un solo modo di derivare la stringa a_i: una produzione della forma $A \rightarrow a_i$. Dunque X_{ii} è l'insieme delle variabili A tali che $A \rightarrow a_i$ è una produzione di G.

INDUZIONE Vogliamo calcolare X_{ij}, che si trova sulla riga $j - i + 1$, dopo aver calcolato tutti gli X nelle righe sottostanti. Supponiamo cioè di conoscere la situazione relativa a tutte le stringhe più corte di $a_i a_{i+1} \cdots a_j$, e in particolare tutti i suoi prefissi e suffissi propri. Possiamo assumere $j - i > 0$ perché il caso $i = j$ forma la base; quindi sappiamo che ogni derivazione $A \overset{*}{\Rightarrow} a_i a_{i+1} \cdots a_j$ deve cominciare da un passo $A \Rightarrow BC$. Quindi da B deriva un prefisso di $a_i a_{i+1} \cdots a_j$, diciamo $B \overset{*}{\Rightarrow} a_i a_{i+1} \cdots a_k$ per un $k < j$. Invece da C deve derivare il resto di $a_i a_{i+1} \cdots a_j$, cioè $C \overset{*}{\Rightarrow} a_{k+1} a_{k+2} \cdots a_j$.

Concludiamo che per porre A in X_{ij} dobbiamo trovare due variabili B e C, e un intero k, tali che:

1. $i \leq k < j$
2. B è in X_{ik}
3. C è in $X_{k+1,j}$
4. $A \rightarrow BC$ è una produzione di G.

Per trovare queste variabili dobbiamo confrontare al massimo n coppie di insiemi già calcolati: $(X_{ii}, X_{i+1,j})$, $(X_{i,i+1}, X_{i+2,j})$, e così via, fino a $(X_{i,j-1}, X_{jj})$. Lo schema, cioè salire lungo la colonna sotto X_{ij} e contemporaneamente scendere lungo la diagonale, è illustrato nella Figura 7.13.

Teorema 7.33 L'algoritmo descritto calcola correttamente X_{ij} per ogni i e j; perciò w è in $L(G)$ se e solo se S è in X_{1n}. Il tempo d'esecuzione dell'algoritmo è $O(n^3)$.

DIMOSTRAZIONE La ragione per cui l'algoritmo trova gli insiemi corretti di variabili è stata spiegata nella discussione sulla base e sul passo induttivo dell'algoritmo. Quanto al tempo d'esecuzione, notiamo che si devono calcolare $O(n^2)$ elementi, ognuno dei quali comporta confronti e operazioni su n coppie di elementi. Va ricordato che, anche se ogni insieme X_{ij} può contenere molte variabili, la grammatica G è fissa e il numero delle sue variabili non dipende da n, lunghezza della stringa w di cui dobbiamo stabilire

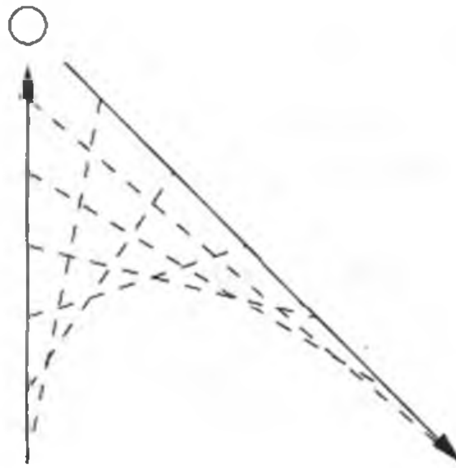


Figura 7.13 Il calcolo di X_{ij} richiede di abbinare la colonna sottostante con la diagonale a destra.

l'appartenenza. Quindi il tempo per confrontare X_{ik} e $X_{k+1,j}$, e per trovare le variabili da inserire in X_{ij} è $O(1)$. Poiché ci sono al massimo n coppie da confrontare per ogni X_{ij}, il costo totale è $O(n^3)$. □

Esempio 7.34 Elenchiamo le produzioni di una grammatica CNF.

$$\begin{array}{lll} S & \rightarrow & AB \mid BC \\ A & \rightarrow & BA \mid a \\ B & \rightarrow & CC \mid b \\ C & \rightarrow & AB \mid a \end{array}$$

Verifichiamo se la stringa *baaba* appartiene a $L(G)$. La Figura 7.14 riproduce la tabella compilata per questa stringa.

Per costruire la prima riga dal basso ci serviamo della regola di base. Dobbiamo considerare solamente le variabili con una produzione il cui corpo sia a (A e C) o b (solo B). Perciò sopra le posizioni occupate da a vediamo l'elemento $\{A, C\}$, e sopra quelle occupate da b vediamo $\{B\}$. In altre parole $X_{11} = X_{44} = \{B\}$, e $X_{22} = X_{33} = X_{55} = \{A, C\}$.

Nella seconda riga vediamo i valori di X_{12}, X_{23}, X_{34} e X_{45}. A titolo d'esempio consideriamo X_{12}. C'è un solo modo di spezzare la stringa dalla posizione 1 alla 2, cioè *ba*, in due sottostringhe non vuote. La prima dev'essere la posizione 1, la seconda la posizione 2. Per generare *ba*, una variabile deve avere un corpo la cui prima variabile sia in $X_{11} = \{B\}$ (deve cioè generare b) e la seconda sia in $X_{22} = \{A, C\}$ (per generare a). Questo corpo può essere soltanto BA o BC. L'esame della grammatica rivela che le produzioni $A \rightarrow BA$ e $S \rightarrow BC$ sono le uniche con questa caratteristica. Perciò le due teste, A ed S, formano X_{12}.

$\{S,A,C\}$				
-	$\{S,A,C\}$			
-	$\{B\}$	$\{B\}$		
$\{S,A\}$	$\{B\}$	$\{S,C\}$	$\{S,A\}$	
$\{B\}$	$\{A,C\}$	$\{A,C\}$	$\{B\}$	$\{A,C\}$
b	a	a	b	a

Figura 7.14 La tabella per la stringa *baaba* costruita dall'algoritmo CYK.

Come esempio più articolato consideriamo il calcolo di X_{24}. Possiamo spezzare la stringa aab, che occupa le posizioni da 2 a 4, terminando la prima stringa dopo la posizione 2 o dopo la 3. Possiamo cioè scegliere $k = 2$ o $k = 3$ nella definizione di X_{24}. Dobbiamo allora esaminare tutti i corpi in $X_{22}X_{34} \cup X_{23}X_{44}$. Questo insieme di stringhe è $\{A,C\}\{S,C\} \cup \{B\}\{B\} = \{AS, AC, CS, CC, BB\}$. Dei suoi cinque elementi solo CC è un corpo; la sua testa è B. Perciò $X_{24} = \{B\}$. □

7.4.5 Anteprima di problemi indecidibili per i CFL

Nei prossimi capitoli svilupperemo una notevole teoria che dimostra rigorosamente l'esistenza di problemi irrisolvibili da qualsiasi algoritmo eseguito da un calcolatore. Ce ne serviremo per presentare diverse questioni di semplice enunciazione, riguardanti grammatiche e CFL, per le quali non disponiamo di algoritmi: li chiameremo "problemi indecidibili". Per ora accontentiamoci di un elenco dei più significativi problemi indecidibili attinenti a grammatiche e linguaggi liberi dal contesto.

1. Una CFG G assegnata è ambigua?

2. Un dato linguaggio CFL è inerentemente ambiguo?

3. L'intersezione di due CFL dati è vuota?

4. Due CFL dati sono uguali?

5. Un CFL dato è uguale a Σ^*, dove Σ è l'alfabeto del linguaggio?

Notiamo che la natura del problema (1), sull'ambiguità, si distingue dalle altre per essere inerente a una grammatica e non a un linguaggio. Gli altri problemi assumono tutti che il linguaggio sia rappresentato da una grammatica o da un PDA, ma vertono appunto su linguaggi. Per esempio, diversamente dal problema (1), il secondo chiede se, data una grammatica G (o anche un PDA), esiste una grammatica equivalente non ambigua. Se G stessa è non ambigua la risposta è certamente "sì"; in caso contrario potrebbe esistere una grammatica non ambigua per lo stesso linguaggio, come abbiamo visto per la grammatica delle espressioni nell'Esempio 5.27.

7.4.6 Esercizi

Esercizio 7.4.1 Ideate un algoritmo per ognuno dei seguenti problemi.

* a) Data una CFG G, $L(G)$ è finito? *Suggerimento*: applicate il *pumping lemma*.

! b) Data una CFG G, $L(G)$ contiene almeno 100 stringhe?

!! c) Data una CFG G e una delle sue variabili A, esiste una forma sentenziale in cui A è il primo simbolo? *Nota*: ricordate che, anche se A compare per la prima volta in mezzo a una forma sentenziale, i simboli alla sua sinistra potrebbero generare ϵ.

Esercizio 7.4.2 Con l'ausilio della tecnica descritta nel Paragrafo 7.4.3 sviluppate algoritmi in tempo lineare per i seguenti problemi sulle CFG.

a) Quali simboli compaiono in almeno una forma sentenziale?

b) Quali simboli sono annullabili (generano ϵ)?

Esercizio 7.4.3 Applicate l'algoritmo CYK per stabilire quali delle stringhe che seguono sono in $L(G)$, dove G è la grammatica dell'Esempio 7.34.

* a) $ababa$.

b) $baaab$.

c) $aabab$.

* **Esercizio 7.4.4** Dimostrate che in ogni grammatica CNF tutti gli alberi sintattici di stringhe lunghe n hanno $2n - 1$ nodi interni (nodi etichettati da variabili).

! **Esercizio 7.4.5** Modificate l'algoritmo CYK in modo che calcoli il numero di alberi sintattici distinti per la stringa di input anziché limitarsi a stabilire se appartiene al linguaggio.

7.5 Riepilogo

- *Eliminazione di simboli inutili*: possiamo eliminare una variabile da una CFG se non genera alcuna stringa di terminali e non compare in almeno una stringa derivata dal simbolo iniziale. Per eliminare correttamente i simboli inutili dobbiamo prima verificare se da una variabile deriva una stringa terminale ed eliminare, insieme con tutte le loro produzioni, quelle che non soddisfano il vincolo. Solo in seguito eliminiamo le variabili non derivabili dal simbolo iniziale.

- *Eliminazione delle ϵ-produzioni e delle produzioni unitarie*: data una CFG, esiste un'altra CFG che genera lo stesso linguaggio, tranne la stringa ϵ, ma è priva di ϵ-produzioni (quelle con corpo formato da ϵ) e di produzioni unitarie (quelle con corpo formato da una sola variabile).

- *Forma normale di Chomsky*: data una CFG che genera almeno una stringa non vuota, ne esiste un'altra che genera lo stesso linguaggio, tranne ϵ, e che è in forma normale di Chomsky: non ci sono simboli inutili e il corpo di ogni produzione consiste o di due variabili o di un terminale.

- *Il pumping lemma*: dato un CFL, in ogni stringa sufficientemente lunga del linguaggio si può trovare una sottostringa tale che i suoi estremi possono essere replicati in tandem, cioè ripetuti un numero arbitrario di volte. Le stringhe replicabili non sono entrambe ϵ. Con questo lemma, e con una versione più forte detta lemma di Ogden, citata nell'Esercizio 7.2.3, si può dimostrare che molti linguaggi non sono liberi dal contesto.

- *Operazioni che preservano i linguaggi liberi dal contesto*: i CFL sono chiusi rispetto alle operazioni di sostituzione, unione, concatenazione, chiusura, inversione e omomorfismo inverso. Non sono chiusi rispetto all'intersezione e al complemento, anche se l'intersezione di un CFL e di un linguaggio regolare è sempre un CFL.

- *Verificare se un CFL è vuoto*: data una CFG, c'è un algoritmo che rivela se essa genera almeno una stringa. Mediante un'implementazione accurata, l'esecuzione dell'algoritmo richiede tempo proporzionale alla dimensione della grammatica.

- *Verificare l'appartenenza a un CFL*: l'algoritmo di Cocke-Younger-Kasami rivela se una stringa appartiene a un dato linguaggio libero dal contesto. Fissato un CFL, l'algoritmo richiede tempo $O(n^3)$, dove n è la lunghezza della stringa in esame.

7.6 Bibliografia

La forma normale di Chomsky è introdotta in [2], la forma normale di Greibach in [4], anche se la costruzione delineata nell'Esercizio 7.1.11 si deve a M.C. Paull.

Molte proprietà fondamentali dei linguaggi liberi dal contesto sono introdotte in [1]. Tra queste il *pumping lemma*, le prime proprietà di chiusura, e alcuni metodi per verificare semplici proprietà, come la vuotezza e la finitezza di un CFL. Aggiungiamo [6] come origine della non chiusura rispetto all'intersezione e al complemento, mentre [3] presenta altri risultati di chiusura, come la chiusura dei CFL rispetto all'omomorfismo inverso. Il lemma di Ogden è introdotto in [5].

Per l'algoritmo CYK sono note tre fonti indipendenti: il lavoro di J. Cocke, che è stato diffuso solo privatamente, senza essere pubblicato; la versione di T. Kasami di un algoritmo sostanzialmente identico, inclusa in un memorandum interno della US-Air-Force; il lavoro di D. Younger, che è stato invece pubblicato [7].

1. Y. Bar-Hillel, M. Perles, E. Shamir, "On formal properties of simple phrase-structure grammars," *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), pp. 143–172.

2. N. Chomsky, "On certain formal properties of grammars," *Information and Control* **2**:2 (1959), pp. 137–167.

3. S. Ginsburg, G. Rose, "Operations which preserve definability in languages," *J. ACM* **10**:2 (1963), pp. 175–195.

4. S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars," *J. ACM* **12**:1 (1965), pp. 42–52.

5. W. Ogden, "A helpful result for proving inherent ambiguity," *Mathematical Systems Theory* **2**:3 (1969), pp. 31–42.

6. S. Scheinberg, "Note on the boolean properties of context-free languages," *Information and Control* **3**:4 (1960), pp. 372–375.

7. D. H. Younger, "Recognition and parsing of context-free languages in time n^3," *Information and Control* **10**:2 (1967), pp. 189–208.

Capitolo 8

Macchine di Turing: introduzione

In questo capitolo cambiamo decisamente direzione. Finora ci siamo occupati soprattutto di semplici classi di linguaggi e dei modi di utilizzarle in relazione a problemi circoscritti, come l'analisi di protocolli, la ricerca in testi e l'analisi sintattica di programmi. Ora cominciamo a chiederci quali linguaggi possano essere definiti mediante un dispositivo di calcolo qualsiasi. Ciò equivale a chiedersi che cosa può fare un calcolatore. Infatti possiamo esprimere qualsiasi problema in modo formale come problema di riconoscimento delle stringhe di un linguaggio, e d'altra parte possiamo considerare un calcolatore come un dispositivo per "risolvere problemi".

Supponendo noto il linguaggio C, cominciamo da un ragionamento informale che dimostra l'esistenza di problemi specifici, irrisolvibili da un calcolatore. Questi problemi si dicono "indecidibili". Presentiamo poi un modello formale, ormai venerando, di calcolatore: la macchina di Turing. Una macchina di Turing non somiglia affatto a un PC. Se per caso un'azienda dovesse pensare di produrla e venderla, risulterebbe disastrosamente lenta; eppure da molto tempo è riconosciuta come un modello preciso di quello che qualsiasi dispositivo fisico di calcolo può fare.

Nel Capitolo 9 ci serviremo della macchina di Turing per sviluppare una teoria dei problemi "indecidibili" (problemi che nessun calcolatore può risolvere). Dimostreremo che certi problemi, facili da enunciare, sono in realtà indecidibili. Un esempio? Stabilire se una grammatica è ambigua. Ne vedremo però molti altri.

8.1 Problemi che i calcolatori non possono risolvere

Questo paragrafo ha lo scopo di presentare informalmente, sulla base della programmazione in C, la dimostrazione che un calcolatore non può risolvere un problema specifico. Il problema consiste nello stabilire se la prima cosa che un programma in C stampa è

`Ciao, mondo`. Possiamo pensare che simulando il programma si possa stabilire che cosa fa, ma dobbiamo fare i conti con programmi che impiegano un tempo straordinariamente lungo per produrre un output. Questo problema – non sapere quando qualcosa avverrà, se mai avverrà – è la ragione principale della nostra incapacità di stabilire che cosa fa un programma. Dimostrare formalmente che non esiste alcun programma in grado di svolgere un certo compito è difficile, e per farlo dobbiamo sviluppare opportuni strumenti formali. In questo paragrafo spieghiamo l'idea intuitiva su cui poggiano le dimostrazioni formali.

8.1.1 Programmi che stampano "Ciao, mondo"

La Figura 8.1 riproduce il primo programma in C in cui si imbatte il lettore del classico testo di Kernighan e Ritchie.[1] È facile scoprire che il programma stampa `Ciao, mondo` e termina. Data la sua immediatezza, oggi si è soliti introdurre un linguaggio di programmazione mostrando come scrivere un programma che stampa `Ciao, mondo`.

```
main()
{
    printf("Ciao, mondo\n");
}
```

Figura 8.1 Il programma di saluto di Kernighan e Ritchie.

Esistono anche altri programmi in grado di stampare `Ciao, mondo`, ma che lo facciano è tutt'altro che ovvio. La Figura 8.2 ne illustra un esempio. Il programma prende come input n e cerca soluzioni intere positive all'equazione $x^n + y^n = z^n$. Se ne trova una, stampa `Ciao, mondo`. Se non esistono tre interi x, y e z che soddisfano l'equazione, la ricerca non ha mai fine e il programma non arriva a stampare la frase.

Per capire che cosa fa questo programma, osserviamo in primo luogo che `exp` è una funzione ausiliaria che calcola esponenziali. Il programma deve esaminare le triple (x, y, z) in un ordine tale da garantire che ogni tripla di interi positivi venga prima o poi esaminata. Per organizzare adeguatamente la ricerca, ricorriamo a una quarta variabile, `total`, che parte da 3 e nel ciclo while è incrementata di un'unità alla volta in modo da raggiungere qualunque intero finito. All'interno del ciclo while dividiamo `total` in tre interi positivi x, y e z, dapprima facendo variare x da 1 a `total-2`, e poi, all'interno del ciclo for, facendo variare y da 1 alla differenza fra `total` e x, diminuita di uno. Il rimanente, che è compreso tra 1 e `total-2`, viene assegnato a z.

---

[1] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, 1978, Prentice-Hall, Englewood Cliffs, NJ.

```
int exp(int i, n)
/* calcola i elevato a n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("Ciao, mondo\n");
            }
        total++;
    }
}
```

Figura 8.2 L'ultimo teorema di Fermat espresso come programma di saluto.

Nel ciclo più interno si verifica se $x^n + y^n = z^n$. Se sì, il programma stampa `Ciao, mondo`; in caso contrario non stampa nulla.

Se il valore di n letto dal programma è 2, il programma trova combinazioni di interi, come `total` = 12, $x = 3$, $y = 4$ e $z = 5$, per cui $x^n + y^n = z^n$. Di conseguenza, per l'input 2, il programma stampa effettivamente `Ciao, mondo`.

D'altro canto, per qualunque intero $n > 2$, il programma non troverà mai una tripla di interi positivi che soddisfi $x^n + y^n = z^n$, e dunque non potrà stampare `Ciao, mondo`. È interessante osservare che fino a qualche anno fa non si sapeva se il programma avrebbe stampato `Ciao, mondo` per valori interi di n molto grandi. Fu Fermat a sostenere per primo, trecento anni fa, che non lo avrebbe fatto, ossia che non esistono soluzioni intere all'equazione $x^n + y^n = z^n$ se $n > 2$. Questo enunciato, noto come "ultimo teorema di Fermat", è stato dimostrato solo di recente.

Perché devono esistere problemi indecidibili

Se è difficile dimostrare che un problema specifico, come il "problema ciao-mondo" trattato qui, è indecidibile, è invece molto facile capire le ragioni per cui quasi tutti i problemi devono essere indecidibili per un sistema di programmazione. Ricordiamo che per "problema" intendiamo l'appartenenza di una stringa a un linguaggio. Il numero di linguaggi diversi su un qualunque alfabeto di più di un simbolo non è numerabile. In altre parole non esiste un modo di assegnare interi ai linguaggi tale che ogni linguaggio corrisponda a un intero e ogni intero sia assegnato a un unico linguaggio.

D'altra parte i programmi, essendo stringhe finite su un alfabeto finito (solitamente un sottoinsieme dell'alfabeto ASCII), *sono* numerabili. Infatti i programmi possono essere ordinati per lunghezza, e i programmi della stessa lunghezza possono essere ordinati lessicalmente. Di conseguenza si può parlare del primo programma, del secondo programma, e in generale dell'i-esimo programma per un qualunque intero i.

Ne deduciamo che esistono infinitamente meno programmi che problemi. Se scegliessimo un linguaggio a caso, quasi sicuramente sarebbe un problema indecidibile. L'unica ragione per cui *sembra* che la maggior parte dei problemi sia decidibile è che di rado siamo interessati a problemi scelti a caso; preferiamo piuttosto considerare problemi semplici e ben strutturati, che effettivamente sono spesso decidibili. Eppure anche tra i problemi di cui ci interessiamo e che possiamo enunciare in termini chiari e concisi, ce ne sono molti indecidibili; il problema "ciao-mondo" è uno di questi.

Il *problema ciao-mondo* consiste nello stabilire se i primi 11 caratteri stampati da un dato programma C, con un dato input, sono `Ciao, mondo`. In seguito diremo spesso, per brevità, che un programma stampa `Ciao, mondo` per indicare che stampa queste due parole come i primi 11 caratteri.

Se i matematici hanno impiegato 300 anni per risolvere una questione relativa a un solo programma lungo 22 righe, il problema generale di dire se un programma dato, su un input dato, stampa `Ciao, mondo` dev'essere veramente arduo. In effetti tutti i problemi che i matematici non hanno saputo risolvere possono essere trasformati nella domanda: "il tal programma, con tale input, stampa `Ciao, mondo`?" Sarebbe dunque notevole se riuscissimo a scrivere un programma in grado di esaminare qualunque programma P e input I per P, e dire se P, eseguito con I come input, stampa `Ciao, mondo`. Dimostreremo che un tale programma non esiste.

8.1.2 Un ipotetico verificatore di ciao-mondo

L'impossibiltà di compiere la verifica di ciao-mondo si dimostra per assurdo. In altre parole supponiamo che esista un programma, che chiameremo H, che prende come input un programma P e un input I, e dice se P con input I stampa `Ciao, mondo`. La Figura 8.3 illustra uno schema di H. In particolare l'unico output di H è la stampa dei due caratteri `sì` oppure dei due caratteri `no`. H fa sempre o l'una o l'altra cosa.

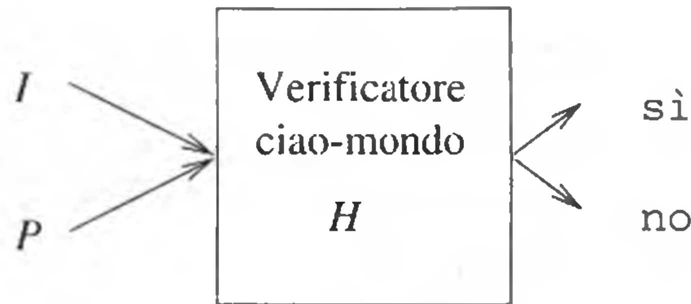


Figura 8.3 Un ipotetico programma H che funge da verificatore di ciao-mondo.

Se un problema ha un algoritmo come H, che dice sempre in maniera corretta se un'istanza del problema riceve la risposta "sì" oppure "no", il problema è detto "decidibile". Negli altri casi il problema è "indecidibile". Il nostro obiettivo è dimostrare che H non esiste, vale a dire che il problema ciao-mondo è indecidibile.

Per dimostrare l'enunciato per assurdo, apporteremo diverse modifiche ad H, fino a giungere a un altro programma, detto H_2, di cui proveremo la non esistenza. Visto che le modifiche ad H sono semplici trasformazioni che possono essere applicate a qualunque programma C, l'unico enunciato dubbio è l'esistenza di H. Questo è dunque l'assunto che avremo contraddetto.

Per facilitare la discussione imporremo alcuni vincoli ai programmi in C, tesi a rendere più semplice il compito di H. Se possiamo dimostrare che un "verificatore di ciao-mondo" non esiste per questi programmi limitati, a maggior ragione non può esistere per una classe di programmi più ampia. I vincoli sono i seguenti.

1. L'output è a caratteri; non usiamo pacchetti grafici o qualunque altro mezzo per ottenere un output che non sia formato da caratteri.

2. L'output a caratteri si ottiene usando `printf`, e non `putchar()` o altre funzioni di output.

Assumiamo ora che il programma H esista. La prima modifica consiste nel mutare l'output `no`, che rappresenta la risposta di H quando il suo programma di input P non stampa

`Ciao, mondo` come primo output per l'input I. Non appena H stampa "n", sappiamo che seguirà la lettera "o".[2] Di conseguenza possiamo modificare qualunque istruzione `printf` in H che stampa "n" in modo che stampi invece `Ciao, mondo`. Un'altra istruzione `printf` che stampi una "o", ma non la "n", viene omessa. Ne risulta che il nuovo programma, che chiameremo H_1, si comporta come H, tranne per il fatto che stampa `Ciao, mondo` ogni volta che H stamperebbe `no`. H_1 è rappresentato nella Figura 8.4.

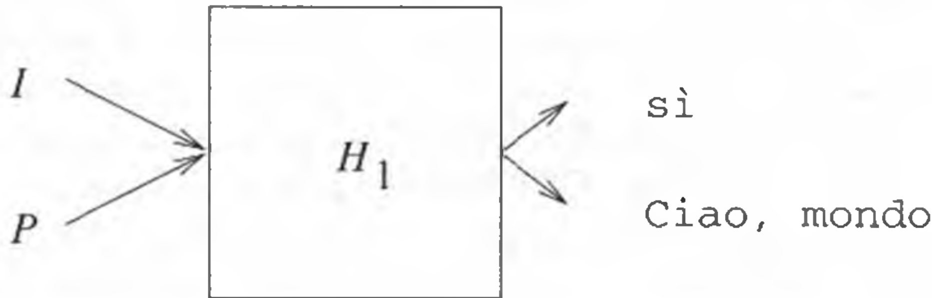


Figura 8.4 H_1 si comporta come H, ma stampa `Ciao, mondo` anziché `no`.

La trasformazione successiva è un po' più complicata. Si tratta essenzialmente dell'intuizione che permise ad Alan Turing di dimostrare il suo risultato di indecidibilità sulle macchine di Turing. Dato che il nostro interesse è limitato a programmi che prendono altri programmi come input e ne dicono qualcosa, limitiamo H_1 in due modi.

a) L'input è ridotto a P omettendo I.

b) Il nuovo programma determina che cosa farebbe P se l'input fosse il suo stesso codice, cioè che cosa farebbe H_1 avendo in input P sia come programma sia come I.

Le modifiche che dobbiamo apportare ad H_1 per produrre il programma H_2 suggerito nella Figura 8.5 sono le seguenti.

1. Per prima cosa H_2 legge l'intero input P e lo salva in un array A, allocato a tale scopo (ad esempio tramite la funzione `malloc`).[3]

2. Poi H_2 simula H_1, ma quando H_1 legge input da P o da I, H_2 legge dalla copia salvata in A. Per tenere traccia di quanto H_1 ha letto di P e di I, H_2 può servirsi di due cursori che segnano le posizioni in A.

[2] È molto probabile che il programma metta `no` in un'unica `printf`, ma potrebbe anche collocare la "n" in una `printf` e la "o" in un'altra.

[3] La funzione di sistema di UNIX `malloc` alloca un blocco di memoria di dimensione specificata nella chiamata. Si tratta di una funzione usata quando la quantità di memoria necessaria non può essere determinata finché non si esegue il programma, come nel caso di lettura di un input di lunghezza arbitraria. Di norma `malloc` viene chiamata più volte, a mano a mano che si legge l'input e quindi si crea un progressivo bisogno di spazio.

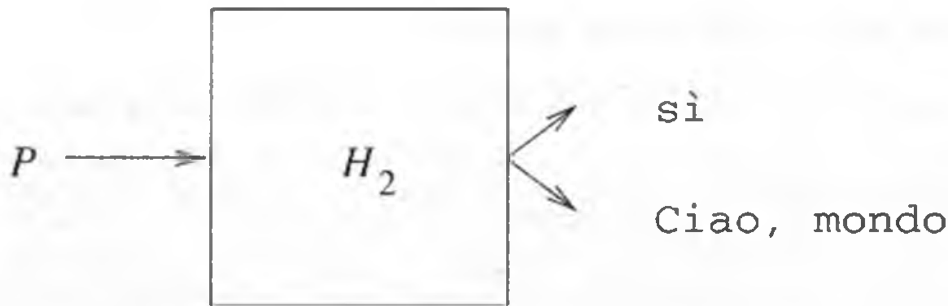


Figura 8.5 H_2 si comporta come H_1, ma usa il suo input P sia per P sia per I.

A questo punto possiamo dimostrare che H_2 non può esistere. Di conseguenza H_1 non esiste, così come non esiste H. Il nocciolo del ragionamento consiste nell'immaginare che cosa fa H_2 quando gli si dà se stesso come input, situazione illustrata nella Figura 8.6. Ricordiamo che H_2, dato un qualunque programma P come input, produce l'output `sì` se P stampa `Ciao, mondo` quando gli viene dato se stesso come input. Inoltre H_2 stampa `Ciao, mondo` se P, dato se stesso come input, non stampa `Ciao, mondo` come primo output.

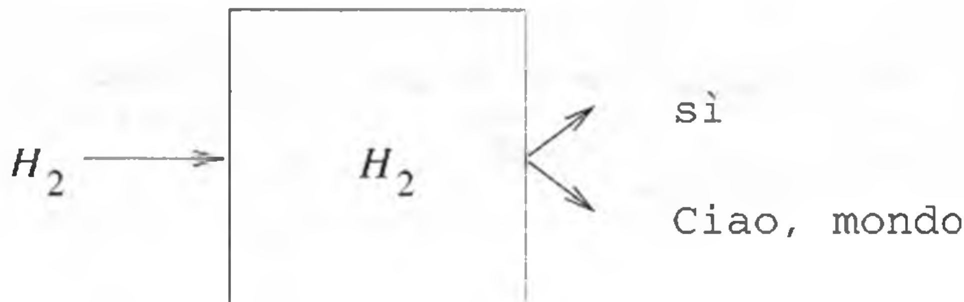


Figura 8.6 Che cosa fa H_2 quando ha come input se stesso?

Supponiamo che H_2 (il quadrato nella Figura 8.6) produca l'output `sì`. Allora H_2 nel quadrato afferma che il suo input H_2, quando riceve se stesso come input, stampa `Ciao, mondo` come primo output. Eppure abbiamo appena ipotizzato che il primo output di H_2 in questa situazione sia `sì` anziché `Ciao, mondo`.

Di conseguenza l'output di H_2 nella Figura 8.6 dovrebbe essere `Ciao, mondo`, dato che dev'essere uno dei due. Ma se H_2, ricevendo se stesso come input, stampa per primo `Ciao, mondo`, l'output del quadrato nella Figura 8.6 dev'essere `sì`. Qualunque sia l'output ipotizzato di H_2, possiamo dedurre che produca invece l'altro.

Si tratta di una situazione paradossale, e possiamo concludere che H_2 non esiste. Abbiamo quindi confutato l'ipotesi che H esista. In altre parole abbiamo dimostrato che nessun programma H può dire o no se un dato programma P con input I stampa `Ciao, mondo` come primo output.

8.1.3 Ridurre un problema a un altro

Abbiamo ora un problema – un dato programma con un dato input stampa per prima cosa `Ciao, mondo`? – che sappiamo essere irrisolvibile da un computer. Un problema che non può essere risolto da un computer è detto *indecidibile*. Daremo la definizione formale del termine "indecidibile" nel Paragrafo 9.3; per ora ne facciamo un uso informale. Supponiamo di voler determinare se un qualche altro problema sia o no risolvibile da un computer. Possiamo cercare di scrivere un programma per risolverlo, ma se non ci riusciamo, potremmo tentare di dimostrare che un simile programma non esiste.

Potremmo dimostrare l'indecidibilità di questo nuovo problema con una tecnica simile a quella usata per il problema ciao-mondo: assumiamo che esista un programma in grado di risolverlo e sviluppiamo un programma paradossale che deve fare due cose contraddittorie, come il programma H_2. In realtà, disponendo di un problema che sappiamo essere indecidibile, non abbiamo più bisogno di dimostrare l'esistenza di una situazione paradossale. Basta mostrare che se potessimo risolvere il nuovo problema, allora potremmo usare tale soluzione per risolvere il problema che già sappiamo essere indecidibile. La strategia è illustrata nella Figura 8.7; la tecnica è detta *riduzione* di P_1 a P_2.

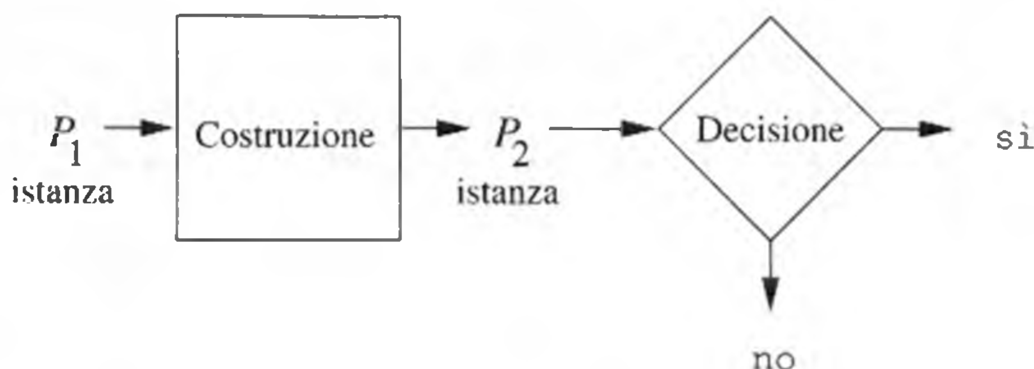


Figura 8.7 Se siamo in grado di risolvere il problema P_2, possiamo usare la soluzione per risolvere il problema P_1.

Supponiamo di sapere che il problema P_1 è indecidibile, e sia P_2 un nuovo problema di cui vogliamo dimostrare l'indecidibilità. Ipotizziamo che esista un programma, rappresentato nella Figura 8.7 da un rombo etichettato "decisione". Il programma stampa `sì` o `no` a seconda che l'istanza del problema P_2 che gli fa da input si trovi o no nel linguaggio del problema stesso.[4]

Per dimostrare l'indecidibilità del problema P_2 dobbiamo ideare una costruzione, rappresentata dal quadrato nella Figura 8.7, che converta le istanze di P_1 in istanze di P_2 che

[4]Ricordiamo che un problema è in realtà un linguaggio. Quando abbiamo affrontato la questione di decidere se un dato programma e un dato input danno come primo output `Ciao, mondo`, in effetti stavamo parlando di stringhe consistenti in un programma sorgente in C seguito da un file (o più file) di input che il programma legge. Questo insieme di stringhe è un linguaggio sull'alfabeto di caratteri ASCII.

Un computer può davvero farlo?

Esaminando un programma come quello della Figura 8.2 potremmo chiederci se effettivamente cerca controesempi all'ultimo teorema di Fermat. In un tipico computer gli interi hanno lunghezza pari a soli 32 bit; se il più piccolo controesempio coinvolgesse interi nell'ordine dei miliardi, si produrrebbe un errore di overflow prima di trovare la soluzione. Si può senz'altro sostenere che un computer con una memoria centrale di 128 megabyte e un disco di 30 gigabyte ha "solo" $256^{30128000000}$ stati ed è quindi un automa a stati finiti.

D'altra parte, trattare i computer come automi a stati finiti (o trattare il cervello umano come un automa a stati finiti, motivo per cui sono nati gli FA) è inutile. Il numero di stati coinvolti è così grande e i limiti sono così vaghi che è impossibile trarre conclusioni proficue. In effetti si può ragionevolmente pensare che l'insieme di stati di un computer sia espandibile arbitrariamente.

Per esempio è possibile rappresentare gli interi come liste di cifre di lunghezza arbitraria. Se la memoria è insufficiente, il programma può stampare la richiesta di smontare il disco, salvarlo e sostituirlo con un disco vuoto. Il computer potrebbe man mano stampare richieste di scambi tra tutti i dischi che gli sono necessari. Si tratterebbe di un programma molto più complesso di quello della Figura 8.2, ma la sua scrittura non andrebbe oltre le nostre capacità. Con simili espedienti qualsiasi programma può ovviare alle limitazioni sulla capacità della memoria o sulla dimensione degli interi o di altri dati.

diano la stessa risposta. In altre parole qualunque stringa nel linguaggio P_1 viene convertita in una stringa nel linguaggio P_2, e qualunque stringa sull'alfabeto di P_1 che *non* sia nel linguaggio P_1 viene convertita in una stringa che non è nel linguaggio P_2. Con questa costruzione possiamo risolvere P_1 nel modo seguente.

1. Data un'istanza di P_1, cioè data una stringa w che può essere o no nel linguaggio P_1, applichiamo l'algoritmo di costruzione che produce una stringa x.

2. Verifichiamo se x si trova in P_2 e diamo la stessa risposta su w e P_1.

Se w è in P_1, allora x è in P_2. Dunque l'algoritmo dice `sì`. Se w non è in P_1, allora x non è in P_2, e l'algoritmo dice `no`. Nell'uno o nell'altro caso dice comunque la verità su w. Dato che abbiamo ipotizzato che non esista alcun algoritmo che decide l'appartenenza di una stringa a P_1, abbiamo una dimostrazione per assurdo che l'algoritmo di decisione ipotizzato per P_2 non esiste; ossia P_2 è indecidibile.

L'importanza della direzione di una riduzione

È un errore comune quello di dimostrare che un problema P_2 è indecidibile riducendo P_2 a un altro problema indecidibile noto P_1, ossia provando l'enunciato "se P_1 è decidibile, allora P_2 è decidibile". Tale enunciato, per quanto senz'altro vero, è inutile perché poggia su un'ipotesi, "P_1 è decidibile", falsa.

L'unico modo per dimostrare l'indecidibilità di un nuovo problema P_2 è ridurre a P_2 un problema indecidibile noto P_1. In questo modo dimostriamo l'enunciato "se P_2 è decidibile, allora P_1 è decidibile". Il contronominale di tale enunciato è "se P_1 è indecidibile, allora P_2 è indecidibile". Sapendo che P_1 è indecidibile possiamo dedurre che P_2 è indecidibile.

Esempio 8.1 Applichiamo questo metodo per mostrare che la questione "il programma Q con input y chiama la funzione `foo`?" è indecidibile. Notiamo che Q può non avere la funzione `foo`, nel qual caso il problema è semplice; i casi critici si hanno quando Q ha una funzione `foo`, ma con input y può raggiungere o no una chiamata a `foo`. Poiché conosciamo un solo problema indecidibile, il ruolo di P_1 nella Figura 8.7 sarà affidato al problema ciao-mondo. P_2 sarà il *problema della chiamata* appena descritto. Supponiamo che esista un programma che risolve questo problema. Il nostro compito consiste nel definire un algoritmo che converta il problema ciao-mondo nel problema della chiamata.

In altre parole, dato un programma Q e il suo input y, dobbiamo costruire un programma R e un input z tali che R, con input z, chiami `foo` se e solo se Q con input y stampa `Ciao, mondo`. La costruzione non è difficoltosa.

1. Se Q ha una funzione chiamata `foo`, ne cambiamo il nome insieme con tutte le chiamate a tale funzione. Ovviamente il nuovo programma Q_1 fa esattamente ciò che fa Q.

2. Aggiungiamo a Q_1 una funzione `foo`. Tale funzione non fa niente e non viene mai chiamata. Il programma che ne risulta è Q_2.

3. Modifichiamo Q_2 in modo tale che ricordi i primi 11 caratteri che stampa e li salvi in un array globale A. Sia Q_3 il programma risultante.

4. Modifichiamo Q_3 in modo che quando esegue un'istruzione di output controlli l'array A per vedere se ha scritto 11 o più caratteri e, in tal caso, se i primi 11 sono proprio `Ciao, mondo`. Se è così, il nuovo programma chiama la funzione aggiunta al passo (2), denominata `foo`. Il programma risultante è R e l'input z è uguale a y.

Supponiamo che Q con input y stampi `Ciao, mondo` come primo output. Allora R, per come è stato costruito, chiama `foo`. Ma se Q con input y non stampa `Ciao, mondo` come primo output, R non chiama mai `foo`. Se possiamo decidere se R con input z chiama `foo`, allora sappiamo anche se Q con input y (ricordiamo che $y = z$) stampa `Ciao, mondo`. Sapendo che non esiste alcun algoritmo capace di decidere il problema ciao-mondo, e che i quattro passi della costruzione di R da Q potrebbero essere svolti da un programma che editi il codice sorgente, il nostro assunto sull'esistenza di un verificatore del problema della chiamata è falso. Tale programma non esiste e il problema è indecidibile. □

8.1.4 Esercizi

Esercizio 8.1.1 Definite riduzioni dal problema ciao-mondo a ognuno dei problemi elencati. Usate lo stile informale di questo paragrafo per descrivere trasformazioni di programma plausibili e non tenete conto dei limiti reali imposti dai computer concreti, come la dimensione massima di un file oppure la capacità della memoria.

*! a) Dati un programma e un input, il programma termina (cioè non entra in un ciclo infinito)?

b) Dati un programma e un input, il programma produce un output?

! c) Dati due programmi e un input, i programmi producono lo stesso output per l'input dato?

8.2 La macchina di Turing

La teoria dei problemi indecidibili non ha soltanto lo scopo di stabilire l'esistenza di questi problemi, un concetto di per sé intellettualmente stimolante, ma anche di fornire una guida ai programmatori su quanto è possibile realizzare mediante la programmazione. La teoria ha anche importanti riflessi pragmatici nella trattazione di problemi che, sebbene decidibili, richiedono considerevoli quantità di tempo per essere risolti (come vedremo nel Capitolo 10). Questi problemi, detti "intrattabili", tendono a presentare ai programmatori e ai progettisti di sistemi difficoltà maggiori rispetto ai problemi indecidibili. Infatti, mentre i problemi indecidibili lo sono di solito palesemente, e nella pratica si cerca di risolverli di rado, i problemi intrattabili si presentano ogni giorno. Inoltre essi si prestano frequentemente a piccole modifiche nei requisiti o a soluzioni euristiche. Il progettista si trova quindi a dover decidere se un problema appartenga alla classe dei problemi intrattabili e, se sì, che cosa occorra fare.

Per dimostrare se un problema è indecidibile o intrattabile ci servono strumenti adeguati. La tecnica presentata nel Paragrafo 8.1 è utile per questioni relative a programmi,

ma non si può adattare con facilità a problemi di altro genere. Per esempio sarebbe molto difficile ridurre il problema ciao-mondo alla questione sull'ambiguità di una grammatica.

Ne segue che bisogna ricostruire la teoria dell'indecidibilità fondandosi non su programmi in C o in altri linguaggi, bensì su un modello molto semplice di computer, chiamato "macchina di Turing". In sostanza si tratta di un automa a stati finiti con un nastro di lunghezza infinita su cui può leggere o scrivere dati. Rispetto ai programmi, la macchina di Turing, come modello di ciò che può essere computato, ha il vantaggio della semplicità perché è possibile rappresentarne precisamente la configurazione ricorrendo a una notazione simile alle ID di un PDA. Anche un programma C ha uno stato, formato dalle variabili relative alle funzioni che sono state chiamate in sequenza, ma la notazione per descriverlo è di gran lunga troppo complessa per poterla usare in dimostrazioni formali.

Con la notazione delle macchine di Turing dimostreremo che certi problemi, apparentemente distanti dalla programmazione tradizionale, sono indecidibili. Per esempio nel Paragrafo 9.4 mostreremo che il "Problema della Corrispondenza di Post", una semplice questione che concerne due sequenze di stringhe, è indecidibile; questo problema permette di dimostrare facilmente l'indecidibilità di questioni riguardanti le grammatiche, per esempio l'ambiguità. In modo analogo, quando presenteremo i problemi intrattabili, vedremo che alcune questioni apparentemente lontane dalla computazione (per esempio la soddisfacibilità di formule booleane) sono intrattabili.

8.2.1 La ricerca della soluzione a tutte le domande matematiche

Agli inizi del '900 il matematico D. Hilbert si chiese se fosse possibile trovare un algoritmo in grado di stabilire la verità o la falsità di qualsiasi proposizione matematica. In particolare si domandò se esistesse un modo per determinare la veridicità di una formula del calcolo dei predicati del primo ordine applicata agli interi. Poiché il calcolo dei predicati del primo ordine per gli interi è sufficientemente potente da esprimere enunciati come "la tal grammatica è ambigua" oppure "il tal programma stampa `Ciao, mondo`", se Hilbert avesse avuto successo questi problemi avrebbero gli algoritmi che ora sappiamo non esistono.

Nel 1931 K. Gödel pubblicò il suo famoso teorema di incompletezza. Egli definì una formula nel calcolo dei predicati applicata agli interi secondo la quale la formula stessa non poteva essere né dimostrata né confutata all'interno del calcolo dei predicati. La tecnica di Gödel somiglia alla costruzione del programma autocontraddittorio H_2 presentata nel Paragrafo 8.1.2, ma tratta funzioni sugli interi anziché programmi C.

Il calcolo dei predicati non era l'unica nozione a disposizione dei matematici come modello di computazione universale. In effetti il calcolo dei predicati, essendo dichiarativo piuttosto che computazionale, doveva competere con una serie di notazioni, incluse le "funzioni ricorsive parziali" (una notazione simile ai linguaggi di programmazione) e con altre notazioni analoghe. Nel 1936 A.M. Turing propose la macchina di Turing come

modello di computazione universale. Si tratta di un modello più simile a un computer che a un programma, anche se i calcolatori elettronici, o anche elettromeccanici, sarebbero arrivati alcuni anni dopo (e Turing stesso si occupò della costruzione di una macchina di questo tipo durante la seconda guerra mondiale).

È interessante notare che tutte le proposte di modello di computazione hanno la stessa potenza, cioè calcolano le stesse funzioni o riconoscono gli stessi linguaggi. L'assunto indimostrabile che qualunque modo generale di computare permette di computare solo le funzioni ricorsive parziali (oppure ciò che le macchine di Turing o i computer moderni possono calcolare, il che è equivalente) è detto *ipotesi di Church* (dal nome del logico A. Church) o *tesi di Church-Turing*.

8.2.2 Notazione per la macchina di Turing

Possiamo rappresentare la macchina di Turing come nella Figura 8.8. La macchina consiste di un *controllo finito*, che può trovarsi in uno stato, scelto in un insieme finito. C'è un *nastro* diviso in caselle, o *celle*; ogni cella può contenere un simbolo scelto in un insieme finito.

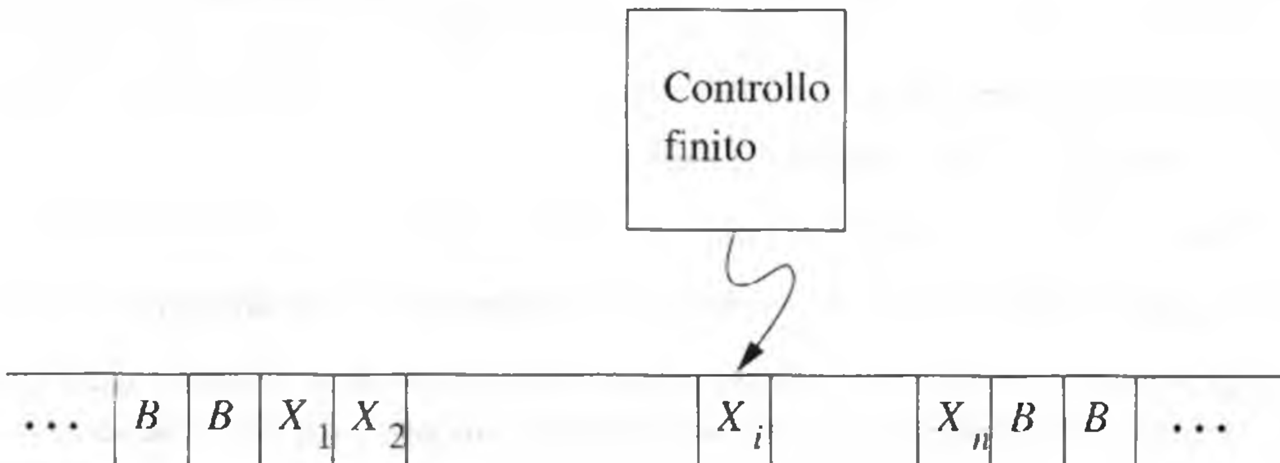


Figura 8.8 Una macchina di Turing.

L'*input*, una stringa di lunghezza finita formata da simboli scelti dall'*alfabeto di input*, viene inizialmente posto sul nastro. Tutte le altre celle, che si estendono senza limiti a sinistra e a destra, contengono all'inizio un simbolo speciale, detto *blank*. Il blank è un *simbolo di nastro*, ma non di input; oltre ai simboli di input e al blank, ci possono essere anche altri simboli di nastro.

Una *testina* mobile è collocata su una delle celle del nastro. Diremo che la macchina di Turing *guarda* quella cella. All'inizio la testina si trova sulla cella più a sinistra rispetto a quelle che contengono l'input.

Una *mossa* della macchina di Turing è una funzione dello stato del controllo e del simbolo di nastro guardato dalla testina. In una mossa la macchina di Turing compie tre azioni.

1. Cambia stato. Lo stato successivo può coincidere con lo stato corrente.
2. Scrive un simbolo di nastro nella cella che guarda. Questo simbolo sostituisce quello che si trovava in precedenza nella cella. Il nuovo simbolo può essere lo stesso che si trova già nella cella.
3. Muove la testina verso sinistra oppure verso destra. Nel nostro formalismo richiediamo un movimento: la testina non può rimanere ferma. Si tratta di una limitazione che non incide sulla capacità computazionale della macchina, dato che qualsiasi sequenza di mosse con testina fissa si può riassumere, insieme con la mossa successiva della testina, in un cambio di stato, un nuovo simbolo di nastro e un movimento a sinistra o a destra.

La notazione formale che useremo per una *macchina di Turing* (TM, *Turing Machine*) è simile a quella usata per gli automi a stati finiti o PDA. Descriviamo una TM come la settupla

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

i cui componenti hanno i seguenti significati.

Q: l'insieme finito degli *stati* del controllo.

Σ: l'insieme finito dei *simboli di input*.

Γ: l'insieme completo dei *simboli di nastro*; Σ è sempre un sottoinsieme di Γ.

δ: la *funzione di transizione*. Gli argomenti di $\delta(q, X)$ sono uno stato q e un simbolo di nastro X. Il valore di $\delta(q, X)$, se definito, è una tripla (p, Y, D), dove:

1. p, elemento di Q, è lo stato successivo
2. Y è il simbolo di Γ scritto nella cella guardata, al posto di qualunque simbolo vi fosse
3. D è una *direzione*, L o R – rispettivamente per "left" (sinistra) e "right" (destra) –, e segnala la direzione in cui si muove la testina.

q_0: lo *stato iniziale* del controllo; è un elemento di Q.

B: il simbolo detto *blank*. Si trova in Γ ma non in Σ, cioè non è un simbolo di input. Inizialmente il blank compare in tutte le celle tranne quelle (finite) che contengono i simboli di input.

F: l'insieme degli stati *finali* o *accettanti*, un sottoinsieme di Q.

8.2.3 Descrizioni istantanee delle macchine di Turing

Per descrivere formalmente che cosa fa una macchina di Turing dobbiamo sviluppare una notazione per le configurazioni o *descrizioni istantanee* (ID, *Instantaneous Descriptions*), come quella sviluppata per i PDA. Poiché in linea di principio una TM ha un nastro infinitamente lungo, potremmo pensare che sia impossibile descrivere in termini concisi le sue configurazioni. In realtà, dopo un numero finito di mosse, la TM può aver visitato solo un numero finito di celle, anche se il numero di celle visitate può crescere oltre qualunque limite finito. Perciò in ogni ID esistono un prefisso e un suffisso infiniti di celle che non sono mai state visitate. Tutte queste celle devono contenere un blank oppure uno dei simboli di input, che sono in numero finito. In una ID mostriamo dunque solo le celle tra il simbolo diverso dal blank più a sinistra e quello più a destra. In casi particolari, quando la testina guarda uno dei blank in testa o in coda, anche un numero finito di blank, a sinistra o a destra della porzione non bianca del nastro dev'essere incluso nella ID.

Oltre al nastro dobbiamo rappresentare anche il controllo e la posizione della testina. A tal fine incorporiamo lo stato nel nastro e lo poniamo immediatamente a sinistra della cella guardata. Per eliminare ogni ambiguità nella stringa composta da nastro e stato, dobbiamo assicurarci di non usare come stato un simbolo che sia anche un simbolo di nastro. In ogni caso, dal momento che l'operazione della TM non dipende dai nomi degli stati, è facile cambiarli in modo che non abbiano nulla in comune con i simboli di nastro. Useremo dunque la stringa $X_1X_2 \cdots X_{i-1}qX_iX_{i+1} \cdots X_n$ per rappresentare una ID in cui:

1. q è lo stato della macchina di Turing
2. la testina guarda l'i-esimo simbolo da sinistra
3. $X_1X_2 \cdots X_n$ è la porzione del nastro tra il simbolo diverso dal blank più a sinistra e quello più a destra. Eccezionalmente, se la testina è a sinistra del simbolo diverso dal blank più a sinistra o a destra di quello più a destra, allora un prefisso o suffisso di $X_1X_2 \cdots X_n$ sarà costituito da blank, e i sarà rispettivamente 1 o n.

Descriviamo le mosse di una macchina di Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ mediante la notazione $\underset{M}{\vdash}$, già usata per i PDA. Quando la TM M è implicita usiamo solo $\vdash$ per denotare le mosse. Come al solito $\overset{*}{\underset{M}{\vdash}}$, o solo $\overset{*}{\vdash}$, indica zero, una, o più mosse della TM M.

Supponiamo che $\delta(q, X_i) = (p, Y, L)$, cioè la prossima mossa è verso sinistra. Allora

$$X_1X_2 \cdots X_{i-1}qX_iX_{i+1} \cdots X_n \underset{M}{\vdash} X_1X_2 \cdots X_{i-2}pX_{i-1}YX_{i+1} \cdots X_n$$

La notazione riflette il passaggio allo stato p e il fatto che ora la testina si trova sulla cella $i - 1$. Ci sono due eccezioni importanti.

1. Se $i = 1$, allora M si muove verso il blank a sinistra di X_1. In tal caso,

$$qX_1X_2\cdots X_n \underset{M}{\vdash} pBYX_2\cdots X_n$$

2. Se $i = n$ e $Y = B$, allora il simbolo B scritto su X_n si unisce alla sequenza di blank in coda e non compare nella ID seguente. Perciò

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-2}pX_{n-1}$$

Supponiamo ora che $\delta(q, X_i) = (p, Y, R)$, cioè la prossima mossa è verso destra. Allora

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-1}YpX_{i+1}\cdots X_n$$

La notazione riflette il fatto che la testina si è mossa verso la cella $i + 1$. Ancora una volta ci sono due eccezioni importanti.

1. Se $i = n$, allora la cella $i + 1$-esima contiene un blank e non faceva parte della ID precedente. Abbiamo perciò

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-1}YpB$$

2. Se $i = 1$ e $Y = B$, allora il simbolo B scritto su X_1 si unisce alla sequenza infinita di blank in testa e non compare nella ID seguente. Perciò,

$$qX_1X_2\cdots X_n \underset{M}{\vdash} pX_2\cdots X_n$$

Esempio 8.2 Definiamo una macchina di Turing e vediamo come si comporta su un input tipico. La TM che costruiamo accetta il linguaggio $\{0^n1^n \mid n \geq 1\}$. Inizialmente alla macchina viene data sul nastro una sequenza finita di 0 e 1, preceduta e seguita da blank. La TM cambia in modo alternato uno 0 in X e un 1 in Y finché tutti gli 0 e gli 1 sono abbinati.

Più precisamente, partendo dall'estremità sinistra dell'input, cambia uno 0 in X e si muove verso destra su tutti gli 0 e Y che vede finché arriva a un 1. Cambia l'1 in Y e si muove verso sinistra, sulle Y e gli 0, finché trova una X. A questo punto cerca uno 0 immediatamente a destra; se ne trova uno, lo cambia in X e ripete il processo, trasformando un 1 corrispondente in Y.

Se l'input costituito dai caratteri diversi dal blank non si trova in 0^*1^*, la TM prima o poi non ha una mossa successiva e termina senza accettare. Se invece conclude la trasformazione di tutti gli 0 in X nello stesso giro in cui cambia l'ultimo 1 in Y, allora ha stabilito che l'input è della forma 0^n1^n e accetta. La specifica formale della TM M è

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

Stato	Simbolo 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figura 8.9 Una macchina di Turing che accetta $\{0^n1^n \mid n \geq 1\}$.

dove δ è data dalla tabella nella Figura 8.9.

Mentre M svolge la sua computazione, la porzione del nastro su cui è passata la testina è sempre una sequenza di simboli descritta dall'espressione regolare $\mathbf{X^*0^*Y^*1^*}$. In altre parole ci sono degli 0 che sono stati trasformati in X, seguiti da 0 che non sono stati trasformati. Ci sono poi degli 1 che sono stati trasformati in Y e 1 che non sono stati ancora trasformati. Possono poi esserci altri 0 e 1.

Lo stato q_0 è lo stato iniziale; M rientra in q_0 ogni volta che ritorna allo 0 residuo più a sinistra. Se M si trova nello stato q_0 e guarda uno 0, la regola in alto a sinistra nella Figura 8.9 impone di andare nello stato q_1, trasformare lo 0 in X e muoversi verso destra. Nello stato q_1, M continua a muoversi verso destra su tutti gli 0 e le Y in cui si imbatte sul nastro, rimanendo in q_1. Se M legge una X o una B, muore. Se invece legge un 1 quando si trova nello stato q_1, lo trasforma in Y, entra nello stato q_2 e comincia a muoversi verso sinistra.

Nello stato q_2, M si muove verso sinistra sugli 0 e le Y, rimanendo in q_2. Quando raggiunge la X più a destra, che segna l'estremità destra del blocco di 0 già trasformati in X, M ritorna nello stato q_0 e si muove verso destra. Possiamo individuare due casi.

1. Se ora M vede uno 0, ripete il ciclo di abbinamento appena descritto.

2. Se M vede una Y, tutti gli 0 sono stati trasformati in X. Se tutti gli 1 sono stati trasformati in Y, allora l'input era di forma 0^n1^n ed M deve accettare. Di conseguenza M entra nello stato q_3 e comincia a muoversi verso destra, sulle Y. Se il primo simbolo diverso da una Y che M vede è un blank, allora il numero di 0 e di 1 era lo stesso; dunque M entra nello stato q_4 e accetta. D'altra parte, se M incontra un altro 1, allora ci sono troppi 1 ed M muore senza accettare. M muore anche se incontra uno 0 (input della forma sbagliata).

Presentiamo un esempio di computazione accettante da parte di M. Il suo input è 0011. Inizialmente M si trova nello stato q_0 e guarda il primo 0, cioè la ID iniziale di M è q_00011. L'intera sequenza di mosse di M è:

$$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash q_2 X0Y1 \vdash$$
$$Xq_0 0Y1 \vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY \vdash$$
$$XXq_0 YY \vdash XXYq_3 Y \vdash XXYYq_3 B \vdash XXYYBq_4 B$$

Come secondo esempio consideriamo che cosa fa M sull'input 0010, che non si trova nel linguaggio accettato.

$$q_0 0010 \vdash Xq_1 010 \vdash X0q_1 10 \vdash Xq_2 0Y0 \vdash q_2 X0Y0 \vdash$$
$$Xq_0 0Y0 \vdash XXq_1 Y0 \vdash XXYq_1 0 \vdash XXY0q_1 B$$

Il comportamento di M su 0010 è simile a quello su 0011 finché nella ID $XXYq_1 0$ M guarda l'ultimo 0 per la prima volta. M deve prima muoversi verso destra restando nello stato q_1, che la porta alla ID $XXY0q_1 B$. Ma nello stato q_1 M non ha mosse sul simbolo di nastro B; di conseguenza M muore e non accetta l'input. □

8.2.4 Diagrammi di transizione per le macchine di Turing

Possiamo rappresentare le transizioni di una macchina di Turing graficamente come abbiamo fatto per i PDA. Un *diagramma di transizione* consiste in un insieme di nodi corrispondenti agli stati della TM. Un arco dallo stato q allo stato p è etichettato da uno o più oggetti della forma X/YD, dove X e Y sono simboli di nastro e D è una direzione, L o R. In altre parole, ogni volta che $\delta(q, X) = (p, Y, D)$, troviamo l' etichetta X/YD sull'arco da q a p. Nei diagrammi la direzione D è rappresentata graficamente da $\leftarrow$ per "sinistra" e $\rightarrow$ per "destra".

Come per altri tipi di diagrammi di transizione, rappresentiamo lo stato iniziale con la parola "Start" e una freccia che entra in tale stato. Gli stati accettanti sono segnalati dal doppio cerchio. Dunque l'unica informazione sulla TM che non si ricava direttamente dal diagramma è il simbolo usato per il blank. Supponiamo che il simbolo sia B, salvo indicazione contraria.

Esempio 8.3 La Figura 8.10 mostra il diagramma di transizione per la macchina di Turing dell'Esempio 8.2, la cui funzione di transizione è stata data nella Figura 8.9. □

Esempio 8.4 Mentre oggi è più comodo pensare alle macchine di Turing come a dispositivi per riconoscere linguaggi o, il che è equivalente, per risolvere problemi, in origine Turing considerava la sua macchina come un calcolatore di funzioni a valori interi. Nel suo schema gli interi erano rappresentati in codice unario, come blocchi di un singolo carattere, e la macchina computava cambiando le lunghezze dei blocchi o costruendo nuovi blocchi altrove sul nastro. In questo semplice esempio mostreremo come una macchina di Turing può calcolare la funzione $\dot{-}$, detta *monus* o *sottrazione propria* e definita da $m \dot{-} n = \max(m - n, 0)$. In altre parole $m \dot{-} n$ è $m - n$ se $m \geq n$ e 0 se $m < n$.

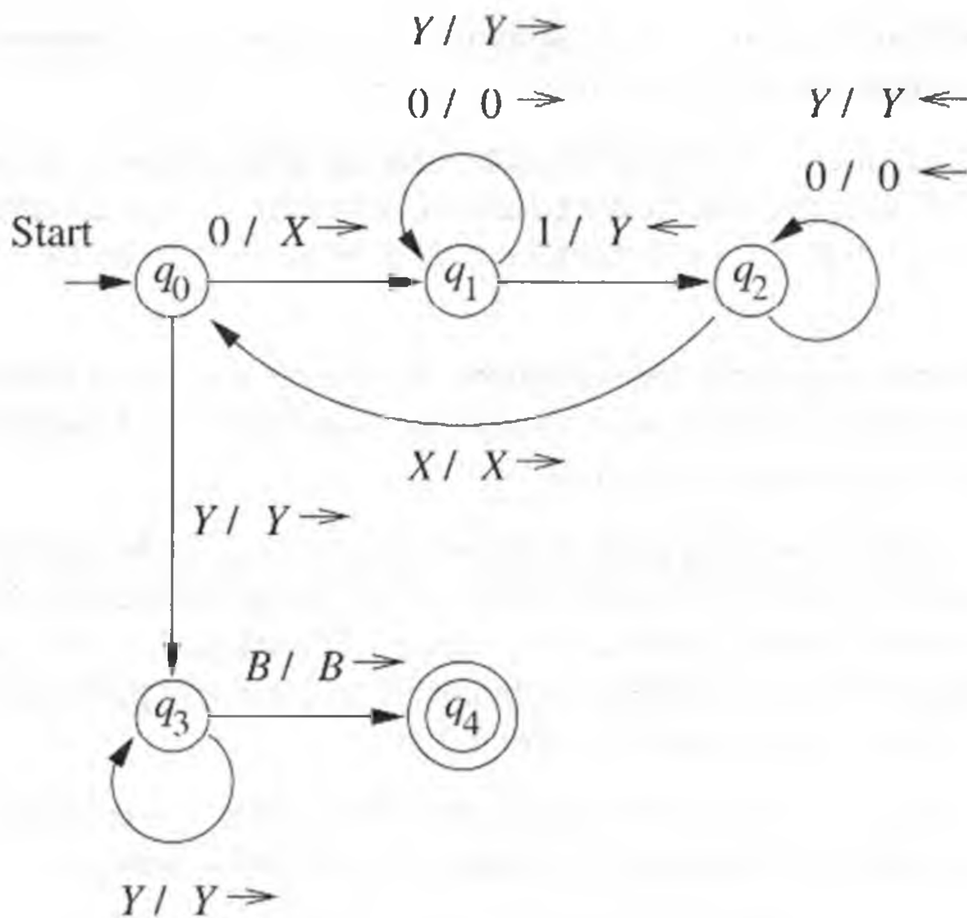


Figura 8.10 Diagramma di transizione per una TM che accetta stringhe della forma $0^n 1^n$.

Una TM che compie tale operazione è specificata da

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B)$$

Poiché questa TM non viene usata per accettare l'input, abbiamo omesso il settimo componente, che è l'insieme di stati accettanti. M parte con un nastro su cui è presente la stringa di input $0^m 1 0^n$ preceduta e seguita da blank. M si arresta con $0^{m \dot{-} n}$ sul nastro, circondato da blank.

Lo schema di funzionamento che viene ripetuto è il seguente: M trova lo 0 residuo più a sinistra e lo sostituisce con un blank. Si sposta allora verso destra, alla ricerca di un 1. Dopo aver trovato un 1, continua a destra finché non arriva a uno 0 e lo sostituisce con un 1. M torna allora verso sinistra cercando lo 0 più a sinistra: lo identifica quando si imbatte per la prima volta in un blank e poi muove di una cella verso destra. L'iterazione si ferma in uno dei casi seguenti.

1. Cercando uno 0 a destra M trova un blank. Allora gli n 0 in $0^m 1 0^n$ sono stati tutti trasformati in 1 e $n + 1$ degli m 0 sono stati trasformati in B. M sostituisce gli

$n + 1$ simboli 1 con uno 0 ed n simboli B, lasciando $m - n$ simboli 0 sul nastro. Poiché in questo caso $m \geq n$, $m - n = m \dot{-} n$.

2. Cominciando il ciclo, M non riesce a trovare uno 0 da mutare in un blank, dato che i primi m 0 sono già stati trasformati in B. Allora $n \geq m$, e dunque $m \dot{-} n = 0$. M sostituisce tutti gli 1 e 0 residui con B e finisce con un nastro completamente vuoto.

La Figura 8.11 fornisce le regole della funzione di transizione δ, di cui abbiamo la rappresentazione grafica come diagramma di transizione nella Figura 8.12. Ecco un compendio del ruolo svolto da ognuno dei sette stati.

q_0: è lo stato che dà avvio al ciclo e che lo interrompe quando è opportuno. Se M guarda uno 0, il ciclo dev'essere ripetuto. Lo 0 viene rimpiazzato da B, la testina si muove verso destra e si entra nello stato q_1. D'altra parte, se M guarda 1, allora sono state fatte tutte le possibili combinazioni tra i due gruppi di 0 sul nastro, ed M va nello stato q_5 per riempire il nastro di blank.

q_1: in questo stato M muove verso destra attraverso il blocco iniziale di 0, alla ricerca dell'1 più a sinistra. Dopo averlo trovato, M passa nello stato q_2.

q_2: M si muove verso destra saltando gli 1 finché non trova uno 0, che trasforma in 1, per poi volgersi a sinistra ed entrare nello stato q_3. È anche possibile che non ci siano 0 dopo il blocco di 1. In tal caso M trova un blank nello stato q_2. Siamo nel caso (1) descritto sopra: n simboli 0 nel secondo blocco sono stati usati per cancellare n degli m simboli 0 nel primo blocco, e la sottrazione è completa. M entra nello stato q_4, il cui scopo è convertire gli 1 sul nastro in blank.

q_3: M si muove verso sinistra saltando gli 0 e gli 1 finché trova un blank. Quando trova B si muove verso destra e ritorna nello stato q_0, dando nuovamente avvio al ciclo.

q_4: qui la sottrazione è completa, ma uno 0 non appaiato nel primo blocco è stato trasformato erroneamente in un B. M si muove dunque verso sinistra trasformando gli 1 in B finché si imbatte in un B sul nastro. Ritrasforma il B in 0 ed entra nello stato q_6, dove si arresta.

q_5: si entra nello stato q_5 a partire da q_0 quando tutti gli 0 del primo blocco sono stati trasformati in B. In questo caso, descritto in (2), il risultato della sottrazione è 0. M trasforma tutti gli 0 e gli 1 residui in B ed entra nello stato q_6.

q_6: l'unico scopo di questo stato è permettere a M di arrestarsi quando ha finito il suo lavoro. Se la sottrazione fosse stata una subroutine di una funzione più complessa, allora q_6 avrebbe iniziato il passo successivo della computazione più ampia.

$\sqsupset$

Stato	Simbolo 0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	–
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	–
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	–	–	–

Figura 8.11 Una macchina di Turing che calcola la funzione di sottrazione propria.

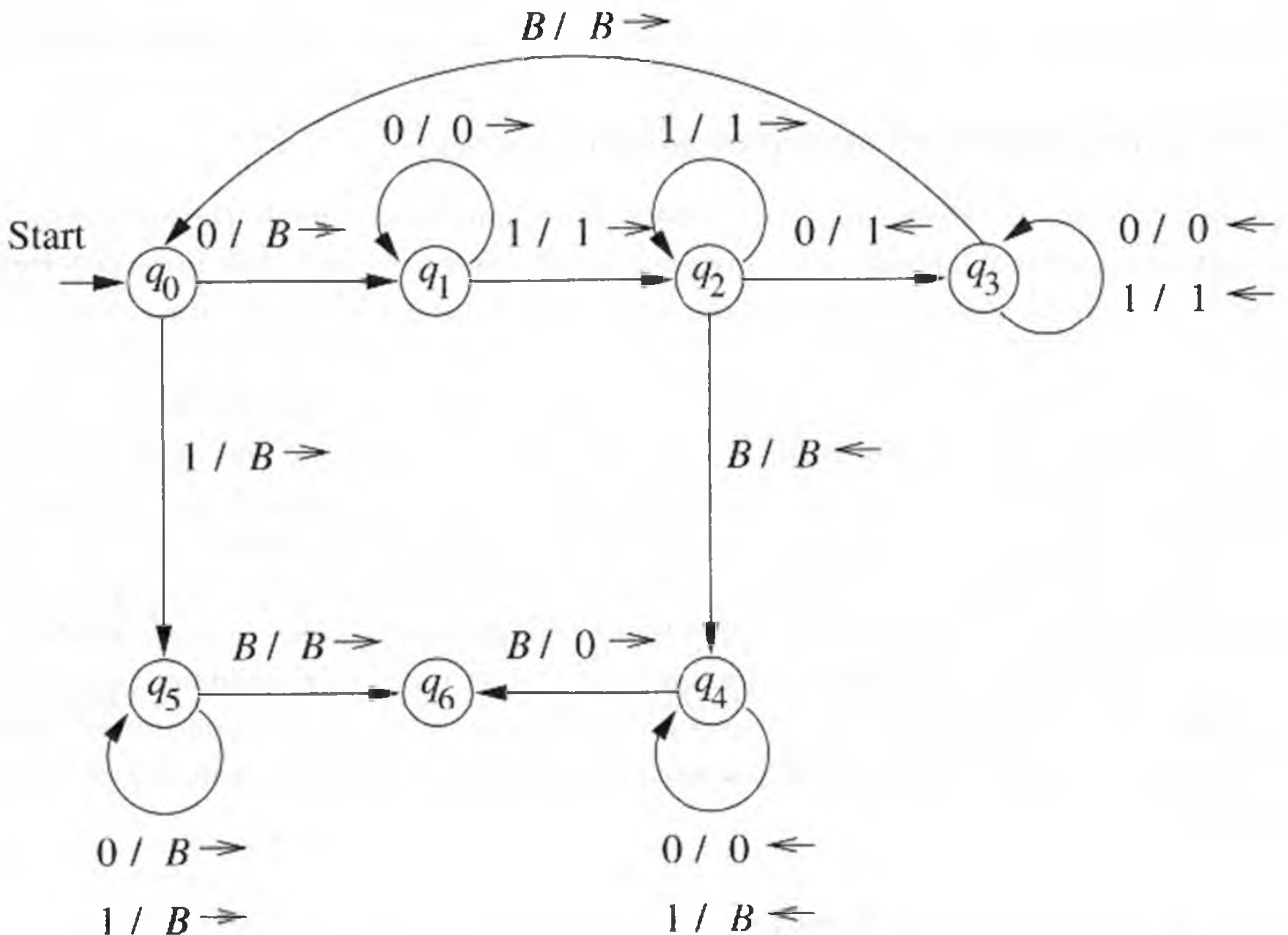


Figura 8.12 Il diagramma di transizione per la TM dell'Esempio 8.4.

Convenzioni notazionali per le macchine di Turing

I simboli usati normalmente per le macchine di Turing sono simili a quelli già visti per gli altri tipi di automa.

1. Le prime lettere minuscole dell'alfabeto indicano i simboli di input.
2. Le lettere maiuscole, di solito in prossimità della fine dell'alfabeto, sono usate per i simboli di nastro che possono essere o no simboli di input.
3. Le ultime lettere minuscole dell'alfabeto indicano stringhe di simboli di input.
4. Le lettere greche indicano stringhe di simboli di nastro.
5. Le lettere q, p e quelle circostanti indicano gli stati.

8.2.5 Il linguaggio di una macchina di Turing

Abbiamo descritto in termini intuitivi il modo in cui una macchina di Turing accetta un linguaggio. La stringa di input viene posta sul nastro e la testina parte dal simbolo di input più a sinistra. Se la TM entra in uno stato accettante, allora l'input è accettato, altrimenti no.

In termini formali, sia $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ una macchina di Turing. Allora $L(M)$ è l'insieme di stringhe w in Σ^* tale che $q_0 w \overset{*}{\vdash} \alpha p \beta$ per uno stato p in F e qualunque stringa di nastro α e β. Abbiamo ipotizzato questa definizione nel discutere la macchina di Turing dell'Esempio 8.2, che accetta le stringhe della forma $0^n 1^n$.

I linguaggi che possiamo accettare usando una macchina di Turing sono spesso denominati ***linguaggi ricorsivamente enumerabili*** o linguaggi RE (*Recursively Enumerable*). Il termine "ricorsivamente enumerabile" deriva da formalismi computazionali che precedono cronologicamente la macchina di Turing, ma che definiscono la stessa classe di linguaggi o funzioni aritmetiche. Vedremo le origini del termine in un *excursus* del Paragrafo 9.2.1.

8.2.6 Le macchine di Turing e l'arresto

Per le macchine di Turing si usa comunemente un'altra nozione di "accettazione": l'accettazione per arresto. Diremo che una TM *si arresta* se entra in uno stato q guardando un simbolo di nastro X e non ci sono mosse in questa situazione; cioè $\delta(q, X)$ è indefinito.

Esempio 8.5 La macchina di Turing M dell'Esempio 8.4 non è stata definita per accettare un linguaggio. L'abbiamo vista piuttosto nel suo aspetto di calcolo di una funzione aritmetica. Notiamo però che M si arresta su tutte le stringhe di 0 e 1 perché, a prescindere da quali stringhe trova sul nastro, finisce per cancellare il secondo gruppo di 0, se lo trova, a fronte del primo gruppo di 0. Di conseguenza deve raggiungere lo stato q_6 e arrestarsi. □

Possiamo sempre assumere che una TM si arresti se accetta. In altre parole, senza cambiare il linguaggio accettato, possiamo rendere $\delta(q, X)$ indefinito per ogni q accettante. In generale, senza specificarlo esplicitamente,

- assumiamo che una TM si arresti sempre quando si trova in uno stato accettante.

Purtroppo non è sempre possibile richiedere che una TM si arresti, anche se non accetta. I linguaggi per cui esiste una macchina di Turing che prima o poi si arresta, indipendentemente dall'accettare o no, sono detti *ricorsivi*, e ne studieremo le proprietà a partire dal Paragrafo 9.2.1. Le macchine di Turing che si arrestano sempre, a prescindere dal fatto che accettino o no, sono un buon modello di "algoritmo". Se esiste un algoritmo per risolvere un dato problema, diciamo che il problema è "decidibile". Le TM che si arrestano sempre hanno dunque un posto importante nella teoria della decidibilità (Capitolo 9).

8.2.7 Esercizi

Esercizio 8.2.1 Scrivete le ID della macchina di Turing della Figura 8.9 quando il nastro di input contiene:

* a) 00

b) 000111

c) 00111.

! **Esercizio 8.2.2** Definite una macchina di Turing per ognuno dei seguenti linguaggi.

* a) L'insieme delle stringhe con un numero uguale di 0 e di 1.

b) $\{a^n b^n c^n \mid n \geq 1\}$.

c) $\{ww^R \mid w$ è una qualunque stringa di 0 e 1$\}$.

Esercizio 8.2.3 Definite una macchina di Turing che prenda come input un numero N e vi aggiunga 1 in binario. Per la precisione il nastro contiene inizialmente un \$ seguito da N in binario. All'inizio la testina guarda il \$ nello stato q_0. La TM dovrebbe arrestarsi

con $N+1$, in binario, sul nastro, guardando il simbolo più a sinistra di $N+1$, nello stato q_f. Se necessario potete cancellare il \$ nel corso del calcolo. Per esempio $q_0\$10011 \overset{*}{\vdash} \$q_f10100$ e $q_0\$11111 \overset{*}{\vdash} q_f100000$.

a) Scrivete le transizioni della vostra macchina di Turing e spiegate lo scopo di ciascuno stato.

b) Mostrate la sequenza di ID della vostra TM quando le viene dato l'input \$111.

*! **Esercizio 8.2.4** In questo esercizio esploriamo l'equivalenza tra la computazione di funzioni e il riconoscimento di linguaggi per le macchine di Turing. Per semplicità consideriamo solo funzioni da interi non negativi a interi non negativi, ma le idee soggiacenti a questo problema si applicano a qualsiasi funzione computabile. Ecco le due definizioni cruciali.

- Il *grafo* di una funzione f sia l'insieme di tutte le stringhe della forma $[x, f(x)]$, dove x è un intero non negativo in binario e $f(x)$ è il valore della funzione f con argomento x, scritto in binario.
- Si dice che una macchina di Turing *computa* la funzione f se, partendo da un intero non negativo x sul nastro, in binario, si arresta (in qualunque stato) con $f(x)$, in binario, sul nastro.

Rispondete ai seguenti quesiti con costruzioni informali, ma chiare.

a) Mostrate come, data una TM che computa f, si può costruire una TM che accetta il grafo di f come linguaggio.

b) Mostrate come, data una TM che accetta il grafo di f, si può costruire una TM che computa f.

c) Si dice che una funzione è *parziale* se può essere indefinita su certi argomenti. Se estendiamo le idee di questo esercizio alle funzioni parziali, non richiediamo che la TM che computa f si arresti qualora il suo input x sia uno degli interi per cui il valore $f(x)$ non è definito. Le vostre costruzioni per le parti (a) e (b) sono valide se la funzione f è parziale? In caso negativo spiegate come modificare la costruzione in modo che funzioni.

Esercizio 8.2.5 Considerate la seguente macchina di Turing:

$$M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_f\})$$

Descrivete in termini informali, ma chiari, il linguaggio $L(M)$ se δ è specificata come segue:

* a) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_0, 0, R)$; $\delta(q_1, B) = (q_f, B, R)$

b) $\delta(q_0, 0) = (q_0, B, R)$; $\delta(q_0, 1) = (q_1, B, R)$; $\delta(q_1, 1) = (q_1, B, R)$; $\delta(q_1, B) = (q_f, B, R)$

! c) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_2, 0, L)$; $\delta(q_2, 1) = (q_0, 1, R)$; $\delta(q_1, B) = (q_f, B, R)$.

8.3 Tecniche di programmazione per le macchine di Turing

Vogliamo ora chiarire come si può usare una macchina di Turing per calcolare in modo simile a un calcolatore convenzionale. L'obiettivo è convincere il lettore che una TM ha la stessa capacità di un tipico calcolatore. In particolare vedremo che la macchina di Turing è in grado di svolgere elaborazioni su altre macchine di Turing in modo analogo al programma del Paragrafo 8.1.2, che esamina altri programmi. Proprio questa capacità "introspettiva" delle macchine di Turing e dei programmi ci permette di dimostrare che certi problemi sono indecidibili.

Per chiarire le possibilità di una TM, presenteremo esempi di interpretazioni del nastro e del controllo finito. Questi espedienti non estendono il modello fondamentale, ma rendono più comoda la notazione. In seguito ce ne serviremo per simulare modelli estesi di macchina di Turing con nuove caratteristiche – per esempio più di un nastro – tramite il modello fondamentale.

8.3.1 Memoria nello stato

Il controllo può servire non solo a rappresentare una posizione nel "programma" di una macchina di Turing, ma anche a conservare una quantità finita di dati. Questa tecnica è illustrata, insieme con il concetto di tracce multiple, nella Figura 8.13. Il controllo non consiste unicamente nello "stato" vero e proprio, q, ma anche in tre dati: A, B e C. Non sono richieste espansioni al modello; è sufficiente considerare lo stato come una ennupla. Nel caso della Figura 8.13 lo stato è $[q, A, B, C]$. Possiamo così descrivere le transizioni in modo più sistematico, esplicitando chiaramente la strategia del programma.

Esempio 8.6 Definiamo una TM

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], \{[q_1, B]\})$$

che ricorda nel controllo il primo simbolo letto (0 o 1) e verifica che non ricompaia nell'input. M accetta dunque il linguaggio $\mathbf{01}^* + \mathbf{10}^*$. Il riconoscimento di un linguaggio

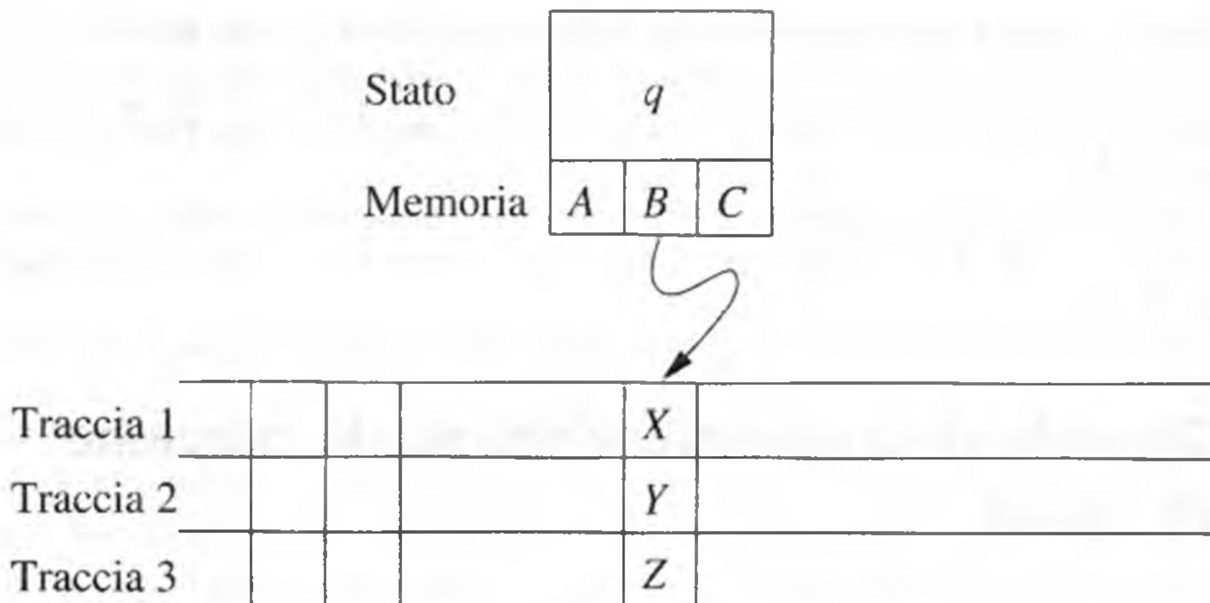


Figura 8.13 Una macchina di Turing con memoria nel controllo e tracce multiple.

regolare come questo non sfrutta a fondo le capacità delle macchine di Turing, ma serve da esempio.

L'insieme degli stati Q è $\{q_0, q_1\} \times \{0, 1, B\}$. Ogni stato è composto di due parti.

a) Una parte di controllo, q_0 o q_1, che indica che cosa sta facendo la TM. Lo stato di controllo q_0 indica che M non ha ancora letto il primo simbolo; q_1 indica che l'ha letto e sta verificando, spostandosi verso destra fino a una cella vuota, che quel simbolo non ricompaia altrove.

b) Un dato che ricorda il primo simbolo letto, 0 o 1. Un simbolo B in questa componente significa che non è stato letto nessun simbolo.

Definiamo la funzione di transizione δ per M.

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ per $a = 0$ o $a = 1$. All'inizio q_0 è lo stato di controllo e B è il valore della porzione di dato. Il simbolo letto viene copiato nella seconda componente dello stato, ed M si muove a destra entrando allo stesso tempo nello stato q_1.

2. $\delta([q_1, a], \overline{a}) = ([q_1, a], \overline{a}, R)$, dove $\overline{a}$ è il "complemento" di a, cioè 0 se $a = 1$ e 1 se $a = 0$. Nello stato q_1, M supera ogni simbolo, 0 o 1, diverso da quello conservato nello stato, e prosegue verso destra.

3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ per $a = 0$ o $a = 1$. Se M raggiunge il primo blank, entra nello stato accettante $[q_1, B]$.

Notiamo che, in M, $\delta([q_1, a], a)$ non è definita per $a = 0$ o $a = 1$. Di conseguenza, se incontra per la seconda volta il simbolo conservato nel suo controllo, M si ferma senza entrare nello stato accettante. □

8.3.2 Tracce multiple

A volte è utile considerare il nastro di una macchina di Turing come se avesse più "tracce". Ogni traccia può ospitare un simbolo, e quindi l'alfabeto di nastro della TM consiste in ennuple, con una componente per ogni traccia. A titolo di esempio, la cella guardata dalla testina nella Figura 8.13 contiene il simbolo $[X, Y, Z]$. Anche l'impiego di tracce multiple, come la memoria nel controllo, non estende le capacità delle macchine di Turing. Si tratta soltanto di un modo per strutturare i simboli di nastro.

Esempio 8.7 Un impiego comune delle tracce multiple è di destinarne una ai dati e una seconda a opportuni segnali. Possiamo "spuntare" i simboli a mano a mano che li "usiamo" o marcare certe posizioni con un segnale. Negli Esempi 8.2 e 8.4 ci siamo valsi di questa tecnica, pur senza considerare esplicitamente il nastro come diviso in tracce. In questo esempio ci serviamo di una seconda traccia esplicita per riconoscere un linguaggio non libero dal contesto:

$$L_{wcw} = \{wcw \mid w \text{ è in } (\mathbf{0} + \mathbf{1})^+\}$$

La macchina di Turing destinata allo scopo è

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$$

Q: l'insieme degli stati è $\{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$, cioè le coppie formate da uno stato di controllo q_i e da una componente di dati, 0, 1 o B. Ci serviamo nuovamente della memoria nel controllo e facciamo in modo che lo stato ricordi un simbolo di input, 0 o 1.

Γ: l'insieme dei simboli di nastro è $\{B, *\} \times \{0, 1, c, B\}$. La prima componente, o traccia, può essere bianca (simbolo B) o "vistata" (simbolo $*$). Ci serviamo di $*$ per contrassegnare i simboli del primo e del secondo gruppo di 0 e 1, per poter infine confermare che la stringa a sinistra della c, segnale di mezzo, è uguale a quella a destra. La seconda componente di un simbolo di nastro reca il simbolo di nastro vero e proprio. Così il simbolo $[B, X]$ corrisponde al simbolo di nastro X, per $X = 0, 1, c, B$.

Σ: i simboli di input sono $[B, 0]$ e $[B, 1]$, che identifichiamo rispettivamente, come spiegato, con 0 e 1.

δ: la funzione di transizione δ è definita dalle regole seguenti, in cui a e b possono valere 0 o 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. Nello stato iniziale M legge il simbolo a (che vale 0 o 1), lo depone nel controllo, va nello stato di controllo q_2, "vista" il simbolo appena letto e si sposta a destra. Notiamo che il "visto" si ottiene cambiando da B a $*$ il primo componente del simbolo di nastro.

2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. M si sposta verso destra in cerca del simbolo c. Ricordiamo che sia a sia b possono valere, indipendentemente, 0 o 1, ma non c.

3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. Quando legge c, M prosegue verso destra, ma cambia in q_3 lo stato di controllo.

4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. Nello stato q_3, M prosegue, superando tutti i simboli vistati.

5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. Se il primo simbolo non vistato che M trova coincide con quello nel controllo, lo vista, perché si abbina con il simbolo corrispondente nel primo blocco di 0 e 1. M va nello stato di controllo q_4, elimina il simbolo dal controllo e comincia a muoversi verso sinistra.

6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. M oltrepassa verso sinistra i simboli vistati.

7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. Quando incontra il simbolo c, M va nello stato q_5 e prosegue verso sinistra. Nello stato q_5, M deve prendere una decisione, a seconda che il simbolo immediatamente a sinistra di c sia o no vistato. Se è vistato, M ha già esaminato l'intero primo blocco di 0 e 1, quello a sinistra di c. Deve allora accertarsi che anche tutti gli 0 e gli 1 a destra di c siano vistati e, in tal caso, accettare. Se il simbolo immediatamente a sinistra di c non è vistato, M trova il simbolo non vistato più a sinistra, lo memorizza e ricomincia il ciclo avviato in q_1.

8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$. Questa diramazione copre il caso in cui il simbolo a sinistra di c non è vistato. M va nello stato q_6 e prosegue a sinistra in cerca di un simbolo vistato.

9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$. Finché i simboli non sono vistati, M resta nello stato q_6 e si sposta a sinistra.

10. $\delta([q_6, B], [*, a]) = ([q_1, B], [*, a], R)$. Quando trova il simbolo vistato, M entra nello stato q_1 e si sposta a destra per memorizzare il prossimo simbolo non vistato.

11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$. Ora M tratta il caso in cui dallo stato q_5 si è appena spostata a sinistra di c e trova un simbolo vistato. M ricomincia a muoversi verso destra entrando nello stato q_7.
12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$. Nello stato q_7, M vede certamente c. Allo stesso tempo entra nello stato q_8 e prosegue verso destra.
13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$. M si sposta nello stato q_8 oltrepassando qualsiasi 0 o 1 vistato.
14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$. Se raggiunge una cella vuota nello stato q_8 senza incontrare alcuno 0 o 1 non vistato, M accetta. Se invece incontra uno 0 o un 1 non vistato, i blocchi prima e dopo c non coincidono: M si arresta senza accettare.

□

8.3.3 Subroutine

Come per i programmi in genere, conviene pensare che una macchina di Turing sia un insieme di parti, o "subroutine", che interagiscono. Una subroutine di una macchina di Turing è un insieme di stati che eseguono una procedura. In questo insieme distinguiamo uno stato iniziale e uno stato temporaneamente senza mosse, che serve da stato di "ritorno" per passare il controllo a un altro insieme, quello che ha chiamato la subroutine. Parliamo di "chiamata" a una subroutine quando c'è una transizione al suo stato iniziale. Poiché la TM non ha modo di ricordare un "indirizzo di ritorno", cioè lo stato in cui andare al termine di una subroutine, se vogliamo chiamare una subroutine da stati diversi dobbiamo farne più copie, con un nuovo insieme di stati per ciascuna. Le "chiamate" portano agli stati iniziali di copie diverse della subroutine; ogni copia "ritorna" a uno stato diverso.

Esempio 8.8 Definiamo una TM che realizzi la funzione "moltiplicazione". La TM comincerà con $0^m 1 0^n 1$ sul nastro e terminerà con 0^{mn}. La strategia risolutiva si articola in quattro punti.

1. Il nastro ospita, in generale, una stringa non vuota della forma $0^i 1 0^n 1 0^{kn}$ per un certo k.

2. In un passo elementare cambiamo in B uno 0 del primo gruppo e aggiungiamo n volte 0 all'ultimo gruppo, producendo una stringa della forma $0^{i-1} 1 0^n 1 0^{(k+1)n}$.

3. L'esito è di copiare in coda il gruppo di n simboli 0 per m volte, una per ogni trasformazione di 0 in B nel primo gruppo. Quando il primo gruppo di 0 è stato interamente trasformato in blank, nell'ultimo gruppo ci sono mn simboli 0.

4. L'ultimo passo consiste nel cancellare la sequenza iniziale 10^n1.

Il cuore dell'algoritmo è una subroutine, che chiameremo Copy, destinata a eseguire il passo (2), cioè a copiare in coda il blocco di n simboli 0. Più esattamente Copy trasforma una ID di forma $0^{m-k}1q_10^n10^{(k-1)n}$ nella ID $0^{m-k}1q_50^n10^{kn}$. Le transizioni di Copy sono illustrate nella Figura 8.14. La subroutine contrassegna il primo 0 con X, si sposta a destra nello stato q_2 fino a un blank, vi copia lo 0 e si sposta a sinistra nello stato q_3 fino alla X. Il ciclo si ripete finché, nello stato q_1, trova un 1 anziché uno 0. A quel punto, nello stato q_4, ritrasforma gli X in 0 e termina nello stato q_5.

La macchina di Turing completa per la moltiplicazione parte dallo stato q_0. Per prima cosa passa, in più mosse, dalla ID $q_00^m10^n$ alla ID $0^{m-1}1q_10^n1$. Le transizioni necessarie sono illustrate nella Figura 8.15, a sinistra della chiamata di subroutine; queste transizioni coinvolgono solo gli stati q_0 e q_6.

Nella Figura 8.15, a destra della chiamata alla subroutine, si vedono gli stati da q_7 a q_{12}. Gli stati q_7, q_8 e q_9 hanno lo scopo di riprendere il controllo dopo che Copy ha copiato un blocco di n simboli 0 e si trova nella ID $0^{m-k}1q_50^n10^{kn}$. Da questi stati la TM giunge infine allo stato $q_00^{m-k}10^n10^{kn}$. A quel punto il ciclo ricomincia, e Copy viene chiamata di nuovo per copiare il blocco di n 0.

Nello stato q_8, però, la TM può rilevare che tutti gli m 0 sono stati mutati in blank (cioè $k = m$). In tal caso avviene una transizione allo stato q_{10}. Insieme a q_{11} questo stato trasforma in blank la sequenza iniziale 10^n1, per poi passare allo stato di arresto q_{12}. Ora la TM si trova nella ID $q_{12}0^{mn}$ e il suo compito è terminato. □

8.3.4 Esercizi

! **Esercizio 8.3.1** Ridefinite le macchine di Turing dell'Esercizio 8.2.2 sfruttando le tecniche di programmazione descritte nel Paragrafo 8.3.

! **Esercizio 8.3.2** Un'operazione frequente nei programmi per macchine di Turing comporta uno "slittamento". In teoria vorremmo creare una cella in più accanto alla posizione corrente della testina per collocarvi un simbolo, ma non possiamo alterare il nastro in questo modo. Dobbiamo allora spostare di una cella a destra il contenuto di ogni cella a destra della testina, per poi tornare alla posizione corrente. Spiegate come realizzare questa operazione. *Suggerimento*: lasciate un simbolo speciale per contrassegnare il punto in cui riportare la testina.

* **Esercizio 8.3.3** Scrivete una subroutine che sposti la testina di una TM dalla posizione corrente verso destra, saltando gli 0 fino a raggiungere un 1 o un blank. Se la posizione corrente non contiene uno 0, la subroutine deve arrestarsi. Potete assumere che i simboli di input siano solo 0, 1 e B (blank). Servitevi della subroutine per definire una TM che accetta le stringhe di 0 e 1 che non hanno due 1 consecutivi.

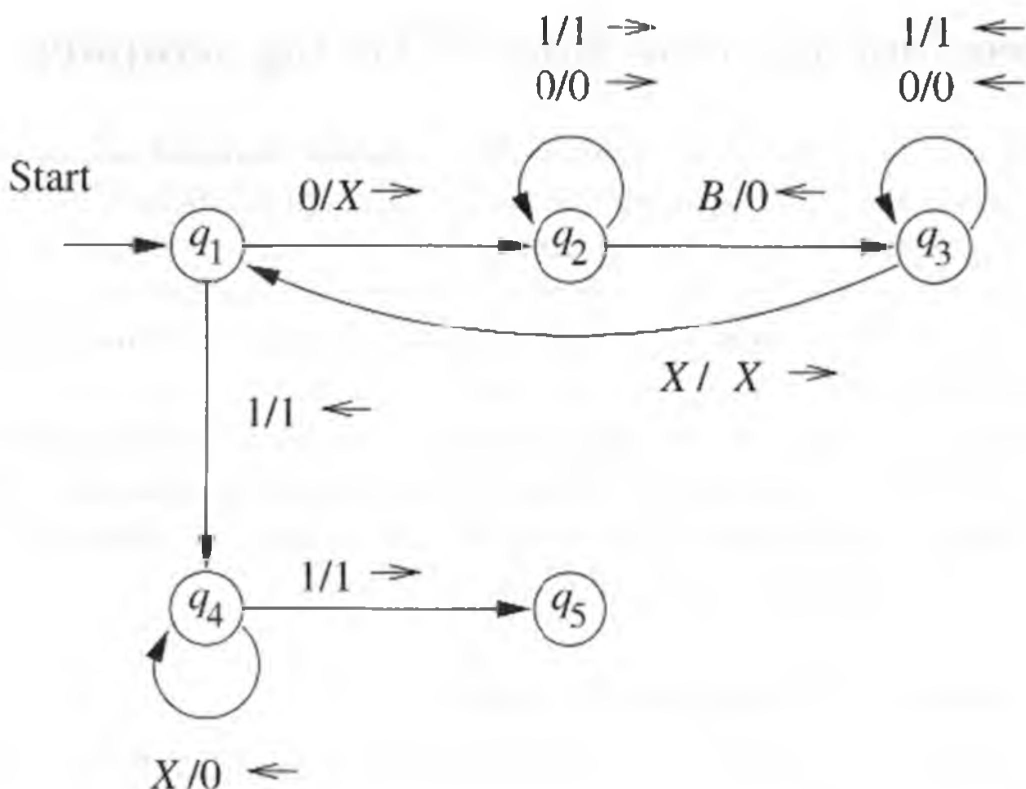


Figura 8.14 La subroutine Copy.

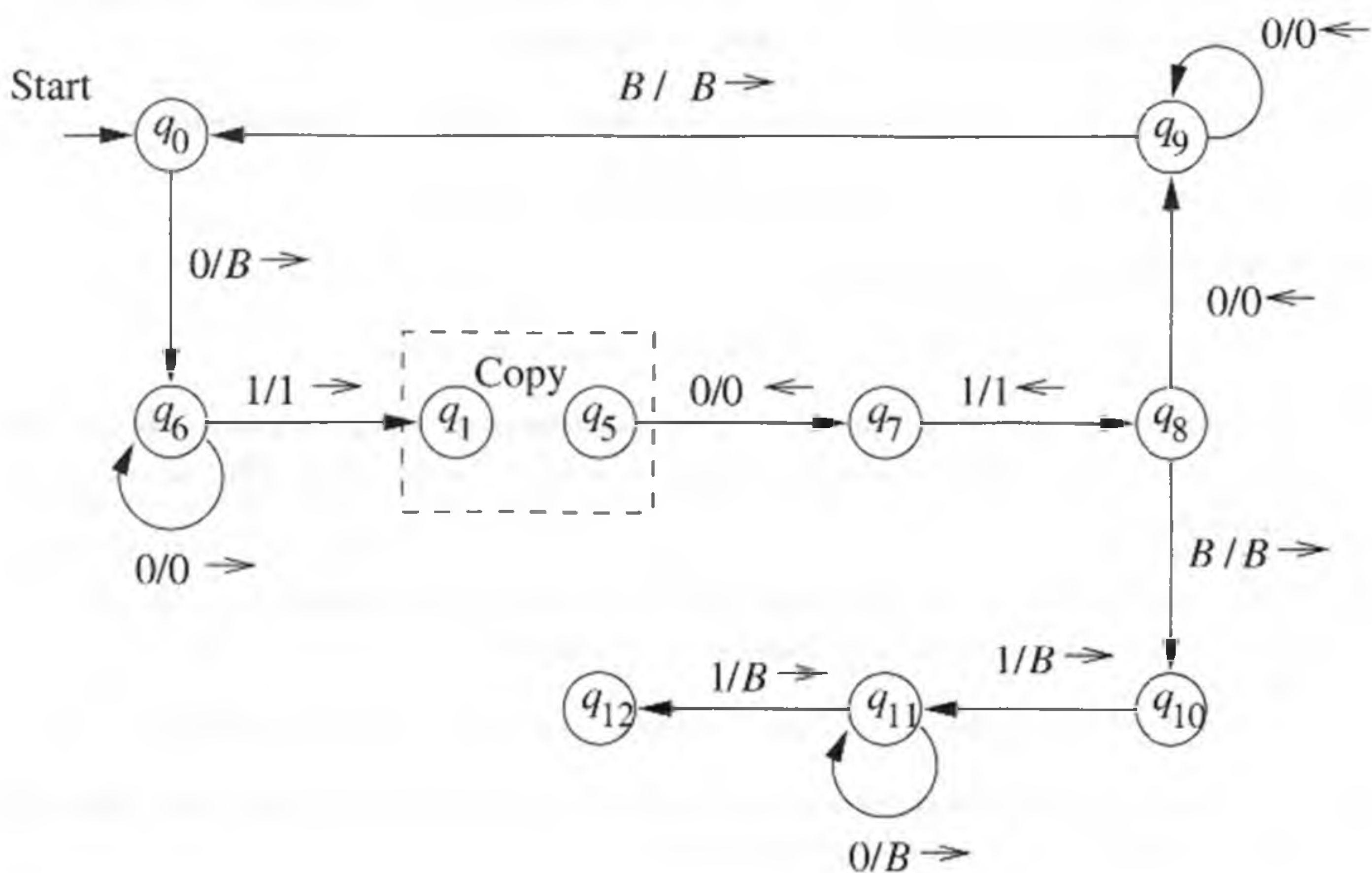


Figura 8.15 Il programma per la moltiplicazione impiega la subroutine Copy.

8.4 Estensioni alla macchina di Turing semplice

In questo paragrafo vedremo alcuni modelli di calcolatore analoghi alle macchine di Turing e con le stesse capacità di riconoscimento dei linguaggi del modello di base di TM fin qui trattato. Uno di essi, la macchina di Turing multinastro, è importante perché rispetto al modello mononastro facilita la simulazione di un vero calcolatore (o di macchine di Turing d'altro tipo). Per quanto riguarda la capacità di riconoscere linguaggi, i nastri in più non aggiungono nulla.

Esaminiamo poi la macchina di Turing non deterministica, un'estensione del modello di base che può scegliere fra un insieme finito di mosse in ogni situazione. Anche questa estensione facilita in certi casi la "programmazione" di una macchina di Turing senza allargare i linguaggi definibili rispetto al modello di base.

8.4.1 Macchine di Turing multinastro

Una TM multinastro si presenta come suggerito dalla Figura 8.16: ha un controllo finito e un numero finito di nastri. Ogni nastro è diviso in celle, e ogni cella può contenere un simbolo dell'alfabeto, finito, di nastro. Come per la TM mononastro, l'insieme dei simboli di nastro comprende un blank e ha un sottoinsieme, i simboli detti di input, che non contiene il blank. L'insieme degli stati comprende uno stato iniziale e un insieme di stati accettanti. Inizialmente valgono queste condizioni.

1. L'input, una sequenza finita di simboli di input, si trova sul primo nastro.

2. Tutte le altre celle di ogni nastro contengono un blank.

3. Il controllo si trova nello stato iniziale.

4. La testina del primo nastro è all'estremo sinistro dell'input.

5. Le altre testine si trovano su celle arbitrarie. Poiché i nastri, eccetto il primo, sono bianchi, la posizione iniziale delle loro testine è irrilevante: tutte le celle sono equivalenti.

Una mossa di una TM multinastro dipende dallo stato e dai simboli letti da ciascuna testina. In una mossa la macchina compie tre operazioni.

1. Il controllo entra in un nuovo stato, che può coincidere con quello corrente.

2. Su ogni cella guardata da una testina viene scritto un nuovo simbolo di nastro, che può essere quello già contenuto nella cella.

3. Ogni testina si muove a sinistra o a destra, oppure sta ferma. I movimenti sono indipendenti: testine diverse si possono muovere in direzioni diverse o star ferme.

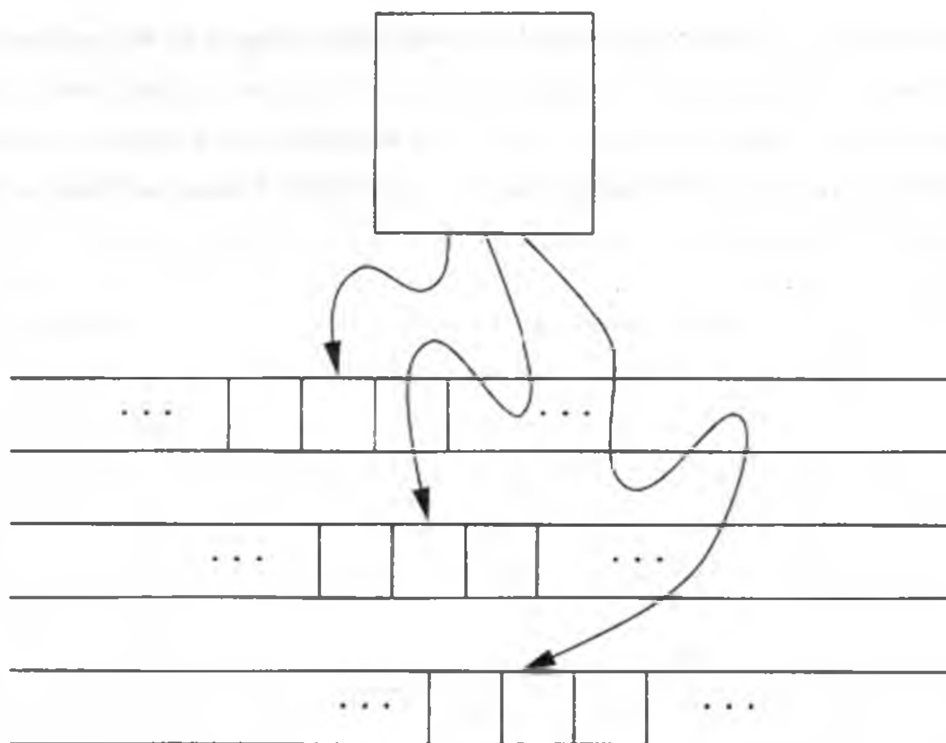


Figura 8.16 Una macchina di Turing multinastro.

Non daremo la definizione formale delle regole di transizione, che sono una generalizzazione immediata di quelle per le TM mononastro, tranne che per la direzione, che qui è indicata dai simboli L (sinistra), R (destra) ed S (ferma). Nella macchina mononastro che abbiamo definito la testina non può stare ferma: manca quindi il simbolo S. Lasciamo al lettore il compito di escogitare una notazione formale appropriata per le descrizioni istantanee della configurazione di una TM multinastro. Le macchine di Turing multinastro accettano entrando in uno stato accettante come quelle mononastro.

8.4.2 Equivalenza di macchine di Turing mononastro e multinastro

Ricordiamo che i linguaggi ricorsivamente enumerabili sono, per definizione, quelli accettati da una TM mononastro. Ovviamente le TM multinastro accettano tutti i linguaggi ricorsivamente enumerabili, perché una TM mononastro è un caso particolare di TM multinastro. Possiamo chiederci allora se esistono linguaggi non ricorsivamente enumerabili, ma accettati da una TM multinastro. La risposta è "no", e lo proveremo spiegando come simulare una TM multinastro per mezzo di una TM mononastro.

Teorema 8.9 Ogni linguaggio accettato da una TM multinastro è ricorsivamente enumerabile.

DIMOSTRAZIONE La dimostrazione è illustrata nella Figura 8.17. Sia L un linguaggio accettato da una TM con k nastri, M. Simuliamo M con una TM mononastro N, il cui

nastro è diviso in $2k$ tracce. Metà delle tracce replicano i nastri di M; ognuna delle restanti ospita un marcatore che indica la posizione corrente della testina del corrispondente nastro di M. Nella Figura 8.17 ipotizziamo $k = 2$. La seconda e la quarta traccia replicano il contenuto del primo e del secondo nastro di M; la traccia 1 reca la posizione della testina del nastro 1; la traccia 3 la posizione della testina del secondo nastro.

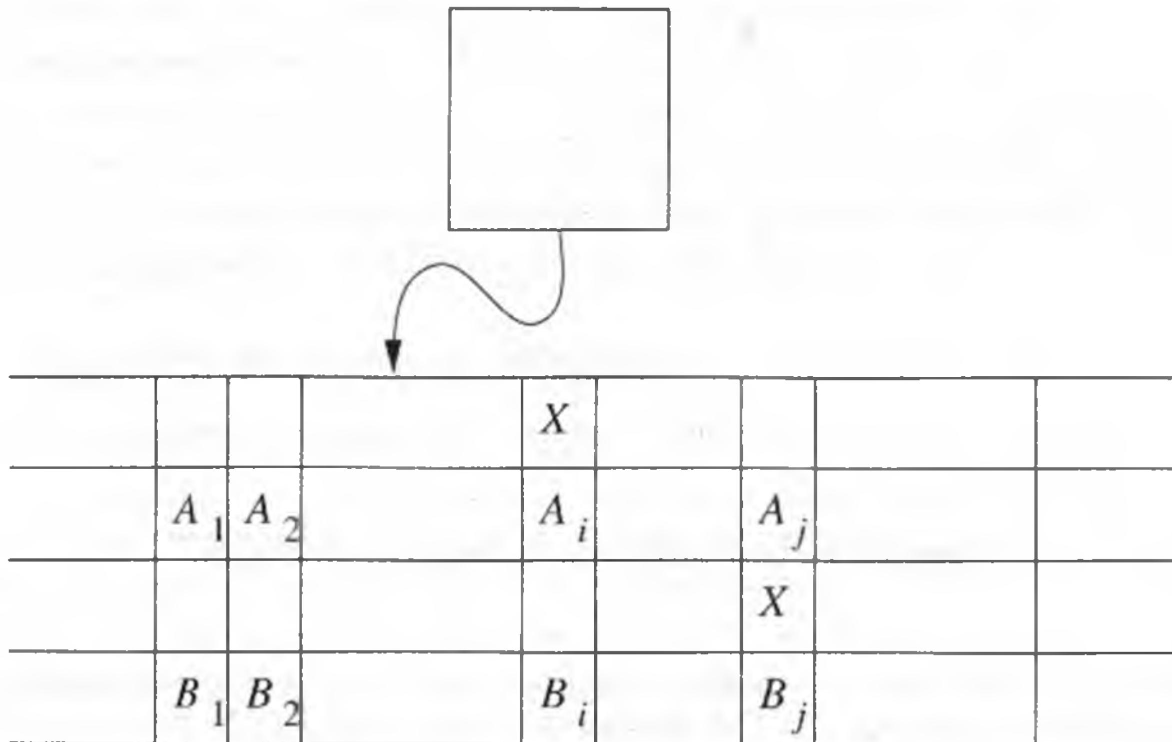


Figura 8.17 Simulazione di una macchina di Turing con due nastri per mezzo di una macchina di Turing mononastro.

Per simulare una mossa di M, la testina di N deve visitare i k marcatori. Per non smarrirsi, N deve tener conto di quanti marcatori stanno alla sua sinistra in ogni istante; il contatore è conservato come componente del controllo di N. Dopo aver visitato ogni marcatore e memorizzato, in una componente del controllo, il simbolo nella cella corrispondente, N sa quali simboli di nastro sono guardati dalle testine di M. Inoltre N conosce lo stato di M, memorizzato nel suo controllo. Perciò N può stabilire la mossa successiva di M.

Ora N rivisita ogni marcatore, modifica il simbolo sulla traccia che rappresenta il nastro corrispondente di M e sposta, se richiesto, ciascun marcatore a destra o a sinistra. Infine cambia lo stato di M registrato nel suo controllo. A questo punto N ha simulato una mossa di M.

Gli stati accettanti di N sono quelli in cui è registrato uno stato accettante di M. In questo modo N accetta quando, e solo quando, M accetta. □

Richiamo sulla finitezza

Confondere un valore che a ogni istante è finito con un insieme finito di valori è un errore frequente. La costruzione da n a 1 nastro serve a illustrare la differenza. In essa sfruttiamo le tracce del nastro per registrare le posizioni delle testine. Perché non possiamo registrare le posizioni come interi nel controllo? Sbagliando, si potrebbe ragionare così: dopo n mosse, le testine della TM devono trovarsi a distanze inferiori a n dalle posizioni iniziali; dunque per ogni testina si deve memorizzare un intero non più grande di n.

In realtà, anche se in ogni istante le posizioni sono espresse da numeri finiti, l'insieme delle posizioni possibili in tutti gli istanti è infinito. Per poter rappresentare qualsiasi posizione, un componente dello stato deve poter registrare qualsiasi intero. Questo componente imporrebbe all'insieme degli stati di essere infinito, sebbene in ogni istante solo un numero finito di essi sia possibile. La definizione di macchina di Turing richiede che l'*insieme* degli stati sia finito. Non è quindi possibile memorizzare la posizione di una testina nel controllo finito.

8.4.3 Tempo di esecuzione e costruzione da n a un nastro

Introduciamo un concetto che diventerà importante in seguito: la "complessità in tempo", o "tempo di esecuzione", di una macchina di Turing. Diciamo che il *tempo di esecuzione* della TM M sull'input w è il numero di passi che M compie prima di arrestarsi. Se M non si arresta su w, diciamo che il tempo di esecuzione è infinito. La *complessità in tempo* della TM M è la funzione $T(n)$, definita come il massimo, su tutti gli input w di lunghezza n, del tempo di esecuzione di M su w. Per una macchina di Turing che non si arresta su tutti gli input, $T(n)$ può essere infinito per alcuni n, o anche per tutti. Dedicheremo particolare attenzione alle TM che si arrestano su tutti gli input, e in particolare a quelle che hanno complessità in tempo polinomiale, a partire dal Paragrafo 10.1.

La costruzione del Teorema 8.9 può sembrare contorta. In effetti la TM mononastro può avere tempi di esecuzione molto più lunghi di quella multinastro. D'altra parte i tempi di esecuzione delle due macchine sono commensurabili in senso debole: quello della TM mononastro non supera il quadrato del tempo dell'altra. L'elevamento al quadrato, seppure non trascurabile, preserva la polinomialità del tempo di esecuzione. Nel Capitolo 10 prenderemo in considerazione due fatti.

a) La differenza fra tempo polinomiale e tassi di crescita più alti è lo spartiacque fra problemi risolvibili e problemi irrisolvibili nella pratica con un calcolatore.

b) Nonostante studi approfonditi, per molti problemi il tempo di esecuzione si può

stimare solo a meno di un polinomio. Perciò nel valutare il tempo necessario per risolvere un problema non è fondamentale distinguere fra una TM mononastro o una multinastro.

Dimostriamo ora che i tempi di esecuzione di una TM multinastro e della corrispondente mononastro sono in rapporto quadratico.

Teorema 8.10 Il tempo necessario alla TM mononastro N del Teorema 8.9 per simulare n mosse della TM con k nastri M è $O(n^2)$.

DIMOSTRAZIONE Dopo n mosse di M i marcatori delle testine non possono essere distanti fra loro più di $2n$ celle. Perciò, partendo dal marcatore più a sinistra, per trovarli tutti N non deve spostarsi di più di $2n$ celle a destra. A quel punto può tornare verso sinistra modificando il contenuto dei nastri simulati di M e spostando i marcatori a sinistra o a destra. Questa operazione non richiede più di $2n$ movimenti a sinistra, oltre a non più di $2k$ mosse nell'altra direzione per scrivere un marcatore X nella cella a destra (se una testina di M si sposta a destra).

Quindi il numero di mosse che N deve fare per simulare una delle prime n mosse non supera $4n + 2k$. Poiché k è una costante, indipendente dal numero di mosse simulate, questo numero è $O(n)$. La simulazione di n mosse richiede non più di n volte quella quantità, cioè $O(n^2)$. □

8.4.4 Macchine di Turing non deterministiche

Una macchina di Turing *non deterministica* (NTM, *Nondeterministic Turing Machine*) si distingue da quella deterministica nella funzione di transizione δ, che associa a ogni stato q e a ogni simbolo di nastro X un insieme di triple:

$$\delta(q, X) = \{(q_1, Y_1, D_1),\ (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

dove k è un intero finito. A ogni passo una NTM sceglie una delle triple come mossa. Non può però scegliere lo stato da una tripla, il simbolo di nastro da un'altra e la direzione da una terza.

Il linguaggio accettato da una NTM M è definito in modo analogo ad altri modelli non deterministici già trattati, come gli NFA e i PDA: M accetta un input w se c'è una sequenza di scelte che conduce dalla ID iniziale con w come input a una ID con stato accettante. Come per gli NFA e i PDA, l'esistenza di scelte alternative che non portano a uno stato accettante è irrilevante.

Le NTM non accettano linguaggi che non siano accettati anche da TM deterministiche (o DTM, se intendiamo sottolinearne il carattere deterministico). La dimostrazione consiste nel costruire, per ogni NTM M_N, una DTM M_D che esamina le ID raggiungibili da M_N per tutte le scelte possibili. Se M_D ne trova una con stato accettante, entra in uno

dei propri stati accettanti. M_D deve procedere sistematicamente, collocando le nuove ID in una coda anziché in uno stack, in modo da simulare in un tempo finito tutte le sequenze di k mosse di M_N, per $k = 1, 2, \ldots$.

Teorema 8.11 Se M_N è una macchina di Turing non deterministica, esiste una macchina di Turing deterministica M_D tale che $L(M_N) = L(M_D)$.

DIMOSTRAZIONE Costruiamo M_D come TM multinastro, secondo lo schema della Figura 8.18. Il primo nastro di M_D contiene una sequenza di ID di M_N che comprendono lo stato di M_N. Una ID di M_N è contrassegnata come "corrente", nel senso che le sue ID successive sono in corso di elaborazione. Nella Figura 8.18 la terza ID è marcata da un x e dal separatore di ID, $*$. Tutte le ID a sinistra della ID corrente sono già state elaborate e si possono ignorare.

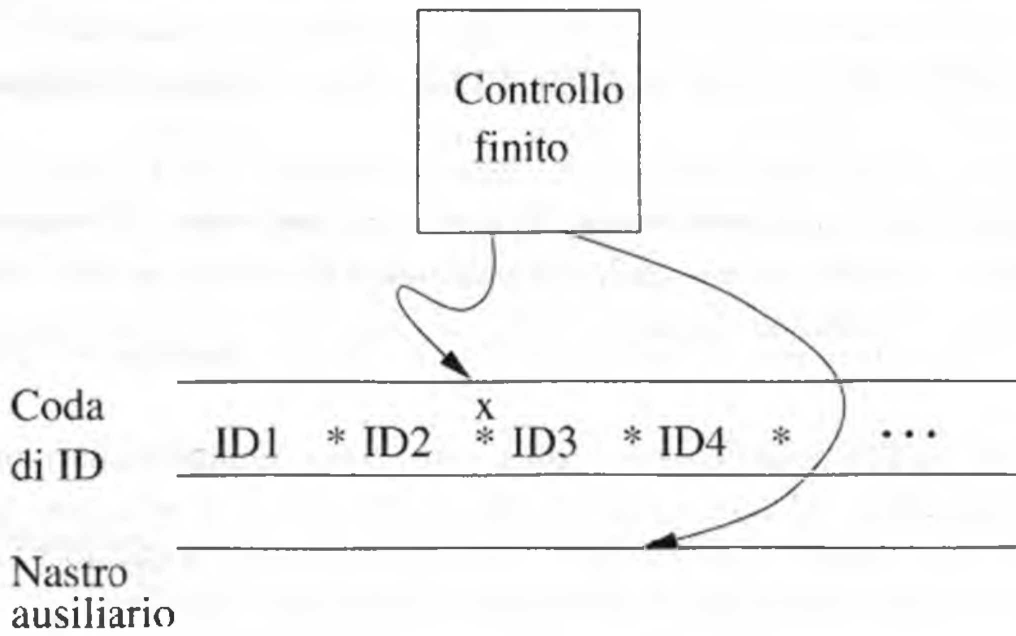


Figura 8.18 Simulazione di una NTM da parte di una DTM.

Per elaborare la ID corrente, M_D compie quattro operazioni.

1. Esamina lo stato e il simbolo guardato della ID corrente. Nel controllo di M_D sono incorporate le scelte di mossa di M_N per ogni stato e simbolo. Se lo stato nella ID corrente è accettante, M_D accetta e smette di simulare M_N.

2. Se lo stato non è accettante, e la combinazione stato-simbolo permette k mosse, M_D copia la ID sul secondo nastro e fa k copie della ID in coda alla sequenza di ID sul nastro 1.

3. M_D modifica ognuna delle k ID secondo una delle k scelte di mossa che M_N può fare dalla ID corrente.

4. M_D torna alla ID corrente contrassegnata in precedenza, cancella il contrassegno e lo sposta alla successiva ID a destra. Il ciclo ricomincia dal passo (1).

La fedeltà della simulazione dovrebbe essere evidente: M_D accetta solo se scopre che M_N può raggiungere una ID accettante. Dobbiamo però essere certi che se M_N entra in una ID accettante dopo una sequenza di n mosse, quella ID diventa prima o poi la ID corrente di M_D, che quindi accetta.

Sia m il massimo numero di scelte di M_N nelle sue configurazioni. Allora in M_N ci sono una ID iniziale, non più di m ID raggiungibili in una mossa, non più di m^2 ID raggiungibili in due mosse, e così via. Dopo n mosse, M_N può aver raggiunto al massimo $1 + m + m^2 + \cdots + m^n$ ID. Questo numero non supera nm^n.

M_D esplora le ID di M_N nell'ordine detto "in ampiezza": prima tutte le ID raggiungibili in 0 mosse (la ID iniziale), poi quelle raggiungibili in una mossa, poi quelle in due mosse, e così via. In particolare M_D fa diventare corrente, esaminandone le successive, ogni ID raggiungibile in non più di n mosse prima di elaborarne una raggiungibile in più di n mosse.

Di conseguenza la ID accettante di M_N viene considerata da M_D fra le prime nm^n. A noi importa solo che M_D elabori questa ID dopo un tempo finito; il limite che abbiamo stabilito garantisce che prima o poi quella ID sarà elaborata. Perciò se M_N accetta, accetta anche M_D. Poiché abbiamo già notato che M_D accetta solo se M_N accetta, concludiamo che $L(M_N) = L(M_D)$. □

Si osservi che la TM deterministica così costruita può impiegare un tempo esponenzialmente più alto della TM non deterministica. Non sappiamo se questo rallentamento esponenziale sia inevitabile. Il Capitolo 10 è dedicato proprio a questo problema e alle conseguenze dell'eventuale scoperta di una simulazione più efficiente.

8.4.5 Esercizi

Esercizio 8.4.1 Per ogni linguaggio dell'Esercizio 8.2.2 descrivete, informalmente ma chiaramente, una macchina di Turing multinastro che lo accetti. Cercate di avere tempi di esecuzione proporzionali alla lunghezza dell'input.

Esercizio 8.4.2 Definiamo la funzione di transizione della TM non deterministica $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_2\})$:

δ	0	1	B
q_0	$\{(q_0, 1, R)\}$	$\{(q_1, 0, R)\}$	$\emptyset$
q_1	$\{(q_1, 0, R),\ (q_0, 0, L)\}$	$\{(q_1, 1, R),\ (q_0, 1, L)\}$	$\{(q_2, B, R)\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Elencate le ID raggiungibili dalla ID iniziale per i seguenti input:

* a) 01

b) 011.

Esercizio 8.4.3 Per ognuno dei seguenti linguaggi definite, informalmente ma chiaramente, una macchina di Turing non deterministica, eventualmente multinastro, che lo riconosca. Cercate di utilizzare il non determinismo per evitare le iterazioni e risparmiare tempo (nel senso non deterministico). In altre parole ideate una NTM con tante diramazioni brevi.

* a) Il linguaggio di tutte le stringhe di 0 e 1 contenenti una stringa di lunghezza 100, ripetuta, anche non consecutivamente. Formalmente questo linguaggio è l'insieme delle stringhe di 0 e 1 della forma $wxyxz$, con $|x| = 100$ e w, y e z di lunghezza arbitraria.

b) Il linguaggio di tutte le stringhe della forma $w_1\#w_2\#\cdots\#w_n$, per ogni n, tali che ogni w_i è una stringa di 0 e 1, e per qualche j, w_j è l'intero j in binario.

c) Il linguaggio di tutte le stringhe della stessa forma di (b), ma tali che, per almeno due valori di j, w_j è uguale a j in binario.

! **Esercizio 8.4.4** Considerate la macchina di Turing non deterministica

$$M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_f\})$$

Descrivete, informalmente ma chiaramente, il linguaggio $L(M)$ se δ è definita dalle seguenti regole: $\delta(q_0, 0) = \{(q_0, 1, R), (q_1, 1, R)\}$; $\delta(q_1, 1) = \{(q_2, 0, L)\}$; $\delta(q_2, 1) = \{(q_0, 1, R)\}$; $\delta(q_1, B) = \{(q_f, B, R)\}$.

* **Esercizio 8.4.5** Considerate una TM non deterministica con nastro infinito in ambedue le direzioni. In un dato istante il nastro è completamente bianco, tranne per una cella, che contiene il simbolo \$. La testina guarda una cella bianca e lo stato è q.

a) Scrivete le transizioni che fanno entrare la NTM nello stato p mentre guarda il \$.

! b) Supponete ora che la TM sia deterministica. Come fare per farle trovare il \$ ed entrare in p?

Esercizio 8.4.6 Definite la macchina di Turing a due nastri descritta sotto, che accetta il linguaggio delle stringhe di 0 e 1 con lo stesso numero dei due simboli. Il primo nastro contiene l'input ed è percorso da sinistra a destra. Il secondo contiene gli 0 o gli 1 in eccesso nella porzione di input già letta. Specificate gli stati e le transizioni, e lo scopo di ogni stato.

Esercizio 8.4.7 In questi esercizi realizziamo uno stack mediante una speciale TM a tre nastri.

1. Il primo nastro serve solo a conservare e leggere l'input. L'alfabeto di input è formato dal simbolo ↑, che interpretiamo come "togli dallo stack", e dai simboli a e b, che interpretiamo come "metti a (o b) nello stack".

2. Il secondo nastro rappresenta lo stack.

3. Il terzo nastro è il nastro di output. Ogni simbolo tolto dallo stack dev'essere scritto sul nastro di output, in coda a quelli scritti in precedenza.

La macchina deve partire con lo stack vuoto e compiere la sequenza di operazioni "togli" e "metti" specificata in input, leggendo da sinistra a destra. Se l'input chiede di togliere da uno stack vuoto, la TM deve arrestarsi in uno stato di errore q_e. Se l'intero input lascia lo stack vuoto, la TM lo accetta entrando nello stato finale q_f. Descrivete, informalmente ma chiaramente, la funzione di transizione della TM. Dichiarate lo scopo di ogni stato.

Esercizio 8.4.8 Nella Figura 8.17 abbiamo visto un esempio della simulazione generica di una TM a k nastri da parte di una TM mononastro.

* a) Supponete di usare quella tecnica per simulare una TM a cinque nastri con alfabeto di nastro di sette simboli. Quanti simboli di nastro ha la TM mononastro?

* b) Un modo alternativo di simulare k nastri con uno solo prevede l'uso di una $(k+1)$-esima traccia per memorizzare le posizioni delle k testine, mentre le prime k tracce simulano i k nastri nel modo ovvio. Si noti che nella $(k+1)$-esima traccia si deve fare attenzione a distinguere le testine e a trattare il caso in cui due testine sono nella stessa posizione. Questo metodo riduce il numero di simboli di nastro necessari per la TM mononastro?

c) Un altro modo di simulare k nastri con uno solo evita di memorizzare le posizioni delle testine. La $(k+1)$-esima traccia si usa per marcare una cella del nastro. A ogni istante ciascun nastro simulato è collocato sulla traccia corrispondente in modo che la testina si trovi sulla cella marcata. Se la TM a k nastri sposta la testina del nastro i, la TM simulante fa scorrere l'intera porzione non bianca dell'i-esima traccia di una cella nella direzione opposta, così che la cella marcata corrisponda sempre a quella guardata dall'i-esima testina della TM a k nastri. Questo metodo riduce il numero di simboli di nastro della TM mononastro? Ha qualche difetto rispetto agli altri metodi discussi?

! **Esercizio 8.4.9** Una macchina di Turing a k testine è dotata di k testine sullo stesso nastro. Ogni sua mossa dipende dallo stato e dal simbolo che ogni testina guarda. In una mossa la

TM cambia stato, scrive un nuovo simbolo su ogni cella guardata da una testina e sposta ogni testina a sinistra o a destra, o la lascia ferma. Poiché più testine possono guardare la stessa cella, assumiamo che le testine siano numerate da 1 a k e che in una cella rimanga scritto il simbolo della testina con il numero più alto. Dimostrate che i linguaggi accettati dalle macchine di Turing a k testine sono quelli accettati dalle TM ordinarie.

!! **Esercizio 8.4.10** Una macchina di Turing *bidimensionale* ha il solito controllo a stati finiti, ma il suo nastro è una griglia bidimensionale di celle, infinita in tutte le direzioni. L'input è collocato su una riga, con la testina al suo estremo sinistro e il controllo nello stato iniziale. La macchina accetta entrando in uno stato finale. Dimostrate che i linguaggi accettati dalle macchine di Turing bidimensionali sono quelli accettati dalle TM ordinarie.

8.5 Macchine di Turing ridotte

Abbiamo esaminato alcune generalizzazioni della macchina di Turing che non ne allargano la capacità di riconoscimento di linguaggi. Consideriamo ora esempi di TM apparentemente limitate, ma che hanno esattamente la stessa capacità. La prima limitazione è secondaria, ma utile in alcune costruzioni che vedremo in seguito: sostituiamo il nastro illimitato nelle due direzioni con un nastro illimitato solo a destra. Inoltre vietiamo alla TM di stampare un blank al posto del simbolo di nastro. Queste limitazioni hanno un pregio: le ID sono formate solo da simboli diversi dal blank e cominciano sempre dall'estremo sinistro dell'input.

Esploriamo poi alcuni tipi di macchina di Turing multinastro che sono automi a pila generalizzati. Vincoliamo anzitutto i nastri a comportarsi come degli stack. Li trattiamo poi solo come "contatori": possono rappresentare solo un valore intero e la TM può distinguere soltanto tra un valore 0 e un valore diverso da 0. Dalla discussione si deduce che esistono diversi tipi, molto semplici, di automa con le stesse capacità di un calcolatore. I problemi indecidibili concernenti le macchine di Turing, che studieremo nel Capitolo 9, si applicano anche a questi semplici dispositivi.

8.5.1 Macchine di Turing con nastri semi-infiniti

Finora abbiamo permesso alla testina di una macchina di Turing di muoversi a sinistra o a destra della posizione iniziale; in realtà è sufficiente permetterle di muoversi nell'area a destra della posizione iniziale. Possiamo quindi supporre che il nastro sia *semi-infinito*, cioè che non ci siano celle a sinistra della posizione iniziale. Nel prossimo teorema dimostriamo che si può sempre costruire una TM con nastro semi-infinito capace di simularne una con nastro illimitato in entrambe le direzioni, come il modello originale.

La costruzione impiega due tracce sul nastro semi-infinito. La traccia superiore rappresenta la cella su cui si trova all'inizio la testina della TM originale e quelle alla sua

destra. La traccia inferiore rappresenta le celle a sinistra della posizione iniziale, in ordine inverso. La disposizione è illustrata nella Figura 8.19. La traccia superiore rappresenta le celle $X_0, X_1, \ldots$, dove X_0 è la posizione di partenza della testina e X_1, X_2, e così via, sono le celle alla sua destra. Le celle X_{-1}, X_{-2}, e così via, rappresentano le celle a sinistra della posizione iniziale. Il simbolo $*$ nella cella più a sinistra della traccia inferiore segnala la fine della traccia e impedisce che la testina "cada" accidentalmente dal bordo sinistro del nastro semi-infinito.

X_0	X_1	X_2	...
$*$	X_{-1}	X_{-2}	...

Figura 8.19 Un nastro semi-infinito può simulare un nastro infinito nelle due direzioni.

Imponiamo un vincolo ulteriore alla macchina di Turing: il divieto di scrivere un blank. Questo semplice vincolo, insieme con il nastro semi-infinito, fa sì che in ogni istante il nastro sia formato da un prefisso senza blank seguito da una serie infinita di blank. Inoltre la sequenza di simboli diversi dal blank parte sempre dalla posizione iniziale nel nastro. Nei Teoremi 9.19 e 10.9 vedremo perché è utile poter assumere che le ID abbiano questa forma.

Teorema 8.12 Ogni linguaggio accettato da una TM M_2 è accettato anche da una TM M_1 con i seguenti vincoli.

1. La testina di M_1 non va mai a sinistra della posizione iniziale.

2. M_1 non scrive mai un blank.

DIMOSTRAZIONE Il secondo vincolo non pone problemi: si definisce un nuovo simbolo di nastro, B', che funge da blank, ma non è il blank B.

a) Se in M_2 vale la regola $\delta_2(q, X) = (p, B, D)$, la modifichiamo in $\delta_2(q, X) = (p, B', D)$.

b) Per ogni stato q definiamo $\delta_2(q, B') = \delta_2(q, B)$.

Il primo vincolo è più impegnativo. Sia

$$M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, B, F_2)$$

la TM M_2 modificata in modo che non scriva mai il blank B. Costruiamo

$$M_1 = (Q_1, \Sigma \times \{B\}, \Gamma_1, \delta_1, q_0, [B, B], F_1)$$

con le seguenti definizioni.

Q_1: gli stati di M_1 sono $\{q_0, q_1\} \cup (Q_2 \times \{U, L\})$. Essi comprendono lo stato iniziale q_0, un secondo stato q_1 e tutti gli stati di M_2 con un secondo componente, di valore U (superiore) o L (inferiore). Il secondo componente indica se M_2 guarda la traccia superiore o quella inferiore (vedi Figura 8.19). In altre parole U significa che la testina di M_2 è sulla sua posizione iniziale o alla sua destra; L significa che è a sinistra.

Γ_1: i simboli di nastro di M_1 sono coppie di simboli di Γ_2, cioè elementi di $\Gamma_2 \times \Gamma_2$. I simboli di input di M_1 sono coppie formate da un simbolo di input di M_2 nel primo componente e un blank nel secondo, cioè coppie della forma $[a, B]$, dove a è in Σ. Il blank di M_1 è formato da blank nei due componenti. Inoltre, per ogni simbolo X in Γ_2, c'è una coppia $[X, *]$ in Γ_1. Qui $*$ è un nuovo simbolo, esterno a Γ_2, che serve a marcare l'estremo sinistro del nastro di M_1.

δ_1: le transizioni di M_1 sono definite come segue.

1. $\delta_1(q_0, [a, B]) = (q_1, [a, *], R)$ per ogni a in Σ. La prima mossa di M_1 colloca il segno * sulla traccia inferiore della cella più a sinistra. Lo stato diventa q_1 e la testina si sposta a destra perché non può andare a sinistra né restare ferma.
2. $\delta_1(q_1, [X, B]) = ([q_2, U], [X, B], L)$ per ogni X in Γ_2. Nello stato q_1, M_1 riproduce le condizioni iniziali di M_2 riportando la testina alla posizione iniziale ed entrando nello stato $[q_2, U]$, lo stato iniziale di M_2, con la testina che legge la traccia superiore di M_1.
3. Se $\delta_2(q, X) = (p, Y, D)$, allora per ogni Z in Γ_2:
   (a) $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$
   (b) $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \overline{D})$

   dove $\overline{D}$ è la direzione opposta a D, cioè L se $D = R$, ed R se $D = L$. Se non si trova al bordo sinistro, M_1 simula M_2 sulla traccia opportuna: la superiore se il secondo componente dello stato è U, l'inferiore se è L. Notiamo che M_1, quando opera sulla traccia inferiore, si muove nella direzione opposta a quella di M_2. Infatti la metà sinistra del nastro di M_2 è stata ripiegata, e quindi rovesciata, lungo la traccia inferiore del nastro di M_1.
4. Se $\delta_2(q, X) = (p, Y, R)$, allora

$$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$$

Questa regola spiega uno dei modi di trattare il marcatore *. Se M_2 si muove a destra dalla posizione iniziale, a prescindere dal fatto che provenga da sinistra o da destra rispetto a quella posizione (in funzione del fatto che il secondo componente dello stato di M_1 sia L o U), M_1 deve muoversi a destra e puntare alla traccia superiore. Dunque M_1 si troverà nella posizione rappresentata da X_1 nella Figura 8.19.

5. Se $\delta_2(q, X) = (p, Y, L)$, allora

$$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$$

Questa regola somiglia alla precedente, ma tratta il caso in cui M_2 va a sinistra della posizione iniziale. M_1 deve muoversi a destra rispetto al marcatore, ma puntando alla traccia inferiore, cioè alla cella indicata da X_{-1} nella Figura 8.19.

F_1: gli stati accettanti F_1 sono gli elementi di $F_2 \times \{U, L\}$, ossia gli stati di M_1 il cui primo componente è uno stato accettante di M_2. Nel momento in cui accetta, M_1 può puntare alla traccia superiore o a quella inferiore.

La dimostrazione del teorema è praticamente completa. Possiamo osservare, per induzione sul numero di mosse di M_2, che M_1 simula sul proprio nastro la ID di M_2, a patto di rovesciare la traccia inferiore e giustapporla alla superiore. Notiamo anche che M_1 entra in uno stato accettante esattamente quando lo fa M_2. Perciò $L(M_1) = L(M_2)$. □

8.5.2 Macchine multistack

Consideriamo ora alcuni modelli computazionali fondati su generalizzazioni dell'automa a pila. Esaminiamo in primo luogo che cosa succede se dotiamo un PDA di più di una pila. Dall'Esempio 8.7 sappiamo che una macchina di Turing può accettare linguaggi che nessun PDA con una pila accetta. Scopriremo che i PDA con due stack accettano gli stessi linguaggi delle macchine di Turing.

Consideriamo poi una classe di macchine dette "a contatori". Esse possono solamente memorizzare un numero finito di interi (detti "contatori") e scegliere fra mosse diverse a seconda che uno dei contatori valga 0. Una macchina a contatori può solo sommare o sottrarre 1 dai contatori e non è in grado di distinguere due valori differenti, diversi da 0. Di fatto un contatore è uno stack in cui si possono collocare solo due simboli: un segnale di fondo-stack, che compare esclusivamente al fondo, e un altro simbolo, che si può mettere e togliere.

Non svolgeremo una trattazione formale delle macchine multistack; il concetto è illustrato dalla Figura 8.20. Una macchina con k-stack è un PDA deterministico dotato di k stack. Al pari di un PDA, esso riceve l'input da una sorgente anziché leggerlo su un

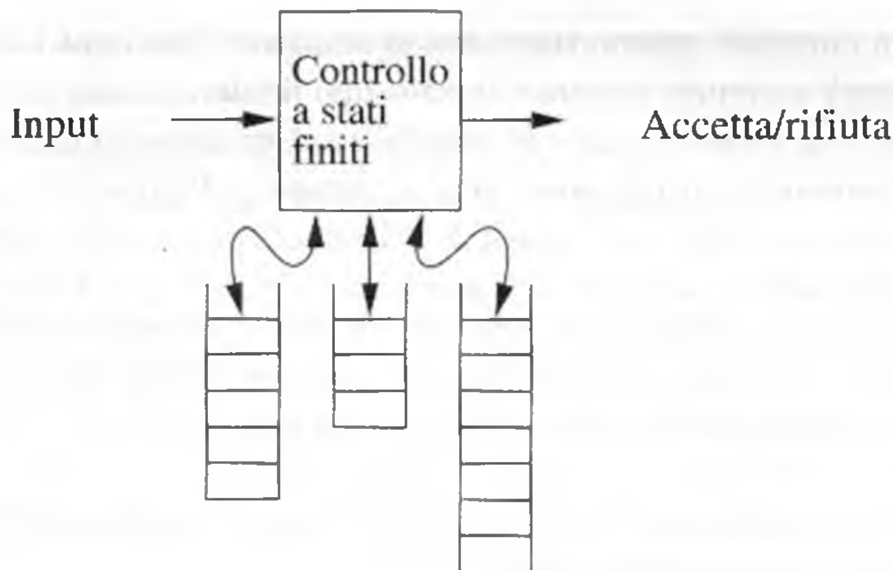


Figura 8.20 Una macchina con tre stack.

nastro o su uno stack come fa una TM. Una macchina multistack ha un controllo finito, cioè uno stato preso da un insieme finito; ha un alfabeto di stack finito, lo stesso per tutti gli stack; ogni sua mossa dipende da tre elementi.

1. Lo stato di controllo.

2. Il simbolo corrente di input, scelto in un alfabeto finito di input. In alternativa la macchina multistack può fare una mossa su input ϵ; in qualsiasi stato, però, non può esserci una scelta fra una ϵ-mossa e una mossa "normale" perché la macchina è deterministica.

3. I simboli in cima a ciascuno stack.

In una mossa la macchina compie diverse operazioni.

a) Cambia stato.

b) Sostituisce i simboli in cima agli stack con stringhe di zero o più simboli. Le stringhe sostituenti possono essere (e di solito sono) diverse da uno stack all'altro.

La tipica regola di transizione per una macchina con k stack ha quindi questa forma:

$$\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k)$$

La interpretiamo così: nello stato q, con X_i in cima all'i-esimo stack (per $i = 1, 2, \ldots, k$), la macchina può consumare a (un simbolo di input o ϵ) dall'input, passare allo stato p,

e sostituire X_i in cima all'i-esimo stack con la stringa γ_i, per ogni $i = 1, 2, \ldots, k$. Le macchine multistack accettano entrando in uno stato finale.

Aggiungiamo una caratteristica che semplifica il trattamento dell'input da parte di queste macchine deterministiche: assumiamo la presenza di un simbolo speciale \$, detto *segnale di fine*, che compare solo alla fine dell'input, ma non fa parte dell'input. La presenza del segnale di fine permette di sapere quando tutto l'input è stato consumato. Nel prossimo teorema vedremo che questo segnale semplifica la simulazione di una macchina di Turing da parte di una macchina multistack. Notiamo che la TM tradizionale non ne ha bisogno perché il primo blank segnala la fine dell'input.

Teorema 8.13 Se un linguaggio L è accettato da una macchina di Turing, allora L è accettato da una macchina con due stack.

DIMOSTRAZIONE Si sfrutta il fatto che due stack possono simulare il nastro di una macchina di Turing; uno stack conserva ciò che sta a sinistra della testina, l'altro ciò che sta a destra (sono escluse le stringhe infinite di blank oltre i simboli diversi dal blank più a sinistra e più a destra). In dettaglio, sia $L = L(M)$ per una TM M con un nastro. Descriviamo le operazioni della macchina con due stack S.

1. S comincia con un indicatore di fondo stack su ogni stack. Questo indicatore può essere il simbolo iniziale degli stack e non può comparire altrove. Nel seguito diremo che "uno stack è vuoto" quando contiene solo l'indicatore di fondo.

2. Sia $w\$$ l'input di S. S copia w nel primo stack fermandosi quando legge il segnale di fine dell'input.

3. S toglie, uno per volta, i simboli dal primo stack e li inserisce nel secondo. Ora il primo stack è vuoto, mentre il secondo contiene w, con l'estremo sinistro in cima.

4. S entra nello stato iniziale (simulato) di M. Il suo primo stack è vuoto, a rappresentare il fatto che a sinistra della cella guardata dalla testina di M ci sono solo blank. Il secondo stack di S contiene w, a rappresentare il fatto che w occupa le celle del nastro di M, quella guardata dalla testina e quelle alla sua destra.

5. Descriviamo ora come S simula una mossa di M.

   (a) S conosce lo stato di M perché lo simula nel proprio controllo.

   (b) S conosce il simbolo X che la testina di M sta guardando: è quello in cima al suo secondo stack. Si ha un'eccezione quando il secondo stack contiene solo l'indicatore di fondo; ciò significa che M si è spostato su un blank ed S può interpretare correttamente la situazione.

   (c) Perciò S conosce la mossa successiva di M.

(d) Il nuovo stato di M viene registrato in un componente del controllo di S al posto del precedente.

(e) Se M sostituisce X con Y e va a destra, S pone Y sul primo stack a rappresentare il fatto che ora Y è a sinistra della testina di M. X viene tolto dal secondo stack di S. Ci sono due eccezioni.

i. Se contiene solo l'indicatore di fondo (e quindi X è un blank), il secondo stack non viene modificato: M si è spostato su un altro blank più a destra.

ii. Se Y è un blank e il primo stack è vuoto, il primo stack rimane vuoto. Il motivo è che a sinistra della testina di M ci sono ancora solo blank.

(f) Se M sostituisce X con Y e va a sinistra, S toglie il simbolo in cima al primo stack (diciamo Z) e sostituisce X con ZY nel secondo stack. In questo modo si tiene conto del fatto che il simbolo che era immediatamente a sinistra della testina ora è sotto di essa. Si ha un'eccezione quando Z è il segnale di fondo; M deve allora porre BY sul secondo stack senza togliere nulla dal primo.

6. S accetta se il nuovo stato di M è accettante, altrimenti simula un'altra mossa di M nello stesso modo.

□

8.5.3 Macchine a contatori

Ci sono due modi di vedere una *macchina a contatori*.

1. Una macchina con la stessa struttura delle macchine multistack (Figura 8.20), ma con un contatore al posto di ogni stack. I contatori memorizzano interi non negativi, ma possiamo distinguere solo un valore nullo da uno non nullo. Dunque la mossa di una macchina a contatori dipende dallo stato, dal simbolo di input e da quali contatori, se ce ne sono, hanno valore nullo. In una mossa la macchina compie due operazioni.

   (a) Cambia stato.

   (b) Somma o sottrae 1 da un contatore. Poiché un contatore non può diventare negativo, la macchina non può sottrarre 1 da un contatore nullo.

2. Una macchina multistack con i vincoli seguenti.

   (a) Ci sono due soli simboli di stack, che denoteremo Z_0 (*indicatore di fondo stack*) e X.

   (b) All'inizio ogni stack contiene Z_0.

(c) Possiamo sostituire Z_0 solo con una stringa della forma $X^i Z_0$ per un $i \geq 0$.

(d) Possiamo sostituire X solo con X^i per un $i \geq 0$. Il simbolo Z_0 può quindi trovarsi soltanto in fondo agli stack; ogni altro simbolo nello stack è X.

Ci serviremo della definizione (1) di macchina a contatori, ma è chiaro che le due definizioni sono equivalenti. Infatti uno stack $X^i Z_0$ si può identificare con il valore i. Nella definizione (2) possiamo distinguere un contatore nullo dagli altri perché vediamo Z_0 anziché X alla sommità dello stack. Non possiamo invece distinguere fra loro due contatori non nulli perché entrambi hanno X in cima allo stack.

8.5.4 La potenza delle macchine a contatori

Per ciò che riguarda i linguaggi accettati da macchine a contatori, alcune osservazioni, per quanto ovvie, sono degne di nota.

- Ogni linguaggio accettato da una macchina a contatori è ricorsivamente enumerabile. Infatti una macchina a contatori è un caso speciale di macchina multistack, e questa è un caso speciale di macchina di Turing multinastro, che per il Teorema 8.9 accetta solo linguaggi ricorsivamente enumerabili.

- Ogni linguaggio accettato da una macchina con un contatore è un CFL. Secondo la Definizione (2) un contatore è uno stack; dunque una macchina con un contatore è un caso particolare di macchina con uno stack, cioè un PDA. I linguaggi delle macchine monocontatore sono in effetti accettati da PDA deterministici, ma la dimostrazione è sorprendentemente complessa. La difficoltà nasce dal fatto che le macchine multistack e a contatori hanno un segnale $ alla fine dell'input. Un PDA non deterministico può scommettere di aver letto l'ultimo simbolo di input e di dover leggere $; quindi un PDA non deterministico privo del segnale di fine input può chiaramente simulare un DPDA con quel segnale. La parte difficile, che non tratteremo, è la prova che un DPDA senza segnale può simulare un DPDA con il segnale.

Le macchine a contatori danno un risultato sorprendente: due contatori sono sufficienti per simulare una macchina di Turing, e quindi per accettare qualsiasi linguaggio ricorsivamente enumerabile. Ce ne occupiamo ora, dimostrando dapprima che bastano tre contatori, e poi simulandone tre con due soltanto.

Teorema 8.14 Ogni linguaggio ricorsivamente enumerabile è accettato da una macchina con tre contatori.

DIMOSTRAZIONE Partiamo dal Teorema 8.13, secondo cui ogni linguaggio ricorsivamente enumerabile è accettato da una macchina con due stack. Dobbiamo spiegare come si simula uno stack tramite contatori. Supponiamo che la macchina a stack usi $r - 1$ simboli di nastro. Possiamo identificarli con le cifre da 1 a $r - 1$ e trattare uno stack $X_1 X_2 \cdots X_n$ come un intero in base r. Lo stack, con la sommità a sinistra come al solito, sarebbe dunque rappresentato dall'intero $X_n r^{n-1} + X_{n-1} r^{n-2} + \cdots + X_2 r + X_1$.

Ci serviamo di due contatori per memorizzare gli interi che rappresentano i due stack. Il terzo contatore serve a regolare gli altri due. In particolare serve a dividere o moltiplicare un numero per r.

Le operazioni su uno stack sono di tre tipi: togliere il simbolo alla sommità, scambiarlo con un altro, immettere un nuovo simbolo. Una mossa della macchina con due stack può richiedere più di un'operazione; in particolare la sostituzione del simbolo alla sommità, X, con una stringa si divide nella sostituzione di X e nel successivo inserimento di simboli. Descriviamo ora come svolgere queste operazioni su uno stack rappresentato da un valore i. Notiamo che per compiere un'operazione che richiede di contare fino a r, o meno, si può usare il controllo della macchina multistack.

1. Per togliere un simbolo dallo stack dobbiamo sostituire i con i/r scartando l'eventuale resto, che è X_1. Partendo con il valore 0 nel terzo contatore, sottraiamo ripetutamente r da i sommando 1 al terzo contatore. Ci fermiamo quando il contatore che valeva i raggiunge lo 0. Ora sommiamo ripetutamente 1 al contatore originale sottraendo 1 dal terzo finché il terzo ritorna a 0. A questo punto il contatore che valeva i vale i/r.

2. Per scambiare X con Y in cima a uno stack rappresentato dal valore i, aumentiamo o diminuiamo i di una quantità non più grande di r (ricordiamo che in cima allo stack c'è la cifra meno significativa n.d.r.). Se $Y > X$ sommiamo $Y - X$ a i; se invece $Y < X$ sottraiamo $X - Y$ da i.

3. Per immettere X in uno stack che vale i, dobbiamo sostituire i con $ir + X$. Moltiplichiamo prima per r. Per farlo, sottraiamo ripetutamente 1 da i sommando r al terzo contatore (che parte sempre da 0). Quando il contatore originale diventa 0, il terzo vale ir. Copiamo il terzo contatore in quello originale e riportiamo a 0 il terzo, come nel punto (1). Infine sommiamo X al contatore originale.

Per completare la costruzione dobbiamo predisporre i contatori in modo che simulino le condizioni iniziali degli stack con il solo simbolo iniziale della loro macchina. Questo passo si compie aumentando il valore dei due contatori di una quantità pari al numero, fra 1 e $r - 1$, che corrisponde al simbolo iniziale. □

Teorema 8.15 Ogni linguaggio ricorsivamente enumerabile è accettato da una macchina con due contatori.

Scelta delle costanti nella simulazione da 3 a 2 contatori

Nella dimostrazione del Teorema 8.15 è fondamentale che 2, 3 e 5 siano numeri primi distinti. Se scegliessimo per esempio $m = 2^i 3^j 4^k$, il valore $m = 12$ potrebbe rappresentare sia $i = 0$, $j = 1$ e $k = 1$, sia $i = 2$, $j = 1$ e $k = 0$. Non potremmo quindi stabilire se i o k vale 0 e non potremmo simulare fedelmente la macchina a tre contatori.

DIMOSTRAZIONE In virtù del teorema precedente dobbiamo solo mostrare come si simulino tre contatori per mezzo di due. A questo scopo rappresentiamo i tre contatori, i, j e k, con un solo numero intero, e cioè $m = 2^i 3^j 5^k$. Un contatore memorizza questo numero; l'altro serve a moltiplicare o dividere m per uno dei tre numeri primi: 2, 3 e 5. Per simulare la macchina con tre contatori dobbiamo svolgere tre operazioni.

1. Sommare 1 a i, j o k. Per sommare 1 a i moltiplichiamo m per 2. Nella dimostrazione del Teorema 8.14 abbiamo visto come moltiplicare un contatore per una costante r usando un secondo contatore. In modo analogo incrementiamo j moltiplicando m per 3, e k moltiplicando m per 5.

2. Stabilire quali tra i, j e k valgono 0. Per stabilire se $i = 0$ dobbiamo determinare se m è divisibile per 2. Copiamo m nel secondo contatore e usiamo lo stato della macchina a contatori per ricordare se abbiamo diminuito m per un numero di volte pari o dispari. Quando m diventa 0, se l'abbiamo diminuito un numero dispari di volte, allora $i = 0$. Ripristiniamo ora m copiando il secondo contatore nel primo. Con la stessa tecnica verifichiamo se $j = 0$ stabilendo se m è divisibile per 3, e verifichiamo se $k = 0$ stabilendo se m è divisibile per 5.

3. Sottrarre 1 da i, j o k. A questo scopo dividiamo m per 2, 3 o 5. La dimostrazione del Teorema 8.14 spiega come dividere per una costante servendosi di un contatore ausiliario. La macchina a tre contatori non può far scendere sotto lo 0 i suoi contatori; è quindi un errore, e la macchina simulatrice si arresta senza accettare, se m non è divisibile per la costante in esame.

□

8.5.5 Esercizi

Esercizio 8.5.1 Descrivete informalmente, ma chiaramente, le macchine a contatori che accettano i linguaggi seguenti. In ciascun caso usate il minor numero possibile di contatori, ma mai più di due.

* a) $\{0^n 1^m \mid n \geq m \geq 1\}$.

b) $\{0^n 1^m \mid 1 \leq m \leq n\}$.

*! c) $\{a^i b^j c^k \mid i = j \text{ o } i = k\}$.

!! d) $\{a^i b^j c^k \mid i = j \text{ o } i = k \text{ o } j = k\}$.

! **Esercizio 8.5.2** Lo scopo di questo esercizio è dimostrare che una macchina monostack con segnale di fine input non è più capace di un PDA deterministico. $L\$$ è la concatenazione del linguaggio L con il linguaggio contenente la sola stringa \$; quindi $L\$$ è l'insieme delle stringhe $w\$$ tali che w è in L. Dimostrate che, se $L\$$ è accettato da un DPDA in cui \$ è il segnale di fine input e non compare nelle stringhe di L, allora anche L è accettato da un DPDA. *Suggerimento*: si tratta in realtà di dimostrare che i linguaggi dei DPDA sono chiusi rispetto all'operazione L/a definita nell'Esercizio 4.2.2. Dovete modificare il DPDA P per $L\$$ sostituendo ognuno dei suoi simboli di stack con tutte le coppie (X, S), dove S è un insieme di stati. A uno stack $X_1 X_2 \cdots X_n$ in P corrisponde, nel DPDA per L, lo stack $(X_1, S_1)(X_2, S_2) \cdots (X_n, S_n)$, dove ogni S_i è l'insieme degli stati q tali che P accetta a partire dalla ID $(q, a, X_i X_{i+1} \cdots X_n)$.

8.6 Le macchine di Turing e i computer

Confrontiamo ora la macchina di Turing e un computer ordinario. Possono sembrare modelli alquanto diversi, ma in effetti accettano esattamente gli stessi linguaggi, ossia i linguaggi ricorsivamente enumerabili. Poiché il concetto di "computer ordinario" non è definito in termini matematici precisi, la trattazione di questo paragrafo è necessariamente informale. Per quanto riguarda le capacità dei computer ci affidiamo all'intuizione, specialmente quando i numeri coinvolti eccedono i limiti insiti nell'architettura di queste macchine (per esempio spazi di indirizzamento a 32 bit). Le affermazioni del presente capitolo possono essere suddivise in due parti.

1. Un computer può simulare una macchina di Turing.

2. Una macchina di Turing può simulare un computer in tempo pari al più a un polinomio nel numero di operazioni svolte dal computer.

8.6.1 Simulazione di una macchina di Turing da parte di un computer

Cominciamo a esaminare come un computer può simulare una macchina di Turing. Data una particolare TM M, dobbiamo scrivere un programma che agisce come M. Un aspetto

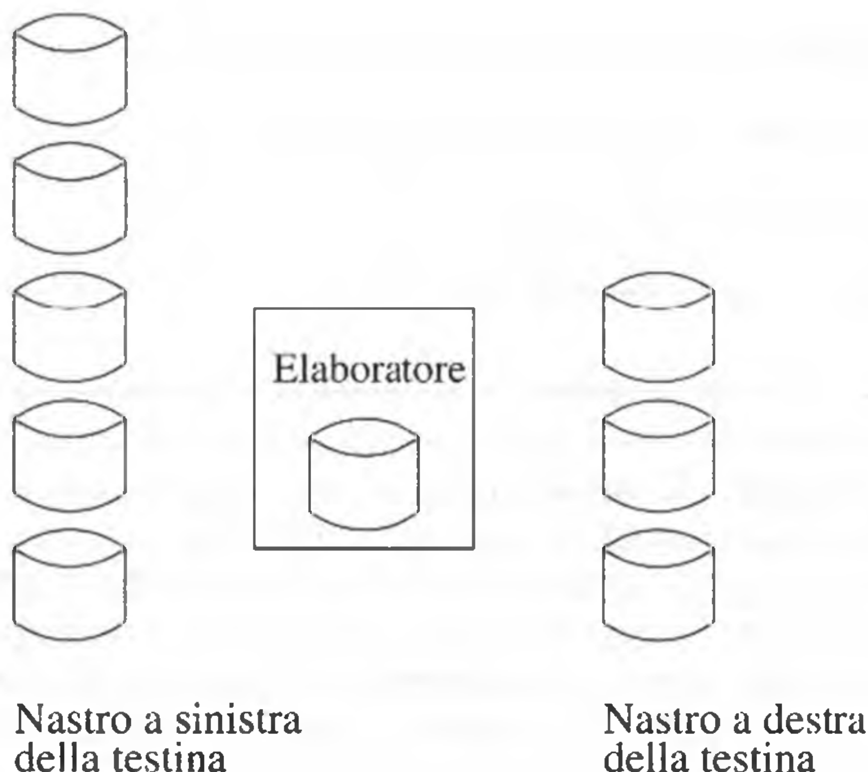


Figura 8.21 Simulazione di una macchina di Turing tramite un comune computer.

di M è il suo controllo finito. Poiché esistono solo un numero finito di stati e un numero finito di regole di transizione, il nostro programma può codificare gli stati come stringhe di caratteri e usare una tabella di transizione che viene consultata per stabilire ciascuna mossa. Analogamente i simboli di nastro possono essere codificati come stringhe di caratteri di lunghezza fissa, visto che ne esiste solo un numero finito.

Una domanda importante si pone quando consideriamo come il nostro programma debba simulare il nastro della macchina di Turing. Si tratta di un nastro che può diventare infinitamente lungo, ma la memoria del computer, vale a dire la memoria centrale, il disco e altri dispositivi di memorizzazione, sono tutti finiti. È possibile simulare un nastro infinito per mezzo di una quantità fissa di memoria?

Se non c'è nessuna opportunità di sostituire i dispositivi di memorizzazione, allora è effettivamente impossibile. In questo caso un computer sarebbe un automa a stati finiti e gli unici linguaggi che potrebbe accettare sarebbero linguaggi regolari. I computer comuni hanno però dispositivi di memoria rimovibili, come per esempio le unità "Zip". In effetti il tipico disco fisso è rimovibile e può essere sostituito da un disco vuoto, ma per il resto identico.

Dato che non esistono limiti evidenti al numero di dischi che possono essere usati, assumiamo che siano disponibili tanti dischi quanti ne richiede il computer. Possiamo perciò predisporre che i dischi siano sistemati in due pile, come suggerito nella Figu-

Il problema di alfabeti molto ampi

Il ragionamento del Paragrafo 8.6.1 diventa discutibile se il numero di simboli di nastro è così grande che il codice di un simbolo non può stare in un disco. Ce ne dovrebbero essere davvero molti, visto che per esempio un disco da 30 GB può rappresentare $2^{240000000000}$ simboli. Anche il numero di stati potrebbe essere così alto da non permetterci di rappresentarne uno neppure usando tutto il disco.

Per risolvere il problema si può anzitutto limitare il numero di simboli di nastro. Possiamo codificare in binario qualsiasi alfabeto. Perciò ogni TM M può essere simulata da un'altra TM M' con tre simboli di nastro: 0, 1 e B. Però M' ha bisogno di molti stati. Per simulare una mossa di M, deve percorrere il nastro e ricordare nel controllo tutti i bit che indicano il simbolo guardato da M. Gli insiemi di stati diventano enormi e può darsi che il PC che simula M' debba smontare e rimontare svariati dischi per decidere qual è lo stato di M' e quale la mossa successiva. Nessuno vorrebbe usare un computer così, e dunque i sistemi operativi non hanno supporto per programmi di questo tipo. Si può comunque programmare il computer "grezzo" per dotarlo di questa capacità.

Per fortuna il problema di simulare una TM con un gran numero di stati o simboli di nastro ammette un'altra soluzione. Vedremo nel Paragrafo 9.2.3 come definire una TM a "programma memorizzato". Questa TM, detta "universale", riceve sul nastro la funzione di transizione di un'altra TM, codificata in binario, e la simula. La TM universale ha un ragionevole numero di stati e di simboli di nastro. Simulando la TM universale, un computer ordinario può accettare qualunque linguaggio ricorsivamente enumerabile senza dover ricorrere alla simulazione di un numero di stati che superi i limiti di memoria del disco.

ra 8.21. Una pila contiene i dati nelle celle del nastro della macchina di Turing che sono poste a una certa distanza a sinistra della testina; l'altra pila contiene i dati a una certa distanza a destra della testina. Più in basso si scende nella pila, più i dati si allontanano dalla testina.

Se la testina della TM si muove verso sinistra fino a raggiungere celle che non sono rappresentate nel disco montato nel computer, allora il programma stampa il messaggio "cambio sinistro". Il disco montato correntemente viene rimosso da un operatore e posto in cima alla pila destra. Il disco in cima alla pila sinistra viene montato nel computer e si riprende la computazione.

Allo stesso modo, se la testina della TM raggiunge celle così lontane a destra da non essere rappresentate nel disco montato, allora viene stampato il messaggio "cambio destro". L'operatore sposta in cima alla pila sinistra il disco montato correntemente e

monta nel computer il disco in cima alla pila destra. Se il computer richiede di montare un disco di una certa pila, ma questa è vuota, ciò significa che la TM è entrata in una regione totalmente bianca del nastro. In questo caso l'operatore deve andare in magazzino e procurarsi un nuovo disco da montare.

8.6.2 Simulazione di un computer da parte di una macchina di Turing

Consideriamo ora il problema opposto: esistono funzioni che un computer comune può svolgere e una macchina di Turing no? Un'importante domanda subordinata è: un computer può svolgere determinate funzioni più velocemente di una macchina di Turing? Nel presente paragrafo dimostriamo che una TM può simulare un computer e nel Paragrafo 8.6.3 proviamo che la simulazione può essere svolta in modo efficiente, nel senso che i tempi di esecuzione del computer e della TM su un dato problema sono separati "solo" da un polinomio. Ricordiamo nuovamente al lettore che ci sono ragioni importanti per considerare simili i tempi di esecuzione in rapporto polinomiale, mentre differenze esponenziali nei tempi di esecuzione sono "eccessive". Riprenderemo la teoria dei tempi di esecuzione polinomiali confrontati con quelli esponenziali nel Capitolo 10.

Per cominciare lo studio della simulazione di un computer da parte di una TM, diamo un modello realistico, per quanto informale, di come opera un tipico computer.

a) Innanzitutto supporremo che la memoria di un computer consista di una sequenza indefinitamente lunga di *parole*, ciascuna con un *indirizzo*. In un computer reale le parole possono essere lunghe 32 oppure 64 bit, ma qui non porremo limiti di lunghezza. Assumiamo che gli indirizzi siano gli interi 0, 1, 2, e così via. In un computer reale i singoli byte sono numerati da interi consecutivi. Pertanto le parole hanno indirizzi multipli di 4 o 8, ma è una differenza irrilevante. In un computer reale ci sarebbe inoltre un limite al numero di parole contenute nella "memoria"; dato che vogliamo trattare il contenuto di un numero arbitrario di dischi o altri dispositivi di memoria, assumeremo che da questo punto di vista non esista limite.

b) Supponiamo che il programma del computer sia memorizzato in alcune parole di memoria, ognuna delle quali rappresenta una semplice istruzione (come nel linguaggio macchina o assembly di un tipico computer). Ne sono esempi le istruzioni che spostano i dati da una parola a un'altra o che sommano una parola a un'altra. Assumiamo che sia permesso l'"indirizzamento indiretto", per cui un'istruzione può fare riferimento a un'altra parola e usarne il contenuto come indirizzo della parola cui viene applicata l'operazione. Tale capacità, che ritroviamo in tutti i computer moderni, è necessaria per accedere a un array, per seguire i link di una lista o, in generale, per compiere operazioni su puntatori.

c) Supponiamo che ciascuna istruzione tocchi un numero limitato (finito) di parole e che ciascuna istruzione cambi il valore di non più di una parola.

d) Un computer tipico dispone di *registri*, che sono parole di memoria con un accesso particolarmente rapido. Spesso ci sono limitazioni tali che operazioni come l'addizione si svolgono solo nei registri. Qui non faremo queste restrizioni e permetteremo che qualunque operazione venga svolta su qualunque parola. Non si terrà conto della velocità relativa delle operazioni su parole diverse: quando ci limitiamo a confrontare le capacità di riconoscimento di linguaggi dei computer e delle macchine di Turing, non è necessario. Anche se siamo interessati a tempi di esecuzione a meno di un polinomio, le velocità relative di accessi a parole diverse sono irrilevanti, dato che le differenze costituiscono "solo" un fattore costante.

La Figura 8.22 suggerisce come si potrebbe definire la macchina di Turing perché simuli un computer. Tale TM usa diversi nastri, ma potrebbe essere convertita in una TM a nastro unico usando la costruzione del Paragrafo 8.4.1. Il primo nastro rappresenta l'intera memoria del computer. Abbiamo usato un codice in cui gli indirizzi delle parole di memoria, in ordine numerico, si alternano con i contenuti delle parole di memoria stesse. Sia gli indirizzi sia i contenuti sono scritti in binario. I simboli marcatori * e # facilitano il riconoscimento del termine degli indirizzi e dei contenuti, e indicano se una stringa binaria è un indirizzo oppure un contenuto. Un altro marcatore, $, indica l'inizio della sequenza di indirizzi e contenuti.

Il secondo nastro è il "contatore di istruzione" e contiene un unico intero in binario, che rappresenta una delle locazioni di memoria sul nastro 1. Il valore memorizzato in questa locazione sarà interpretato come la prossima istruzione da eseguire.

Il terzo nastro contiene un "indirizzo di memoria" o il suo contenuto dopo che esso è stato individuato sul nastro 1. Per eseguire un'istruzione, la TM deve trovare il contenuto di uno o più indirizzi di memoria che contengono dati usati nella computazione. In primo luogo l'indirizzo desiderato viene copiato sul nastro 3 e confrontato con gli indirizzi sul nastro 1 finché si ottiene una corrispondenza. Il contenuto di tale indirizzo viene copiato sul terzo nastro e spostato ovunque ce ne sia bisogno, di solito in uno degli indirizzi bassi, che rappresentano i registri del computer.

La TM simula il *ciclo-istruzione* del computer nel modo seguente.

1. Percorriamo il primo nastro alla ricerca di un indirizzo che corrisponda al numero di istruzione sul nastro 2. Partiamo dal $ sul primo nastro e ci muoviamo verso destra, confrontando ogni indirizzo con i contenuti del nastro 2. Il confronto di indirizzi sui due nastri è facile, dato che ci basta muovere le testine verso destra, in tandem, controllando che i simboli guardati siano sempre gli stessi.

2. Quando troviamo l'indirizzo dell'istruzione esaminiamo il suo valore. Assumiamo che quando una parola è un'istruzione i suoi primi bit rappresentano l'azione

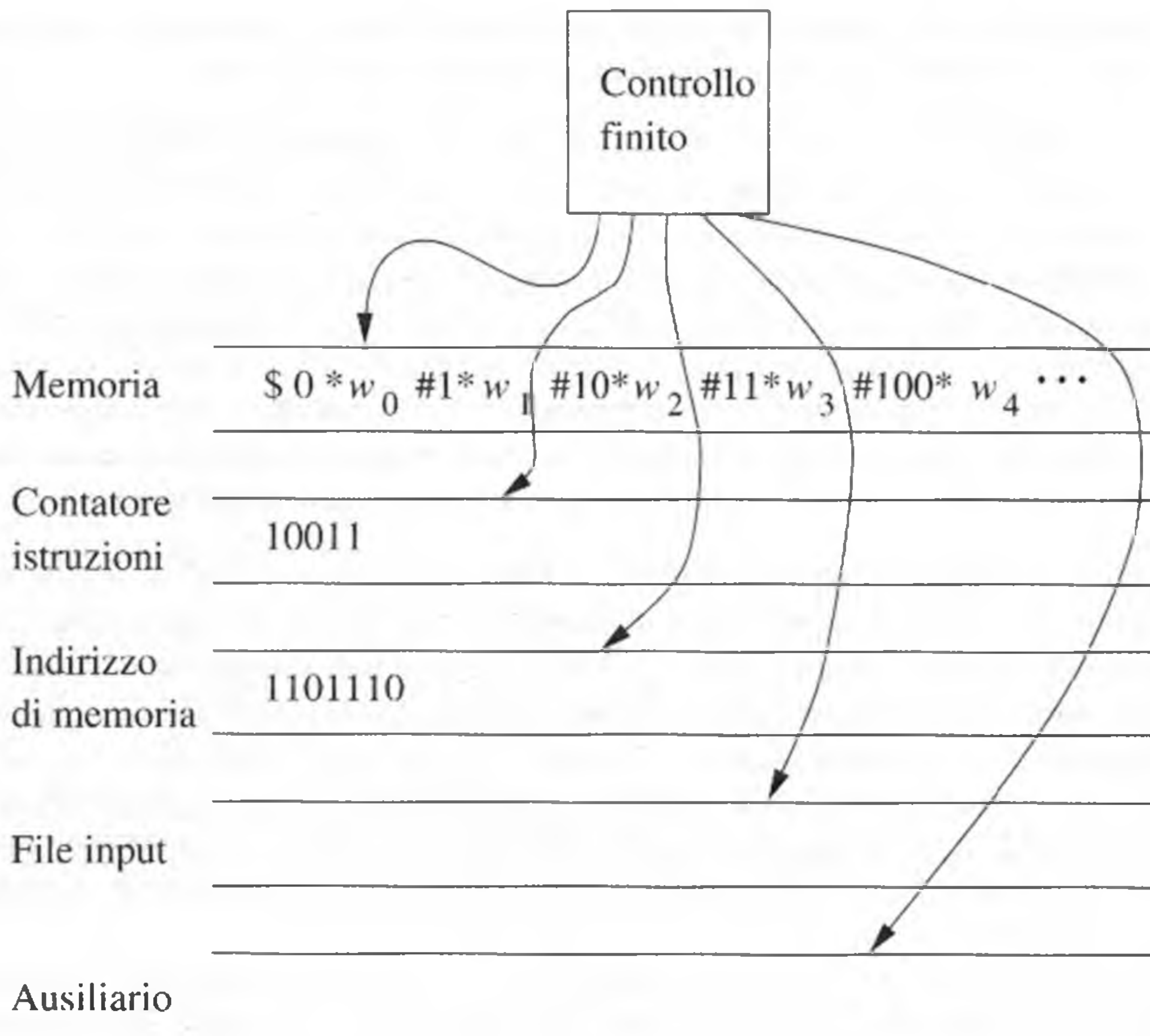


Figura 8.22 Una macchina di Turing che simula un computer ordinario.

da svolgere (per esempio copia, somma, salto) e quelli rimanenti codificano un indirizzo o più indirizzi coinvolti nell'operazione.

3. Se l'istruzione richiede il valore di un indirizzo, esso ne farà parte. Copiamo tale indirizzo sul terzo nastro e segnaliamo la posizione dell'istruzione usando una seconda traccia del primo nastro (che non è rappresentata nella Figura 8.22), in modo tale da poter tornare all'istruzione qualora sia necessario. Cerchiamo ora l'indirizzo di memoria sul primo nastro e copiamo il suo valore sul nastro 3, il nastro che contiene l'indirizzo di memoria.

4. Eseguiamo l'istruzione o la parte di istruzione che si riferisce a questo valore. Non possiamo trattare tutte le possibili istruzioni macchina, ma possiamo mostrare un campione dei tipi di azioni che potremmo svolgere con il nuovo valore.

(a) Copiare il valore in un altro indirizzo. Otteniamo il secondo indirizzo dall'istruzione; lo troviamo collocandolo sul nastro 3 e cercandolo sul nastro 1, come visto in precedenza. Dopo averlo trovato, copiamo il valore nello spazio riservato al valore dell'indirizzo. Se c'è bisogno di uno spazio maggiore per il nuovo valore oppure il nuovo valore usa meno spazio del vecchio, cambiamo lo spazio disponibile in questo modo.

   i. Copiamo su un nastro ausiliario l'intero nastro non bianco a destra del punto in cui va a porsi il nuovo valore.

   ii. Scriviamo il nuovo valore usando uno spazio adeguato.

   iii. Copiamo nuovamente il nastro ausiliario sul nastro 1, immediatamente a destra del nuovo valore.

   Può succedere, in un caso particolare, che l'indirizzo non compaia ancora sul primo nastro perché non è stato usato dal computer in precedenza. In tal caso troviamo il punto nel primo nastro in cui dovrebbe stare, operiamo come sopra per creare uno spazio adeguato, e vi memorizziamo sia l'indirizzo sia il nuovo valore.

(b) Sommiamo il valore appena trovato al valore di un altro indirizzo. Torniamo all'istruzione per localizzare l'altro indirizzo, che troveremo sul nastro 1. Facciamo una somma binaria del valore di tale indirizzo e del valore memorizzato sul nastro 3. Percorrendo i due valori a partire dalle loro estremità a destra, una TM può svolgere facilmente una somma con riporto. Se per il risultato ci volesse maggiore spazio, usiamo la tecnica descritta per creare spazio sul nastro 1.

(c) L'istruzione è un "salto", cioè la direttiva di prendere la prossima istruzione dall'indirizzo che è il valore memorizzato sul nastro 3. Copiamo semplicemente il nastro 3 sul nastro 2 e ricominciamo il ciclo-istruzione.

5. Dopo aver eseguito l'istruzione e determinato che non si tratta di un salto, aggiungiamo 1 al contatore di istruzione sul nastro 2 e ricominciamo il ciclo-istruzione.

Ci sono molti altri dettagli relativi a questa simulazione. Dato che il computer deve leggere il suo input (la parola di cui sta verificando l'appartenenza a un linguaggio) da un file, nella Figura 8.22 abbiamo suggerito un quarto nastro che contiene l'input simulato e dal quale la TM può leggerlo.

Si rappresenta inoltre un nastro ausiliario. La simulazione delle istruzioni di un computer potrebbe effettivamente impiegare uno o più nastri ausiliari per svolgere operazioni aritmetiche come la moltiplicazione.

Infine assumiamo che il computer produca un output che indica se il suo input viene accettato o no. Per tradurre quest'azione in termini eseguibili dalla macchina di Turing,

ipotizzeremo che esista un'istruzione del computer che chiamiamo "accettare", eventualmente legata a una chiamata di funzione da parte del computer per scrivere `sì` su un file di output. Quando simula l'esecuzione di quest'istruzione del computer, la TM entra in un suo stato accettante e si arresta.

Se questa trattazione non è la dimostrazione formale e completa che una TM può simulare un tipico computer, dovrebbe però fornire abbastanza dettagli per convincere il lettore che una TM costituisce una valida rappresentazione delle capacità di un computer. Di conseguenza in futuro useremo solo la macchina di Turing come rappresentazione formale di quanto può essere computato da un qualunque tipo di dispositivo di calcolo.

8.6.3 Confronto dei tempi di esecuzione dei computer e delle macchine di Turing

Dobbiamo ora affrontare il punto relativo al tempo di esecuzione di una macchina di Turing che simula un computer. Come suggerito precedentemente, possiamo fare tre importanti osservazioni.

- Il tempo di esecuzione è un aspetto importante perché ci serviremo della TM non solo per stabilire che cosa sia calcolabile in assoluto, ma anche quanto sia calcolabile con tanta efficienza da poter applicare una soluzione basata sull'uso di un computer.

- In genere si ritiene che lo spartiacque tra problemi *trattabili*, ossia quelli che possono essere risolti efficientemente, e *intrattabili*, ossia quelli che possono essere risolti, ma non abbastanza velocemente da rendere fruibile la soluzione, si collochi tra quanto può essere computato in un tempo polinomiale e quanto richiede un tempo di esecuzione maggiore di qualsiasi polinomio.

- Abbiamo dunque bisogno di assicurarci che se un problema può essere risolto in tempo polinomiale con un tipico computer, allora può essere risolto in tempo polinomiale da una macchina di Turing, e viceversa. Data questa "equivalenza polinomiale", le nostre conclusioni su ciò che una macchina di Turing può o non può fare con adeguata efficienza si applicano anche a un computer.

Ricordiamo che nel Paragrafo 8.4.3 abbiamo determinato che la differenza nel tempo di esecuzione tra una TM a nastro unico e una TM multinastro è polinomiale, e in particolare quadratica. Di conseguenza basta mostrare che qualunque cosa il computer è in grado di fare, la può fare anche la TM multinastro descritta nel Paragrafo 8.6.2, in tempo polinomiale rispetto al tempo richiesto dal computer. Sappiamo che la stessa conclusione è quindi valida anche per una TM a nastro unico.

Prima di fornire la dimostrazione che la macchina di Turing descritta può simulare n passi di un computer in un tempo $O(n^3)$, dobbiamo affrontare la questione della moltiplicazione come istruzione di un computer. Il problema è che non abbiamo posto limiti al numero di bit che possono essere contenuti in una parola. Se il computer dovesse partire con una parola contenente, per esempio, l'intero 2 e moltiplicarla per se stessa per n passi consecutivi, allora la parola conterrebbe il numero 2^{2^n}. Per essere rappresentato, questo numero richiede $2^n + 1$ bit; dunque il tempo impiegato dalla macchina di Turing per simulare queste n istruzioni sarebbe al minimo esponenziale in n.

Una soluzione consiste nel richiedere che le parole mantengano una lunghezza massima fissa, poniamo 64 bit. Allora le moltiplicazioni (o altre operazioni) che producessero una parola troppo lunga causerebbero l'arresto del computer, e la macchina di Turing non lo simulerebbe oltre. Saremo più tolleranti e permetteremo che il computer usi parole in grado di crescere fino a qualunque lunghezza. Faremo però in modo che un'istruzione possa produrre solo una parola che è più lunga di un bit rispetto al più lungo dei suoi argomenti.

Esempio 8.16 In conformità con la restrizione espressa sopra, la somma è permessa, dato che il risultato può essere di un solo bit più lungo della lunghezza massima degli addendi. La moltiplicazione non è invece permessa, in quanto due parole di m-bit possono avere un prodotto di lunghezza $2m$. Possiamo però simulare una moltiplicazione di interi di m-bit tramite una sequenza di m addizioni, inframmezzate da spostamenti del moltiplicando di un bit a sinistra (il che è un'altra operazione che aumenta la lunghezza della parola solo di un'unità). Di conseguenza possiamo ancora moltiplicare parole di lunghezza arbitraria, ma il tempo impiegato dal computer è proporzionale al quadrato della lunghezza degli operandi. □

Ipotizzando una crescita massima di un bit per ogni istruzione eseguita, siamo in grado di dimostrare il rapporto polinomiale tra i due tempi di esecuzione. L'idea alla base della dimostrazione consiste nel notare che dopo l'esecuzione di n istruzioni il numero di parole presenti sul nastro di memoria della TM è $O(n)$, e ogni parola richiede $O(n)$ celle della macchina di Turing per essere rappresentata. Perciò il nastro è lungo $O(n^2)$ celle e la TM può localizzare il numero finito di parole necessario per un'istruzione in un tempo $O(n^2)$.

Dobbiamo però imporre alle istruzioni un altro vincolo. Anche se un'istruzione non produce come risultato una parola lunga, potrebbe impiegare molto tempo per calcolare il risultato. Faremo quindi la supposizione aggiuntiva che una macchina di Turing multinastro possa eseguire l'istruzione stessa, applicata a parole di lunghezza fino a k, in $O(k^2)$ passi. Senz'altro le tipiche operazioni del computer, come l'addizione, lo spostamento e il confronto di valori, possono essere svolte in $O(k)$ passi della TM multinastro, e quindi il vincolo imposto non è troppo rigido.

Teorema 8.17 Se un computer:

1. ha solo istruzioni che aumentano la lunghezza massima delle parole di non più di un'unità

2. ha solo istruzioni che una TM multinastro può eseguire su parole di lunghezza k in $O(k^2)$ passi o meno

allora la macchina di Turing descritta nel Paragrafo 8.6.2 può simulare n passi del computer in $O(n^3)$ dei suoi passi.

DIMOSTRAZIONE Partiamo dall'osservazione che il primo nastro (la memoria) della TM nella Figura 8.22 contiene inizialmente solo il programma del computer. Il programma può essere lungo, ma è fisso e di lunghezza costante, indipendente da n, il numero dei passi di istruzione eseguiti dal computer. Esiste perciò una costante c, che è la parola più lunga o l'indirizzo più lungo tra quelli presenti nel programma. Inoltre esiste una costante d, che è il numero di parole occupate dal programma.

Di conseguenza, dopo aver eseguito n passi, il computer non può aver creato alcuna parola più lunga di $c + n$, e perciò non può aver creato o usato alcun indirizzo che a sua volta sia più lungo di $c + n$ bit. Ciascuna istruzione crea al massimo un unico nuovo indirizzo che prende un valore, per cui il numero totale di indirizzi dopo l'esecuzione di n istruzioni è al massimo $d + n$. Dato che ciascuna combinazione indirizzo-parola richiede al massimo $2(c + n) + 2$ bit, inclusi l'indirizzo, il contenuto e due simboli marcatori per separarli, il numero totale di celle di nastro della TM occupate dopo che n istruzioni sono state simulate è al massimo $2(d + n)(c + n + 1)$. Poiché c e d sono costanti, tale numero di celle è $O(n^2)$.

Sappiamo ora che ognuna delle ricerche di indirizzi, in numero fisso, coinvolte in un'unica istruzione, può essere eseguita in un tempo $O(n^2)$. Dato che le parole sono $O(n)$ in lunghezza, il nostro secondo assunto indica che ognuna delle istruzioni stesse può essere svolta dalla TM in un tempo $O(n^2)$. L'unico costo residuo e significativo di un'istruzione riguarda il tempo impiegato dalla TM per creare maggiore spazio sul suo nastro allo scopo di contenere una parola nuova oppure una estesa. La tecnica impiegata richiede di copiare al massimo $O(n^2)$ dati dal nastro 1 al nastro ausiliario, e viceversa. Perciò anch'essa richiede solo un tempo $O(n^2)$ per ogni istruzione del computer.

Il risultato è che la TM simula un passo del computer in $O(n^2)$ dei suoi passi. Di conseguenza, come abbiamo sostenuto nell'enunciato del teorema, n passi del computer possono essere simulati in $O(n^3)$ passi della macchina di Turing. □

Come osservazione finale risulta chiaro che, a condizione di elevare al cubo il numero di passi, una TM multinastro può simulare un computer. Sappiamo anche dal Paragrafo 8.4.3 che una TM a nastro unico può simulare una TM multinastro elevando al quadrato il numero dei passi. La conclusione è riassunta nel prossimo teorema.

Teorema 8.18 Un computer del tipo descritto nel Teorema 8.17 può essere simulato per n passi da una macchina di Turing a nastro unico impiegando $O(n^6)$ passi della macchina di Turing. □

8.7 Riepilogo

- *La macchina di Turing*: la TM è una macchina di calcolo astratta con la potenza sia dei computer reali sia di altre definizioni matematiche per quanto riguarda ciò che può essere calcolato. La TM consiste di un controllo a stati finiti e di un nastro infinito diviso in celle. Ciascuna cella contiene un simbolo di nastro; i simboli di nastro sono in numero finito. Una cella è la posizione corrente della testina del nastro. La TM compie mosse basate sul suo stato corrente e sul simbolo di nastro nella cella guardata dalla testina. In una mossa la TM cambia stato, soprascrive la cella corrente con un simbolo di nastro e muove la testina di una cella verso sinistra o verso destra.

- *Accettazione da parte di una macchina di Turing*: la TM parte con l'input, una stringa di lunghezza finita di simboli, sul nastro; il resto del nastro contiene il simbolo blank in ogni cella. Il blank è uno dei simboli di nastro e l'input è scelto da un sottoinsieme dei simboli di nastro, tranne il blank, detti simboli di input. La TM accetta il suo input se entra in uno stato accettante.

- *Linguaggi ricorsivamente enumerabili* : i linguaggi accettati dalle TM sono detti linguaggi ricorsivamente enumerabili (RE). I linguaggi RE sono dunque quelli che possono essere riconosciuti o accettati da un qualunque tipo di dispositivo di calcolo.

- *Descrizioni istantanee di una TM*: possiamo descrivere la configurazione corrente di una TM mediante una stringa di lunghezza finita che include tutte le celle del nastro, dal simbolo diverso dal blank più a sinistra a quello più a destra. Lo stato e la posizione della testina si indicano ponendo lo stato all'interno della sequenza dei simboli di nastro, alla sinistra della cella guardata.

- *Memoria nel controllo finito*: per definire una TM per un particolare linguaggio, conviene a volte pensare che lo stato abbia due o più componenti. Un componente è quello del controllo, e ha lo stesso funzionamento di un normale stato. Gli altri componenti contengono dati che la TM deve ricordare.

- *Tracce multiple*: spesso è utile considerare i simboli di nastro come vettori con un numero fisso di componenti. Possiamo visualizzare ciascun componente come una traccia distinta del nastro.

- *Macchine di Turing multinastro*: un modello di TM esteso ha un numero fisso di nastri, più grande di uno. Una mossa di questa TM è basata sullo stato e sul vettore di simboli guardati dalle testine su ognuno dei nastri. In una mossa la TM multinastro cambia stato, soprascrive i simboli sulle celle guardate da ognuna delle testine e sposta alcune o tutte le testine di una cella in una delle due direzioni. Per quanto sia in grado di riconoscere determinati linguaggi più rapidamente della convenzionale TM a nastro unico, la TM multinastro non riesce a riconoscere un linguaggio che non sia RE.

- *Macchine di Turing non deterministiche*: la NTM ha un numero finito di scelte della mossa seguente (stato, nuovo simbolo e movimento della testina) per ciascuno stato e ciascun simbolo guardato. Accetta un input se una qualunque tra le diverse sequenze di scelte porta a una ID con stato accettante. Per quanto apparentemente più potente della TM deterministica, la NTM non è in grado di riconoscere un linguaggio che non sia RE.

- *Macchine di Turing con nastro semi-infinito*: possiamo limitare una TM a un nastro che sia infinito solo verso destra e privo di celle a sinistra della posizione iniziale della testina. Una TM di questo tipo può accettare qualunque linguaggio RE.

- *Macchine multistack*: possiamo limitare i nastri di una TM multinastro in modo che si comportino come stack. L'input si trova su un nastro separato, che viene letto una volta da sinistra verso destra, a imitazione della modalità di input per un automa a stati finiti o un PDA. Una macchina a stack unico è in effetti un DPDA, mentre una macchina a due stack può accettare qualunque linguaggio RE.

- *Macchine a contatori*: possiamo limitare ulteriormente gli stack di una macchina multistack in modo che abbiano solo un simbolo diverso dall'indicatore di fondo. Perciò ciascuno stack funziona come un contatore e permette di memorizzare un intero non negativo e di verificare se l'intero memorizzato è 0, ma nient'altro. Una macchina con due contatori è sufficiente per accettare qualunque linguaggio RE.

- *Simulazione di una macchina di Turing da parte di un computer reale*: è possibile simulare una TM per mezzo di un computer reale se accettiamo che esista una riserva potenzialmente infinita di dispositivi di memoria rimovibili, come i dischi, per simulare la porzione non bianca del nastro della TM. Poiché le risorse materiali per produrre dischi non sono infinite, si tratta di un'argomentazione discutibile. Ma dato che la quantità di memoria esistente nell'universo è sconosciuta e indubbiamente vasta, ipotizzare una risorsa infinita, come il nastro delle TM, nella pratica è realistico e generalmente accettato.

- *Simulazione di un computer da parte di una macchina di Turing*: una TM può simulare la memoria e il controllo di un computer reale impiegando un nastro che

memorizza tutte le locazioni e il loro contenuto: registri, memoria centrale e altri dispositivi di memoria. Di conseguenza possiamo essere certi che un compito non eseguibile da una TM non può essere eseguito neppure da un computer reale.

8.8 Bibliografia

La macchina di Turing è tratta da [8]. Per caratterizzare che cosa può essere computato, all'incirca nello stesso periodo si fecero diverse proposte non fondate sulla nozione di macchina. Segnaliamo i lavori di Church [1], Kleene [5] e Post [7]. Tutti questi studi erano stati preceduti dai risultati di Gödel [3], che in effetti aveva dimostrato che un computer non può rispondere a tutte le domande matematiche.

Lo studio delle macchine di Turing multinastro, in particolare la questione di come il loro tempo di esecuzione si confronti con quello di un modello a nastro unico, cominciò con Hartmanis e Stearns [4]. L'esame delle macchine multistack e a contatori è tratto da [6], sebbene la costruzione presentata qui provenga da [2].

L'impiego di ciao-mondo come surrogato per l'accettazione o l'arresto da parte di una macchina di Turing, descritto nel Paragrafo 8.1, è contenuto in note inedite di S. Rudich.

1. A. Church, "An undecidable problem in elementary number theory," *American J. Math.* **58** (1936), pp. 345–363.

2. P. C. Fischer, "Turing machines with restricted memory access," *Information and Control* 9:4 (1966), pp. 364–379.

3. K. Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme," *Monatshefte für Mathematik und Physik* **38** (1931), pp. 173–198.

4. J. Hartmanis, R. E. Stearns, "On the computational complexity of algorithms," *Transactions of the AMS* **117** (1965), pp. 285–306.

5. S. C. Kleene, "General recursive functions of natural numbers," *Mathematische Annalen* **112** (1936), pp. 727–742.

6. M. L. Minsky, "Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines," *Annals of Mathematics* **74**:3 (1961), pp. 437–455.

7. E. Post, "Finite combinatory processes-formulation," *J. Symbolic Logic* **1** (1936), pp. 103–105.

8. A. M. Turing, "On computable numbers with an application to the Entscheidungsproblem," *Proc. London Math. Society* **2**:42 (1936), pp. 230–265. Vedi anche *ibid.* **2**:43, pp. 544–546.

Capitolo 9

Indecidibilità

Iniziamo questo capitolo ribadendo, nel contesto delle macchine di Turing, il ragionamento svolto nel Paragrafo 8.1, in cui sostenevamo l'esistenza di problemi che non possono essere risolti da un computer. La seconda "dimostrazione" che abbiamo proposto comporta una difficoltà: siamo stati costretti a ignorare i limiti concreti di cui soffre ogni implementazione del C (o di qualunque altro linguaggio di programmazione) in un vero computer. Questi limiti, come la dimensione dello spazio di indirizzamento, non sono fondamentali. Al contrario, possiamo aspettarci che con il passare degli anni certe quantità, per esempio la dimensione dello spazio di indirizzamento e la capacità della memoria centrale, aumenteranno indefinitamente.

Concentrandoci sulla macchina di Turing, che non conosce quelle restrizioni, possiamo cogliere meglio l'essenza delle capacità di un dispositivo di calcolo, se non oggi, almeno per il futuro. In questo capitolo diamo una dimostrazione formale dell'esistenza di un problema, attinente alle macchine di Turing, che nessuna macchina di Turing può risolvere. Poiché sappiamo dal Paragrafo 8.6 che le macchine di Turing possono simulare i computer concreti, anche quelli non soggetti alle limitazioni odierne, ragioneremo in modo rigoroso sul fatto che il seguente problema:

- una macchina di Turing data accetta (il codice di) se stessa come input?

non può essere risolto da un computer, per quanto se ne allentino i limiti pratici.

Distinguiamo dunque i problemi che possono essere risolti da una macchina di Turing in due classi: quelli per cui c'è un *algoritmo* (cioè una macchina di Turing che si arresta, a prescindere dal fatto di accettare o no il suo input) e quelli che vengono risolti solo da macchine di Turing che possono girare per sempre su input non accettati. La seconda forma di accettazione è problematica. Per quanto a lungo la TM giri, non è infatti dato sapere se l'input è accettato o no. Ci concentriamo dunque sulle tecniche per dimostrare che un problema è "indecidibile", ossia che non ha un algoritmo, indipendentemente dal

fatto che sia accettato o no da una macchina di Turing che non si arresta su un certo input. Dimostriamo l'indecidibilità del seguente problema:

- una data macchina di Turing accetta un dato input?

Sfruttiamo allora il risultato di indecidibilità per presentare diversi altri problemi indecidibili. Per esempio mostriamo che tutti i problemi non banali relativi al linguaggio accettato da una macchina di Turing sono indecidibili, così come una quantità di problemi che non hanno nulla a che vedere con macchine di Turing, programmi o computer.

9.1 Un linguaggio non ricorsivamente enumerabile

Ricordiamo che un linguaggio L è *ricorsivamente enumerabile* (RE) se $L = L(M)$ per una TM M. Nel Paragrafo 9.2 presenteremo i linguaggi "ricorsivi", o "decidibili", che non solo sono ricorsivamente enumerabili, ma sono accettati da una TM che si arresta sempre, a prescindere dal fatto che accetti o no.

Il nostro obiettivo finale è dimostrare l'indecidibilità del linguaggio formato dalle coppie (M, w) tali che:

1. M è una macchina di Turing (adeguatamente codificata in binario) con alfabeto di input $\{0, 1\}$
2. w è una stringa di 0 e di 1
3. M accetta l'input w.

Se questo problema, con input limitati all'alfabeto binario, è indecidibile, allora senz'altro è indecidibile anche il problema più generale, in cui le TM possono avere qualsiasi alfabeto.

Il primo passo consiste nel porre la questione come problema di appartenenza a un particolare linguaggio. Perciò dobbiamo dare una codifica delle macchine di Turing che usi solo 0 e 1, indipendentemente dal numero di stati. Una volta ottenuta la codifica, possiamo trattare qualunque stringa binaria come se fosse una macchina di Turing. Se la stringa non è una rappresentazione ben formata di una TM, possiamo pensarla come la rappresentazione di una TM priva di mosse. Di conseguenza possiamo considerare ogni stringa binaria come una TM.

L'obiettivo intermedio, che è anche l'argomento di questo paragrafo, tocca il linguaggio L_d, il "linguaggio di diagonalizzazione", che consiste di tutte le stringhe w tali che la TM rappresentata da w non accetta l'input w. Dimostreremo che non c'è alcuna TM che accetti L_d. Ricordiamo che la non esistenza di una macchina di Turing per un linguaggio

è una proprietà più forte dell'indecidibilità del linguaggio (cioè della non esistenza di un algoritmo o di una TM che si arresta sempre).

La funzione del linguaggio L_d è analoga a quella dell'ipotetico programma H_2 del Paragrafo 8.1.2, che stampa `Ciao, mondo` ogni volta che il suo input *non* stampa `Ciao, mondo` quando riceve se stesso come input. In termini più precisi, proprio come H_2 non può esistere perché quando riceve se stesso come input la sua reazione è paradossale, L_d non può essere accettato da una macchina di Turing. Infatti, se lo fosse, la TM in questione entrerebbe in disaccordo con se stessa nel momento in cui dovesse ricevere come input il proprio codice.

9.1.1 Enumerazione delle stringhe binarie

Nel seguito dovremo assegnare numeri interi a tutte le stringhe binarie in modo che ciascuna stringa corrisponda a un unico intero e ciascun intero corrisponda a un'unica stringa. Se w è una stringa binaria, trattiamo $1w$ come un intero binario i. Chiameremo allora w la i-esima stringa. In altre parole ϵ è la prima stringa, 0 la seconda, 1 la terza, 00 la quarta, 01 la quinta, e così via. Con questo criterio le stringhe risultano ordinate per lunghezza, con le stringhe di lunghezza uguale ordinate lessicograficamente. D'ora in avanti ci riferiremo alla i-esima stringa come w_i.

9.1.2 Codici per le macchine di Turing

Il prossimo obiettivo è ideare un codice binario per le macchine di Turing così che ogni TM con alfabeto di input $\{0, 1\}$ possa essere considerata come una stringa binaria. Poiché ora sappiamo enumerare le stringhe binarie, potremo identificare le macchine di Turing con gli interi e parlare della "i-esima macchina di Turing M_i". Per rappresentare una TM $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$ come una stringa binaria, è necessario assegnare interi agli stati, ai simboli di nastro e alle direzioni L ed R.

- Supporremo che gli stati siano $q_1, q_2, \ldots, q_r$ per un certo r. Lo stato iniziale sarà sempre q_1, e q_2 sarà l'unico stato accettante. Avendo ipotizzato che la TM si arresti quando entra in uno stato accettante, non c'è mai bisogno di più di uno stato accettante.

- Supporremo che i simboli di nastro siano $X_1, X_2, \ldots, X_s$ per un certo s. X_1 sarà sempre il simbolo 0, X_2 sarà 1 e X_3 sarà B, il blank. Gli altri simboli di nastro possono essere assegnati arbitrariamente agli altri interi.

- Ci riferiremo alla direzione L come D_1 e alla direzione R come D_2.

Dato che per ogni TM M possiamo assegnare interi ai suoi stati e ai simboli di nastro in molti modi diversi, per una TM ci sarà più di una codifica. Si tratta comunque di

un elemento irrilevante in quanto segue, perché mostreremo che nessuna codifica può rappresentare una TM M tale che $L(M) = L_d$.

Una volta scelti gli interi che rappresentano stati, simboli e direzioni, possiamo codificare la funzione di transizione δ. Supponiamo che una regola di transizione sia $\delta(q_i, X_j) = (q_k, X_l, D_m)$, per certi valori interi i, j, k, l ed m. Codifichiamo questa regola per mezzo della stringa $0^i10^j10^k10^l10^m$. Poiché i, j, k, l ed m valgono ciascuno almeno 1, nel codice di una transizione non compaiono mai due o più 1 consecutivi.

Un codice per l'intera TM M consiste di tutti i codici delle transizioni in un ordine fissato, separati da coppie di 1:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

dove ognuna delle C è il codice di una transizione di M.

Esempio 9.1 Consideriamo la TM

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$

dove δ consiste nelle regole:

$$\begin{aligned} \delta(q_1, 1) &= (q_3, 0, R) \\ \delta(q_3, 0) &= (q_1, 1, R) \\ \delta(q_3, 1) &= (q_2, 0, R) \\ \delta(q_3, B) &= (q_3, 1, L) \end{aligned}$$

I codici per le regole sono, rispettivamente:

$$\begin{array}{l} 0100100010100 \\ 0001010100100 \\ 00010010010100 \\ 0001000100010010 \end{array}$$

Per esempio la prima regola può essere scritta come $\delta(q_1, X_2) = (q_3, X_1, D_2)$ in quanto $1 = X_2$, $0 = X_1$ e $R = D_2$. Di conseguenza il suo codice è $0^1 10^2 10^3 10^1 10^2$, come indicato sopra. Un codice per M è:

$$01001000101001100010101001001100010010010100110001000100010010$$

Notiamo che esistono molti altri codici possibili di M. In particolare i codici per le quattro transizioni possono essere elencati in 4! ordini diversi, producendo 24 codici per M. □

Nel Paragrafo 9.2.3 dovremo codificare coppie costituite da una TM e da una stringa, (M, w). Per tale coppia usiamo il codice di M seguito da 111, seguito a sua volta da w. Poiché nessun codice valido di una TM contiene tre 1 in fila, possiamo essere certi che la prima occorrenza di 111 separa il codice di M da w. Per esempio, se M fosse la TM dell'Esempio 9.1 e w fosse 1011, allora il codice di (M, w) sarebbe la stringa mostrata alla fine dell'Esempio 9.1, seguita da 1111011.

9.1.3 Il linguaggio di diagonalizzazione

Nel Paragrafo 9.1.2 abbiamo codificato le macchine di Turing. Disponiamo quindi di una nozione concreta di M_i, la "i-esima macchina di Turing": la TM M il cui codice è w_i, l'i-esima stringa binaria. Molti interi non corrispondono a nessuna TM. Per esempio 11001 non comincia per 0 e 0010111010010100 ha tre 1 consecutivi. Se w_i non è un codice di TM valido, considereremo M_i come la TM con un unico stato e nessuna transizione. In altre parole, per questi valori di i, M_i è la macchina di Turing che si arresta immediatamente su qualunque input. Di conseguenza $L(M_i)$ è $\emptyset$ se w_i non è un codice valido.

A questo punto possiamo dare una definizione fondamentale.

- Il linguaggio L_d, detto *linguaggio di diagonalizzazione*, è l'insieme delle stringhe w_i tali che w_i non è in $L(M_i)$.

In altre parole L_d consiste di tutte le stringhe w tali che la TM M con codice w non accetta quando riceve w come input.

La ragione per cui L_d è detto linguaggio di "diagonalizzazione" è chiarita dalla Figura 9.1. La tabella indica per ogni i e j se la TM M_i accetta la stringa di input w_j; 1 significa "sì, accetta" e 0 significa "no, non accetta".[1] Possiamo interpretare l'i-esima riga come il *vettore caratteristico* del linguaggio $L(M_i)$; in altre parole gli 1 in questa riga indicano le stringhe che appartengono al linguaggio.

I valori sulla diagonale segnalano se M_i accetta w_i. Per costruire L_d complementiamo la diagonale. Per esempio, se la Figura 9.1 fosse la tabella corretta, allora la diagonale complementata comincerebbe con $1, 0, 0, 0, \ldots$. Perciò L_d conterrebbe $w_1 = \epsilon$, e non le stringhe da w_2 a w_4, che sono 0, 1, 00, e via di seguito.

Il metodo di complementare la diagonale per costruire il vettore caratteristico di un linguaggio che non compare in nessuna riga è detto *diagonalizzazione*. Esso funziona perché il complemento della diagonale è di per sé un vettore caratteristico che descrive l'appartenenza a un determinato linguaggio, e cioè a L_d. Questo vettore caratteristico si trova in disaccordo con ogni riga della tabella nella Figura 9.1 in almeno una colonna. Quindi il complemento della diagonale non può essere il vettore caratteristico di una macchina di Turing.

9.1.4 Dimostrazione che L_d non è ricorsivamente enumerabile

Sfruttando il ragionamento riguardante i vettori caratteristici e la diagonale, dimostreremo ora in termini formali un risultato fondamentale sulle macchine di Turing: nessuna macchina di Turing accetta L_d.

[1] Il lettore può osservare che la tabella reale è molto diversa da quella della figura. Poiché i numeri interi bassi non possono rappresentare codici di TM validi, e di conseguenza rappresentano la TM banale che non compie alcuna mossa, le prime righe della tabella sono in effetti formate interamente da 0.

$j \longrightarrow$

$i \downarrow$	1	2	3	4	...
1	0	1	1	0	...
2	1	1	0	0	...
3	0	0	1	1	...
4	0	1	0	1	...
.	.	.	.	.	
.	.	.	.	.	
.	.	.	.	.	

Diagonale

Figura 9.1 La tabella che rappresenta l'accettazione di stringhe da parte delle macchine di Turing.

Teorema 9.2 L_d non è un linguaggio ricorsivamente enumerabile. In altre parole non esiste alcuna macchina di Turing che accetta L_d.

DIMOSTRAZIONE Supponiamo $L_d = L(M)$ per una TM M. Poiché L_d è un linguaggio sull'alfabeto $\{0, 1\}$, M rientra nell'elenco di macchine di Turing che abbiamo costruito, dato che questo elenco include tutte le TM con alfabeto di input $\{0, 1\}$. Di conseguenza esiste almeno un codice per M, poniamo i; ossia $M = M_i$.

Chiediamoci ora se w_i è in L_d.

- Se w_i è in L_d, allora M_i accetta w_i. Pertanto, per la definizione di L_d, w_i non è in L_d, poiché L_d contiene solo le stringhe w_j tali che M_j *non* accetta w_j.

- Analogamente, se w_i non è in L_d, allora M_i non accetta w_i. Di conseguenza, per la definizione di L_d, w_i *è* in L_d.

Poiché w_i non può essere né non essere in L_d, abbiamo una contraddizione con l'ipotesi che M esista. In altre parole L_d non è un linguaggio ricorsivamente enumerabile. □

9.1.5 Esercizi

Esercizio 9.1.1 Scrivete le stringhe seguenti.

* a) w_{37}.

b) w_{100}.

Esercizio 9.1.2 Scrivete un codice possibile per la macchina di Turing della Figura 8.9.

! **Esercizio 9.1.3** Diamo due definizioni simili alla definizione di L_d. Impiegando un ragionamento analogo alla diagonalizzazione, mostrate in ambedue i casi che il linguaggio non è accettato da una macchina di Turing. Osserviamo che non è possibile sviluppare un ragionamento basato sulla diagonale stessa, per cui bisogna trovare un'altra sequenza infinita di punti nella matrice suggerita dalla Figura 9.1.

* a) L'insieme di tutte le w_i tali che w_i non è accettata da M_{2i}.

b) L'insieme di tutte le w_i tali che w_{2i} non è accettata da M_i.

! **Esercizio 9.1.4** Abbiamo considerato solo le macchine di Turing con alfabeto di input $\{0, 1\}$. Supponiamo di voler assegnare un intero a tutte le macchine di Turing, a prescindere dall'alfabeto di input. A rigor di termini questo non è possibile perché, mentre i nomi degli stati e dei simboli di nastro che non sono simboli di input sono arbitrari, i simboli di input sono significativi. Per esempio i linguaggi $\{0^n 1^n \mid n \geq 1\}$ e $\{a^n b^n \mid n \geq 1\}$, per quanto in qualche modo simili, *non* sono lo stesso linguaggio e sono accettati da TM diverse. Supponiamo tuttavia di avere un insieme infinito di simboli, $\{a_1, a_2, \ldots\}$ da cui sono selezionati tutti gli alfabeti. Mostrate come si potrebbe assegnare un intero a tutte le TM dotate di un insieme finito di tali simboli come alfabeto di input.

9.2 Un problema indecidibile ma ricorsivamente enumerabile

Abbiamo studiato un problema, il linguaggio di diagonalizzazione L_d, che non viene accettato da nessuna macchina di Turing. Il prossimo obiettivo consiste nel precisare la struttura dei linguaggi ricorsivamente enumerabili (RE), ossia quelli accettati dalle TM, suddividendoli in due classi. La prima, che corrisponde a ciò che consideriamo normalmente un algoritmo, dispone di TM che non solo riconoscono un linguaggio, ma segnalano anche quando una stringa di input non appartiene al linguaggio. Una macchina di Turing di questo tipo finisce sempre per arrestarsi, che raggiunga o no uno stato accettante.

La seconda classe di linguaggi è costituita dai linguaggi RE che non sono accettati da alcuna macchina di Turing con la garanzia di arrestarsi. Questi linguaggi sono accettati in un modo "scomodo": se l'input è nel linguaggio, prima o poi lo veniamo a sapere; in caso contrario la macchina di Turing può girare per sempre, e non possiamo decidere che l'input non verrà accettato. Come vedremo, un esempio di questo tipo di linguaggio è l'insieme di coppie codificate (M, w) tali che la TM M accetta l'input w.

9.2.1 Linguaggi ricorsivi

Diremo *ricorsivo* un linguaggio L se $L = L(M)$ per una macchina di Turing M tale che:

1. se w è in L, allora M accetta (e dunque si arresta)
2. se w non è in L, allora M si arresta pur non entrando in uno stato accettante.

Una TM di questo tipo corrisponde alla nostra nozione informale di "algoritmo": una sequenza ben definita di passi che termina sempre e produce una risposta. Se consideriamo il linguaggio L come un "problema", come succederà spesso, allora il problema L è detto ***decidibile*** se si tratta di un linguaggio ricorsivo, e ***indecidibile*** se non si tratta di un linguaggio ricorsivo.

L'esistenza o non esistenza di un algoritmo per risolvere un problema è sovente più importante dell'esistenza di una TM che risolve il problema. Come detto, una macchina di Turing senza garanzia di arresto non dà sufficienti informazioni per concludere che una stringa non si trova nel linguaggio. In un certo senso, quindi, queste TM non risolvono il problema. Di conseguenza la distinzione di problemi e linguaggi in due classi, ossia i decidibili, che vengono risolti mediante un algoritmo, e gli indecidibili, è spesso più importante della divisione tra linguaggi ricorsivamente enumerabili (quelli per cui c'è una TM) e linguaggi non ricorsivamente enumerabili (che non hanno alcuna TM). La Figura 9.2 illustra la relazione fra tre classi di linguaggi:

1. linguaggi ricorsivi
2. linguaggi ricorsivamente enumerabili ma non ricorsivi
3. linguaggi non ricorsivamente enumerabili (*non RE*).

Abbiamo collocato al suo posto il linguaggio non RE L_d e abbiamo inoltre indicato il linguaggio L_u, o "linguaggio universale", di cui dimostreremo fra breve la non ricorsività, sebbene si tratti di un RE.

9.2.2 Complementi di linguaggi ricorsivi e RE

Un modo efficace per dimostrare che un linguaggio appartiene al secondo cerchio della Figura 9.2 (vale a dire che è RE ma non ricorsivo) è lo studio del suo complemento. Mostreremo che i linguaggi ricorsivi sono chiusi rispetto all'operazione di complementazione. Perciò se un linguaggio L è RE, ma il suo complemento $\overline{L}$ non lo è, L non può essere ricorsivo. Infatti se L fosse ricorsivo, anche $\overline{L}$ sarebbe ricorsivo e dunque RE. Dimostriamo ora quest'importante proprietà di chiusura.

Teorema 9.3 Se L è un linguaggio ricorsivo, lo è anche $\overline{L}$.

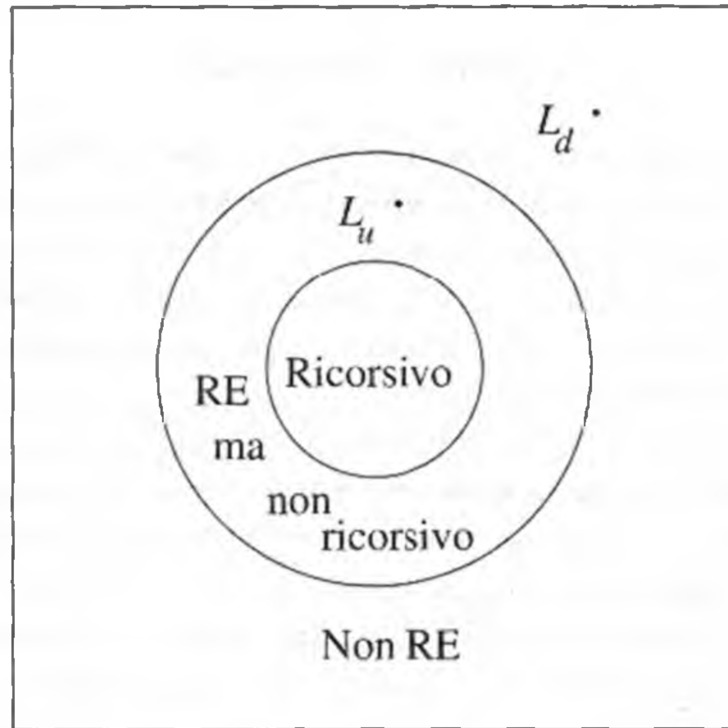


Figura 9.2 Relazione tra linguaggi ricorsivi, RE e non RE.

DIMOSTRAZIONE Sia $L = L(M)$ per una TM M che si arresta sempre. Costruiamo una TM $\overline{M}$ tale che $\overline{L} = L(\overline{M})$ secondo lo schema illustrato nella Figura 9.3. Come si vede, $\overline{M}$ si comporta esattamente come M. Per formare $\overline{M}$ modifichiamo M nel modo seguente.

1. Gli stati accettanti di M diventano stati non accettanti di $\overline{M}$ senza transizioni; in questi stati $\overline{M}$ si arresta senza accettare.

2. $\overline{M}$ ha un nuovo stato accettante r; non esiste alcuna transizione uscente da r.

3. Per ogni combinazione di uno stato non accettante e di un simbolo di nastro di M tale che M non abbia alcuna transizione (cioè M si arresta senza accettare), aggiungiamo una transizione verso lo stato accettante r.

Poiché è garantito che M si arresta, lo stesso vale per $\overline{M}$. Inoltre $\overline{M}$ accetta proprio le stringhe che M non accetta. Di conseguenza $\overline{M}$ accetta $\overline{L}$. □

Un'altra importante proprietà del complemento di un linguaggio restringe ulteriormente le possibili collocazioni di un linguaggio e del suo complemento nel diagramma della Figura 9.2. Questa proprietà è enunciata nel prossimo teorema.

Teorema 9.4 Se un linguaggio L e il suo complemento sono RE, allora L è ricorsivo. Osserviamo che in questo caso, per il Teorema 9.3, anche $\overline{L}$ è ricorsivo.

Perché "ricorsivo"

Oggi le funzioni ricorsive sono familiari ai programmatori, eppure non sembrano avere nulla a che vedere con le macchine di Turing che si arrestano sempre. Per di più il concetto opposto (non ricorsivo o indecidibile) si riferisce a linguaggi che non possono essere riconosciuti da nessun algoritmo, anche se siamo abituati a pensare a "non ricorsivo" come riferito a computazioni così semplici da non richiedere chiamate di funzioni ricorsive.

Il termine "ricorsivo" come sinonimo di "decidibile" risale alla matematica precedente l'era dei computer. Allora era prassi servirsi di formalismi basati sulla ricorsione (e non su iterazione o cicli) a sostegno del concetto di computazione. Queste notazioni, di cui non ci occuperemo qui, sono affini ai linguaggi di programmazione funzionali come LISP o ML. In quel senso, dire che un problema è "ricorsivo" ha la connotazione positiva di "sufficientemente semplice da poterlo risolvere con una funzione ricorsiva che termina sempre". Questo è proprio il significato odierno del termine in relazione alle macchine di Turing.

Il termine "ricorsivamente enumerabile" fa riferimento allo stesso campo concettuale. Una funzione può elencare tutti gli elementi di un linguaggio in un certo ordine; in altre parole li può "enumerare". I linguaggi i cui elementi possono essere elencati in un certo ordine sono gli stessi che sono accettati da una TM, anche se la TM può girare per sempre su input che non accetta.

DIMOSTRAZIONE La dimostrazione è illustrata nella Figura 9.4. Siano $L = L(M_1)$ e $\overline{L} = L(M_2)$. M_1 ed M_2 sono simulate in parallelo da una TM M. Possiamo trasformare M in una TM a due nastri e poi convertirla in una TM a nastro singolo per facilitare la simulazione. Un nastro di M simula quello di M_1, l'altro simula il nastro di M_2. Gli stati di M_1 ed M_2 sono componenti dello stato di M.

Se l'input w di M è in L, M_1 accetta in un tempo finito, quindi M accetta e si arresta. Se w non è in L, allora è in $\overline{L}$, dunque M_2 prima o poi accetta. A quel punto M si arresta senza accettare. Di conseguenza M si arresta su tutti gli input, ed $L(M)$ è esattamente L. Poiché M si arresta sempre e $L(M) = L$, concludiamo che L è ricorsivo. □

Possiamo riassumere i Teoremi 9.3 e 9.4 come segue. Delle nove possibilità di collocare un linguaggio L e il suo complemento nel diagramma della Figura 9.2, solo quattro sono consentite.

1. Sia L sia $\overline{L}$ sono ricorsivi, cioè si trovano entrambi nel cerchio interno.

2. Né L né $\overline{L}$ sono RE, cioè sono entrambi nel cerchio esterno.

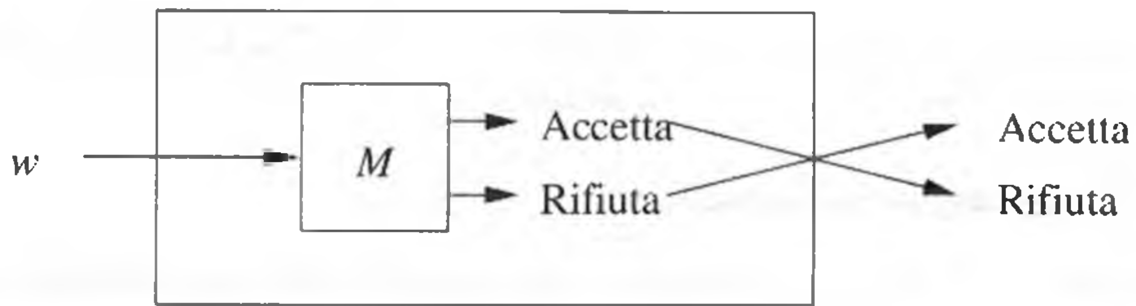


Figura 9.3 Costruzione di una TM che accetta il complemento di un linguaggio ricorsivo.

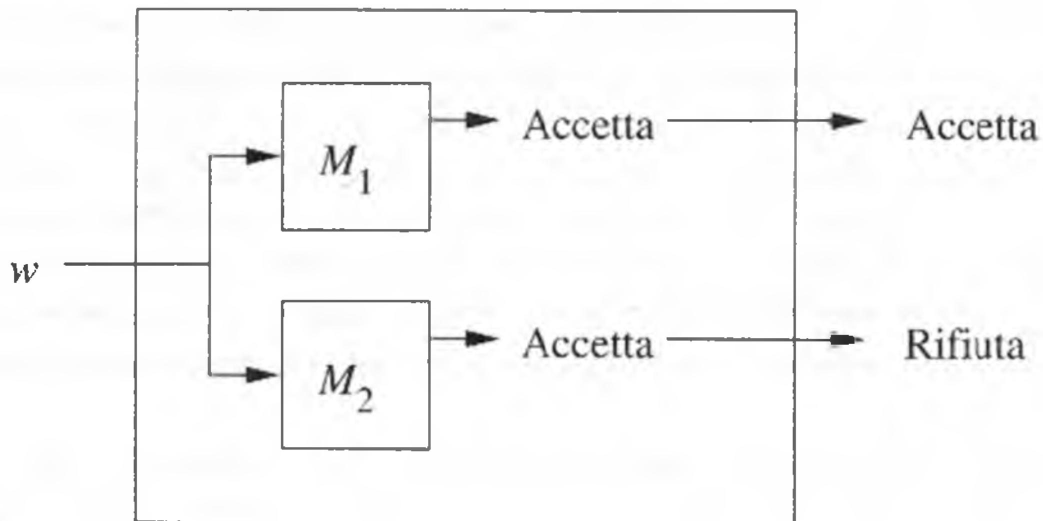


Figura 9.4 Simulazione di due TM che accettano un linguaggio e il suo complemento.

3. L è RE ma non ricorsivo, ed $\overline{L}$ non è RE; uno si trova nel cerchio intermedio, l'altro nel cerchio esterno.

4. $\overline{L}$ è RE ma non ricorsivo, ed L non è RE; il caso è analogo al punto (3), ma L ed $\overline{L}$ sono invertiti.

A dimostrazione di quanto detto, il Teorema 9.3 esclude la possibilità che un linguaggio (L o $\overline{L}$) sia ricorsivo e l'altro si trovi in una delle altre due classi. Il Teorema 9.4 esclude la possibilità che entrambi siano RE ma non ricorsivi.

Esempio 9.5 Consideriamo a titolo di esempio il linguaggio L_d, che non è RE. Perciò $\overline{L_d}$ non può essere ricorsivo. È però possibile che $\overline{L_d}$ sia non RE, o RE ma non ricorsivo. In realtà $\overline{L_d}$ è RE ma non ricorsivo.

$\overline{L_d}$ è l'insieme delle stringhe w_i tali che M_i accetta w_i. Si tratta di un linguaggio simile al linguaggio universale L_u formato da tutte le coppie (M, w) tali che M accetta w.

Questo linguaggio, come mostreremo nel Paragrafo 9.2.3, è RE. Lo stesso ragionamento dimostra che $\overline{L_d}$ è RE. □

9.2.3 Il linguaggio universale

Nel Paragrafo 8.6.2 abbiamo già discusso in termini informali come utilizzare una macchina di Turing per simulare un computer caricato con un programma arbitrario. Una singola TM può quindi essere usata come un "computer a programma memorizzato", che legge un programma e i dati relativi da uno o più nastri su cui risiede l'input. In questo paragrafo riproponiamo l'idea al più alto grado di formalità consentito dal considerare una macchina di Turing come rappresentazione di un programma memorizzato.

Definiamo L_u, il *linguaggio universale*, come l'insieme delle stringhe binarie che codificano, nella notazione del Paragrafo 9.1.2, una coppia (M, w), dove M è una TM con alfabeto di input binario e w è una stringa in $(\mathbf{0} + \mathbf{1})^*$ tale che w sia in $L(M)$. In altre parole L_u è l'insieme delle stringhe che rappresentano una TM e un input da essa accettato. Mostreremo che esiste una TM U, detta *macchina di Turing universale*, tale che $L_u = L(U)$. Poiché l'input di U è una stringa binaria, U è in effetti una M_j che possiamo trovare nell'elenco delle macchine di Turing con input binario, sviluppato nel Paragrafo 9.1.2.

Per semplicità descriviamo U come macchina di Turing multinastro, nello spirito della Figura 8.22. Nel caso di U le transizioni di M sono memorizzate inizialmente sul primo nastro insieme con la stringa w. Un secondo nastro sarà usato per contenere il nastro simulato di M, ricorrendo allo stesso formato utilizzato per il codice di M. In altre parole il simbolo di nastro X_i di M sarà rappresentato da 0^i e i simboli di nastro saranno separati da singoli 1. Il terzo nastro di U contiene lo stato di M, con lo stato q_i rappresentato da i 0. U è raffigurata nella Figura 9.5.

Descriviamo come opera U.

1. Esamina l'input per assicurarsi che il codice di M sia un codice legittimo per una TM. Se non è così, U si arresta senza accettare. Poiché si presuppone che i codici non validi rappresentino la TM senza mosse, che non accetta alcun input, si tratta di un'azione corretta.

2. Prepara il secondo nastro con l'input w nella sua forma codificata. Sul secondo nastro scrive 10 per ogni 0 e 100 per ogni 1 di w. I blank sul nastro simulato di M, che sono rappresentati da 1000, non compariranno sul nastro; tutte le celle oltre quelle usate per w conterranno il blank di U. U sa comunque che, se dovesse cercare un simbolo simulato di M e trovare il proprio blank, deve sostituire quest'ultimo con la sequenza 1000 per simulare il blank di M.

3. Scrive 0, lo stato iniziale di M, sul terzo nastro e muove la testina del secondo nastro verso la prima cella simulata.

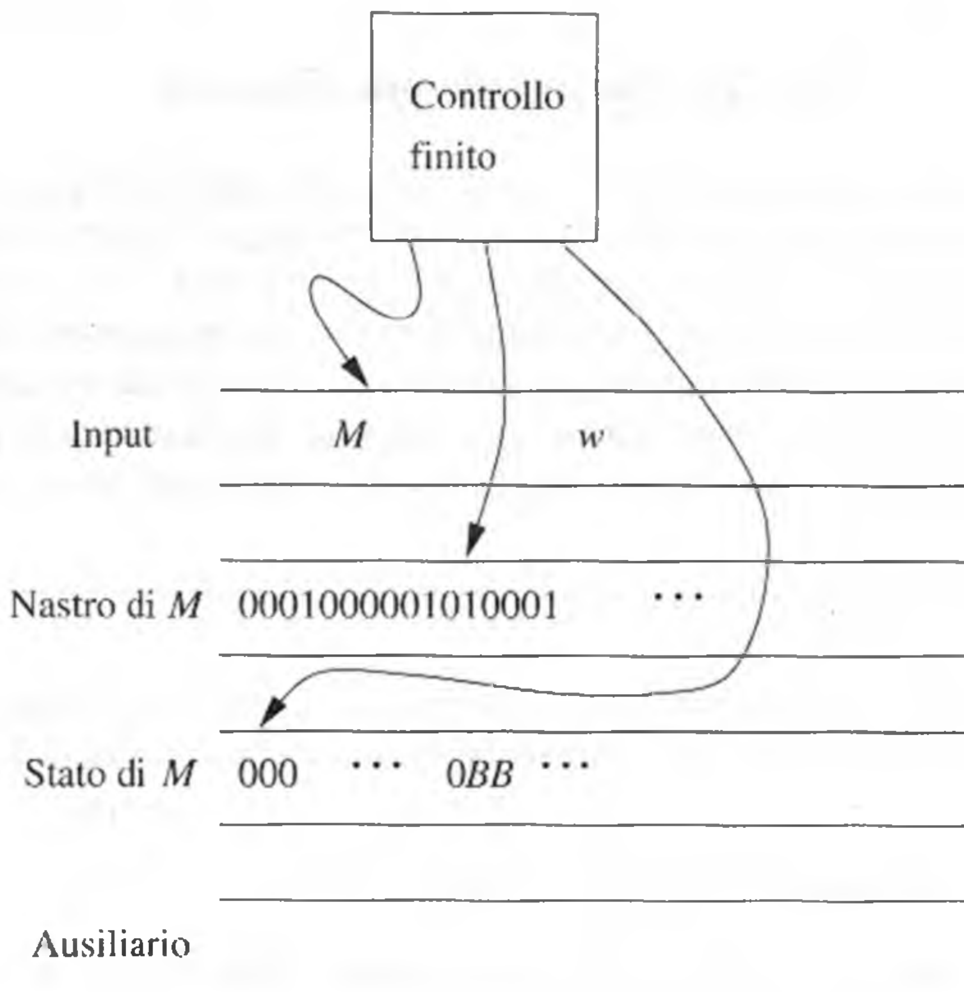


Figura 9.5 Struttura di una macchina di Turing universale.

4. Per simulare una mossa di M, U percorre il primo nastro alla ricerca di una transizione $0^i10^j10^k10^l10^m$ tale che 0^i sia lo stato sul nastro 3 e 0^j sia il simbolo di nastro di M collocato sul nastro 2 a partire dalla posizione guardata da U. Questa è la transizione che M farebbe come passo successivo. U deve fare tre cose.

   (a) Trasformare il contenuto del nastro 3 in 0^k, ossia simulare il cambiamento di stato di M. A tal fine U trasforma prima in blank tutti gli 0 sul nastro 3, e poi copia 0^k dal nastro 1 al nastro 3.

   (b) Sostituire 0^j sul nastro 2 con 0^l, cioè trasformare il simbolo di nastro di M. Se c'è bisogno di maggiore o minore spazio, cioè $i \neq l$, U si serve del nastro ausiliario e della tecnica illustrata nel Paragrafo 8.6.2.

   (c) Muove la testina del nastro 2 verso la posizione del successivo 1 a sinistra o a destra, a seconda delle due opzioni: $m = 1$ (mossa verso sinistra) o $m = 2$ (mossa verso destra). Di conseguenza U simula il movimento di M verso sinistra o destra.

Una TM universale più efficiente

Una simulazione efficiente di M da parte di U, che non imponga di "slittare" simboli sul nastro, richiede di determinare in primo luogo il numero di simboli di nastro usati da M. Se il numero di simboli è compreso tra $2^{k-1}+1$ e 2^k, U può usare un codice binario a k-bit per rappresentarli univocamente. Una cella del nastro di M può allora essere simulata da k celle di U. Per facilitare ulteriormente le cose, U può riscrivere le transizioni di M applicando il codice binario a lunghezza fissa anziché il codice unario a lunghezza variabile che abbiamo introdotto.

5. Se M non ha transizioni per lo stato simulato e il simbolo di nastro, non ci sarà alcuna transizione in (4). M si arresta nella configurazione simulata e U deve fare altrettanto.

6. Se M entra nel suo stato accettante, allora U accetta.

In questo modo U simula M su w. U accetta la coppia codificata (M, w) se e solo se M accetta w.

9.2.4 Indecidibilità del linguaggio universale

Abbiamo individuato un problema che è RE ma non ricorsivo: il linguaggio L_u. Per molti aspetti sapere che L_u è indecidibile (non è un linguaggio ricorsivo) vale più della precedente scoperta che L_d non è RE. Il motivo è che possiamo ridurre L_u a un altro problema P per dimostrare che non esiste alcun algoritmo in grado di risolvere P, a prescindere dal fatto che P sia o no RE. La riduzione di L_d a P è però possibile solo se P non è RE, quindi non possiamo usare L_d per mostrare l'indecidibilità di quei problemi che sono RE ma non ricorsivi. D'altro canto, se vogliamo dimostrare che un problema non è RE, possiamo usare solo L_d; L_u non serve, dato che *è* RE.

Teorema 9.6 L_u è RE ma non ricorsivo.

DIMOSTRAZIONE Nel Paragrafo 9.2.3 abbiamo dimostrato che L_u è RE. Supponiamo che L_u sia ricorsivo. Per il Teorema 9.3 anche $\overline{L_u}$, il complemento di L_u, è ricorsivo. Se abbiamo una TM M che accetta $\overline{L_u}$, possiamo costruire una TM che accetta L_d (grazie al metodo illustrato sopra). Poiché sappiamo già che L_d non è RE, ne ricaviamo una contraddizione con l'ipotesi che L_u sia ricorsivo.

Supponiamo $L(M) = \overline{L_u}$. Come suggerito dalla Figura 9.6 possiamo modificare la TM M in una TM M' che accetta L_d.

Il problema dell'arresto

A volte si parla del *problema dell'arresto* per le macchine di Turing come di un problema simile a L_u, cioè RE ma non ricorsivo. In effetti la macchina di Turing originale di A. M. Turing accetta per terminazione, non per stato finale. Per una TM M possiamo definire $H(M)$ come l'insieme dei w tali che M si arresta quando riceve w in input, senza tener conto se M lo accetta o no. In quel caso il *problema dell'arresto* è l'insieme delle coppie (M, w) tali che w è in $H(M)$. Questo è un altro esempio di problema/linguaggio RE ma non ricorsivo.

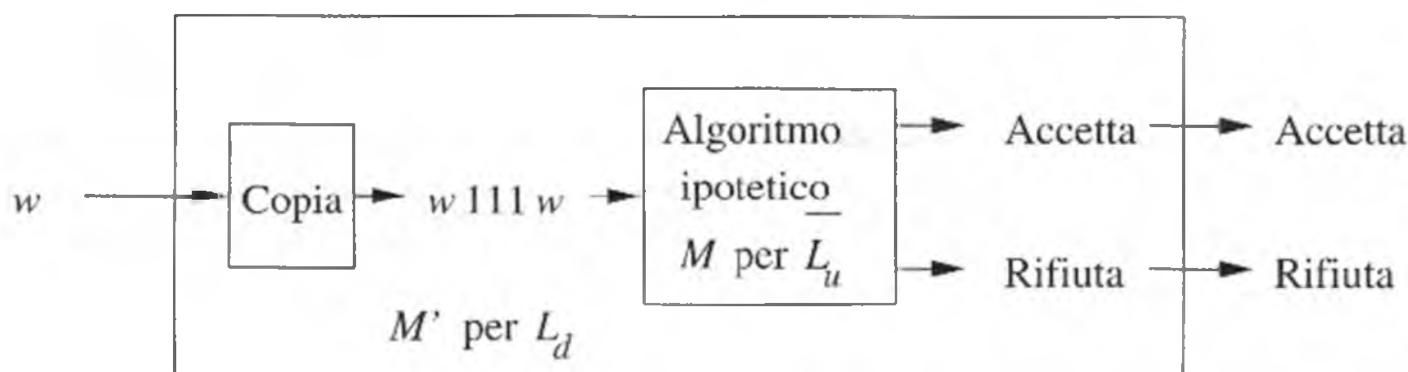


Figura 9.6 Riduzione di L_d a $\overline{L_u}$.

1. Data una stringa w in input, M' la trasforma in $w111w$. Per esercizio si può scrivere il programma di una TM che compie questo passo su un singolo nastro. Il ragionamento a sostegno della fattibilità del programma consiste nell'uso di un secondo nastro per copiare w e nella conversione della TM a due nastri in una a nastro singolo.

2. M' simula M sul nuovo input. Se nella nostra numerazione w è w_i, allora M' determina se M_i accetta w_i. Poiché M accetta $\overline{L_u}$, accetterà se e solo se M_i non accetta w_i; quindi w_i è in L_d.

Di conseguenza M' accetta w se e solo se w è in L_d. Poiché sappiamo che per il Teorema 9.2 M' non può esistere, concludiamo che L_u non è ricorsivo. □

9.2.5 Esercizi

Esercizio 9.2.1 Mostrate che il problema dell'arresto, cioè l'insieme delle coppie (M, w) tali che M si arresta (con o senza accettazione) quando riceve l'input w, è RE ma non

ricorsivo. (Rileggete il riquadro "Il problema dell'arresto" nel Paragrafo 9.2.4.)

Esercizio 9.2.2 Nel riquadro "Perché ricorsivo" del Paragrafo 9.2.1 abbiamo ricordato la nozione di "funzione ricorsiva", in competizione con la macchina di Turing come modello di "calcolabile". In quest'esercizio esploriamo un esempio di notazione per le funzioni ricorsive. Una *funzione ricorsiva* è una funzione F definita da un insieme finito di regole. Ogni regola specifica il valore della funzione F per determinati argomenti; la specifica può servirsi di variabili, costanti intere non negative, la funzione successore (somma uno), la funzione F stessa ed espressioni costruite per composizione di funzioni. Per esempio la *funzione di Ackermann* viene definita dalle regole seguenti:

1. $A(0, y) = 1$ per ogni $y \geq 0$
2. $A(1, 0) = 2$
3. $A(x, 0) = x + 2$ per $x \geq 2$
4. $A(x + 1, y + 1) = A\big(A(x, y + 1), y\big)$ per ogni $x \geq 0$ e $y \geq 0$.

* a) Valutate $A(2, 1)$.

! b) Quale funzione di x è $A(x, 2)$?

! c) Valutate $A(4, 3)$.

Esercizio 9.2.3 Descrivete in termini informali macchine di Turing multinastro che *enumerano* i seguenti insiemi di interi, nel senso che, a partire da nastri vuoti, stampano su uno di essi la stringa $10^{i_1}10^{i_2}1\cdots$ per rappresentare l'insieme $\{i_1, i_2, \ldots\}$.

* a) L'insieme di tutti quadrati perfetti $\{1, 4, 9, \ldots\}$.

b) L'insieme di tutti i numeri primi $\{2, 3, 5, 7, 11, \ldots\}$.

!! c) L'insieme di tutti gli i tali che M_i accetta w_i. *Suggerimento*: non è possibile generare questi i nell'ordine numerico perché questo linguaggio, cioè $\overline{L_d}$, è RE ma non ricorsivo. I linguaggi "RE ma non ricorsivi" si possono infatti definire come quelli enumerabili, ma non nell'ordine numerico. Per poterli enumerare dobbiamo simulare tutte le M_i su w_i. Non possiamo però farne girare una per un tempo indefinito, altrimenti non potremmo simulare nessuna M_j con $j \neq i$ se M_i non si arresta su w_i. Dobbiamo quindi istituire dei turni e fare in modo che nel turno k si provino solo un numero finito di M_i, e solo per un numero finito di passi. Possiamo così completare ogni turno in un tempo finito. Se, per ogni TM M_i e per ogni numero di passi s, c'è un turno in cui M_i viene simulata per almeno s passi, prima o poi scopriremo ogni M_i che accetta w_i, e i entrerà nell'enumerazione.

* **Esercizio 9.2.4** Sia $L_1, L_2, \ldots, L_k$ una collezione di linguaggi sull'alfabeto Σ tali che:

1. per ogni $i \neq j$, $L_i \cap L_j = \emptyset$, ossia nessuna stringa appartiene a due linguaggi
2. $L_1 \cup L_2 \cup \cdots \cup L_k = \Sigma^*$, cioè ogni stringa si trova in uno dei linguaggi
3. ogni L_i, per $i = 1, 2, \ldots, k$, è ricorsivamente enumerabile.

Dimostrate che ognuno dei linguaggi dev'essere ricorsivo.

*! **Esercizio 9.2.5** Sia L ricorsivamente enumerabile e sia $\overline{L}$ non RE. Considerate il linguaggio

$$L' = \{0w \mid w \text{ è in } L\} \cup \{1w \mid w \text{ non è in } L\}$$

Potete dire con certezza se L' o il suo complemento sono ricorsivi, RE, o non RE? Giustificate la risposta.

! **Esercizio 9.2.6** Non abbiamo discusso le proprietà di chiusura dei linguaggi ricorsivi, tranne che per la complementazione nel Paragrafo 9.2.2. Verificate se i linguaggi ricorsivi e i linguaggi RE sono chiusi rispetto alle seguenti operazioni. Per dimostrare la chiusura potete servirvi di costruzioni informali, purché chiare.

* a) Unione.

b) Intersezione.

c) Concatenazione.

d) Chiusura di Kleene (star).

* e) Omomorfismo.

f) Omomorfismo inverso.

9.3 Problemi indecidibili relativi alle macchine di Turing

Ci serviamo ora dei linguaggi L_u ed L_d, di cui conosciamo lo *status* per quanto riguarda la decidibilità e l'enumerabilità ricorsiva, per presentare altri linguaggi indecidibili o non RE. In tutte le dimostrazioni applicheremo la tecnica della riduzione. I primi problemi indecidibili di cui parliamo riguardano le macchine di Turing. La trattazione culmina nella dimostrazione del "teorema di Rice", secondo il quale qualsiasi proprietà non banale delle macchine di Turing che dipenda solo dal linguaggio accettato dalla TM dev'essere indecidibile. Nel Paragrafo 9.4 studieremo alcuni problemi indecidibili che non riguardano le macchine di Turing e i loro linguaggi.

9.3.1 Riduzioni

Abbiamo introdotto la nozione di riduzione nel Paragrafo 8.1.3. In generale, se abbiamo un algoritmo per convertire le istanze di un problema P_1 in istanze di un problema P_2 che hanno la stessa risposta, allora diciamo che P_1 *si riduce a* P_2. Possiamo avvalerci di questa dimostrazione per provare che P_2 è "difficile" almeno quanto P_1. Di conseguenza, se P_1 non è ricorsivo, allora P_2 non può essere ricorsivo. Se P_1 è non RE, allora P_2 non può essere RE. Come abbiamo detto nel Paragrafo 8.1.3, bisogna sempre ridurre un problema difficile e noto a uno di cui si vuole dimostrare che è almeno altrettanto difficile, mai il contrario.

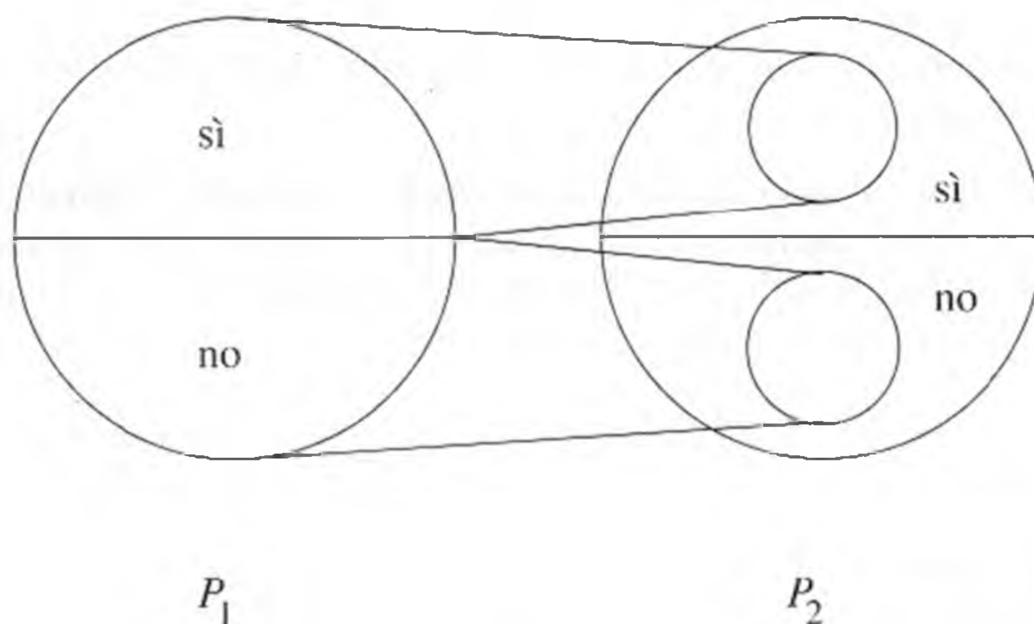


Figura 9.7 Le riduzioni trasformano istanze positive in positive e istanze negative in negative.

Come indicato nella Figura 9.7 una riduzione deve trasformare qualsiasi istanza di P_1 che ha una risposta affermativa ("sì") in un'istanza di P_2 con una risposta affermativa ("sì"), e ogni istanza di P_1 con una risposta negativa ("no") in un'istanza di P_2 con una risposta negativa ("no"). Non è essenziale che ogni istanza di P_2 sia l'immagine di una o più istanze di P_1: di fatto è normale che solo una piccola frazione di P_2 sia l'immagine della riduzione.

In termini formali una riduzione da P_1 a P_2 è una macchina di Turing che riceve un'istanza di P_1 scritta sul nastro e si arresta con un'istanza di P_2 sul nastro. Nella pratica descriveremo generalmente le riduzioni come se fossero programmi per computer che ricevono in input un'istanza di P_1 e producono come output un'istanza di P_2. L'equivalenza delle macchine di Turing e dei programmi per computer permette di descrivere

la riduzione in entrambi i modi. L'importanza delle riduzioni è sottolineata dal seguente teorema, di cui vedremo numerose applicazioni.

Teorema 9.7 Se esiste una riduzione da P_1 a P_2, allora:

a) se P_1 è indecidibile, lo è anche P_2

b) se P_1 è non RE, lo è anche P_2.

DIMOSTRAZIONE Supponiamo dapprima che P_1 sia indecidibile. Se è possibile decidere P_2, allora possiamo combinare la riduzione da P_1 a P_2 con l'algoritmo che decide P_2 per costruire un algoritmo che decide P_1. L'idea è illustrata graficamente nella Figura 8.7. Più in dettaglio, supponiamo di avere un'istanza w di P_1. Applichiamo a w l'algoritmo che la converte in un'istanza x di P_2. Applichiamo poi a x l'algoritmo che decide P_2. Se l'algoritmo dice "sì", allora x è in P_2. Poiché abbiamo ridotto P_1 a P_2, sappiamo che la risposta a w per P_1 è "sì", ossia w è in P_1. Analogamente, se x non è in P_2, allora w non è in P_1, e qualunque risposta sia data alla domanda "x è in P_2?" è anche la risposta corretta a "w è in P_1?".

Abbiamo così contraddetto l'ipotesi che P_1 sia indecidibile. Concludiamo che se P_1 è indecidibile, allora anche P_2 è indecidibile.

Consideriamo ora la parte (b). Supponiamo che P_1 sia non RE e P_2 sia RE. Disponiamo di un algoritmo che riduce P_1 a P_2, ma abbiamo solo una procedura per riconoscere P_2; in altre parole esiste una TM che dice "sì" se il suo input è in P_2, ma può non arrestarsi se il suo input non è in P_2. Come per (a), tramite l'algoritmo di riduzione convertiamo un'istanza w di P_1 in un'istanza x di P_2. Applichiamo poi a x la TM per P_2. Se x è accettato, accettiamo w.

Questa procedura descrive una TM (che può anche non arrestarsi) il cui linguaggio è P_1. Se w è in P_1, allora x è in P_2, per cui questa TM accetterà w. Se w non è in P_1, allora x non è in P_2. La TM può arrestarsi o no, ma di certo non accetterà w. Poiché abbiamo presupposto che non esista nessuna TM per P_1, abbiamo dimostrato per assurdo che non esiste nessuna TM neanche per P_2; in altre parole, se P_1 è non RE, allora anche P_2 è non RE. □

9.3.2 Macchine di Turing che accettano il linguaggio vuoto

Come esempio di riduzioni che coinvolgono le macchine di Turing, studiamo due linguaggi, che chiameremo L_e ed L_{ne}, composti da stringhe binarie. Se w è una stringa binaria, allora rappresenta una TM M_i nell'enumerazione del Paragrafo 9.1.2.

Se $L(M_i) = \emptyset$, ossia M_i non accetta alcun input, allora w è in L_e. Quindi L_e è il linguaggio formato da tutte le TM codificate il cui linguaggio è vuoto. D'altro canto, se $L(M_i)$ non è il linguaggio vuoto, allora w è in L_{ne}. Dunque L_{ne} è il linguaggio di tutti i codici delle macchine di Turing che accettano almeno una stringa di input.

Nel seguito sarà opportuno considerare le stringhe come le macchine di Turing che rappresentano. Possiamo così definire i due linguaggi appena citati:

- $L_e = \{M \mid L(M) = \emptyset\}$
- $L_{ne} = \{M \mid L(M) \neq \emptyset\}$

Osserviamo che L_e ed L_{ne} sono entrambi linguaggi sull'alfabeto binario $\{0, 1\}$ e che sono l'uno il complemento dell'altro. Vedremo che L_{ne} è il "più facile" dei due linguaggi; è RE ma non ricorsivo. Da parte sua L_e è non RE.

Teorema 9.8 L_{ne} è ricorsivamente enumerabile.

DIMOSTRAZIONE Dobbiamo solo esibire una TM che accetta L_{ne}. Il modo più semplice consiste nel descrivere una TM non deterministica M, il cui schema è rappresentato nella Figura 9.8. Per il Teorema 8.11 M può essere convertita in una TM deterministica.

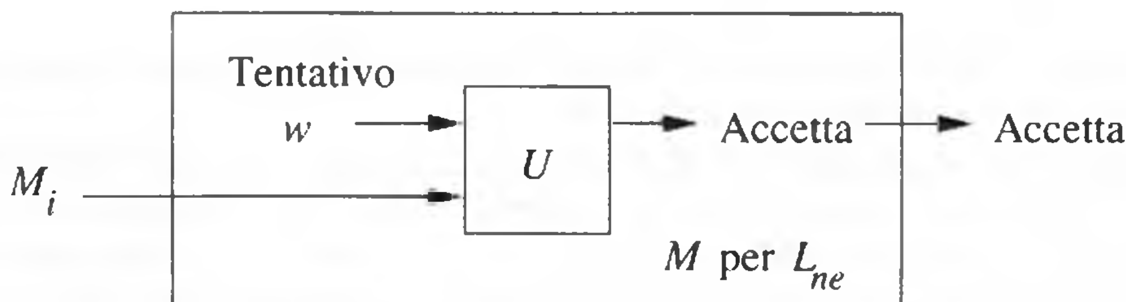


Figura 9.8 Costruzione di una NTM che accetta L_{ne}.

Descriviamo le operazioni di M.

1. M riceve in input un codice di TM M_i.
2. Impiegando la sua capacità non deterministica M "tenta" un input w che M_i potrebbe accettare.
3. M verifica se M_i accetta w. Per questa parte M può simulare la TM universale U che accetta L_u.
4. Se M_i accetta w, allora M accetta il proprio input, ossia M_i.

In questo modo, se M_i accetta anche solo una stringa, M la tenterà (tra tutte le altre, ovviamente) e accetterà M_i. Se però $L(M_i) = \emptyset$, nessun tentativo w porta all'accettazione da parte di M_i, ed M non accetta M_i. Di conseguenza $L(M) = L_{ne}$. $\square$

Il passo successivo consiste nel dimostrare che L_{ne} non è ricorsivo. A tal fine riduciamo L_u a L_{ne}. In altre parole descriviamo un algoritmo che trasforma un input (M, w) in un output M', il codice di un'altra macchina di Turing, tale che w si trova in $L(M)$ se e solo se $L(M')$ non è vuoto. Questo significa che M accetta w se e solo se M' accetta almeno una stringa. Il trucco è far sì che M' ignori il suo input e simuli invece M su input w. Se M accetta, allora M' accetta il proprio input; di conseguenza l'accettazione di w da parte di M equivale a $L(M')$ non vuoto. Se L_{ne} fosse ricorsivo, allora avremmo un algoritmo che indica se M accetta o no w: costruiamo M' e verifichiamo se $L(M') = \emptyset$.

Teorema 9.9 L_{ne} non è ricorsivo.

DIMOSTRAZIONE Seguiremo lo schema della dimostrazione data. Dobbiamo definire un algoritmo che converte un input formato da una coppia codificata in binario (M, w) in una TM M' tale che $L(M') \neq \emptyset$ se e solo se M accetta l'input w. La costruzione di M' è delineata nella Figura 9.9. Come vedremo, se M non accetta w, allora M' non accetta nessuno dei suoi input; cioè $L(M') = \emptyset$. Se però M accetta w, allora M' accetta ogni input, e di conseguenza $L(M')$ sicuramente non è $\emptyset$.

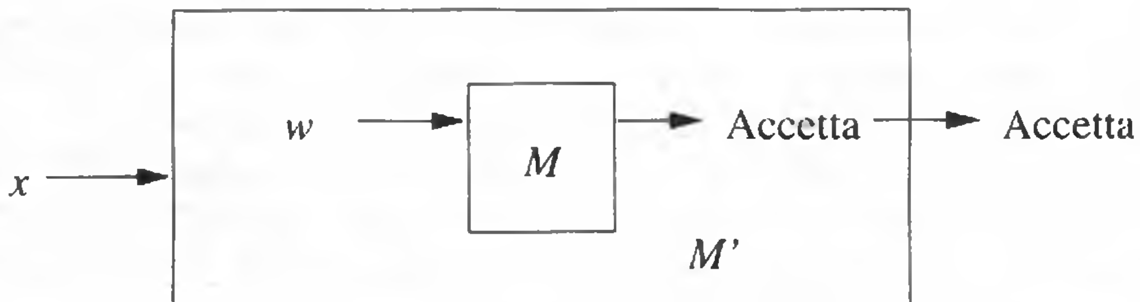


Figura 9.9 Schema della TM M' costruita da (M, w) nel Teorema 9.9. M' accetta un input arbitrario se e solo se M accetta w.

Definiamo M' affinché compia le seguenti operazioni.

1. M' ignora il proprio input x, o meglio, lo sostituisce con la stringa che rappresenta la TM M e la stringa di input w. Poiché M' è concepita per una specifica coppia (M, w), di lunghezza n, possiamo costruire M' in modo che abbia una sequenza di stati $q_0, q_1, \ldots, q_n$, dove q_0 è lo stato iniziale.

   (a) Nello stato q_i, per $i = 0, 1, \ldots, n-1$, M' scrive l'$(i+1)$-esimo bit del codice per (M, w), va nello stato q_{i+1} e si muove verso destra.

   (b) Nello stato q_n, se necessario M' si muove verso destra rimpiazzando eventuali simboli diversi dal blank (che formano la parte finale di x se tale input di M' è più lungo di n) con blank.

2. Quando M' raggiunge un blank nello stato q_n, usa un'analoga serie di stati per ricollocare la testina all'estremità sinistra del nastro.

3. Ricorrendo a stati aggiuntivi M' simula una TM universale U sul suo nastro corrente.

4. Se U accetta, allora accetta anche M'. Se U non accetta mai, allora neanche M' accetta mai.

La descrizione di M' dovrebbe convincere il lettore che è possibile definire una macchina di Turing capace di trasformare il codice di M e la stringa w nel codice di M'. In altre parole esiste un algoritmo per effettuare la riduzione di L_u a L_{ne}. Vediamo inoltre che se M accetta w, allora M' accetta qualsiasi input x presente sul nastro all'inizio. Il fatto che x sia stato ignorato è irrilevante; per la definizione di accettazione da parte di una TM sappiamo che qualunque cosa sia collocata sul nastro prima di iniziare le operazioni è quanto la TM accetta. Di conseguenza, se M accetta w, allora il codice di M' si trova in L_{ne}.

Viceversa, se M non accetta w, allora M' non accetta mai, indipendentemente dalla natura del suo input. Perciò in questo caso il codice di M' non si trova in L_{ne}. Siamo riusciti a ridurre L_u a L_{ne} per mezzo dell'algoritmo che costruisce M' da M e w; possiamo concludere che, poiché L_u non è ricorsivo, non lo è neanche L_{ne}. L'esistenza della riduzione è sufficiente a completare la dimostrazione. Per illustrare gli effetti della riduzione portiamo avanti il ragionamento di un altro passo. Se L_{ne} fosse ricorsivo, allora potremmo sviluppare un algoritmo per L_u come segue.

1. Convertiamo (M, w) nella TM M' come abbiamo visto sopra.

2. Usiamo l'algoritmo ipotetico per L_{ne} per segnalare se $L(M') = \emptyset$ o no. Nel primo caso diciamo che M non accetta w; se $L(M') \neq \emptyset$ diciamo che M accetta w.

Dal Teorema 9.6 sappiamo che non esiste un tale algoritmo per L_u. Abbiamo quindi contraddetto l'ipotesi che L_{ne} sia ricorsivo, e concludiamo che L_{ne} non è ricorsivo. □

A questo punto possiamo conoscere lo stato di L_e. Se L_e fosse RE, allora per il Teorema 9.4 sia L_e sia L_{ne} sarebbero ricorsivi. Dato che per il Teorema 9.9 L_{ne} non è ricorsivo, concludiamo che:

Teorema 9.10 L_e non è RE. □

Perché un problema e il suo complemento sono diversi

Intuitivamente sappiamo che un problema e il suo complemento sono in realtà lo stesso problema. Per risolvere l'uno possiamo usare l'algoritmo dell'altro e all'ultimo passo invertire l'esito: diciamo "sì" anziché "no", e viceversa. Se il problema e il suo complemento sono ricorsivi, l'intuizione è corretta.

Esistono però, come abbiamo visto in questo paragrafo, altri due casi. Il primo è che né il problema né il suo complemento sono RE. I due problemi risultano ancora simili in un certo senso, perché entrambi non risolvibili da alcun tipo di TM. Il secondo, più interessante, esemplificato da L_e ed L_{ne}, si ha quando l'uno è RE e l'altro non RE.

Per il linguaggio che è RE possiamo definire una TM che prende un input w e va in cerca di una ragione per cui w si trova nel linguaggio. Così, data una TM M come input, per L_{ne} avviamo la nostra TM alla ricerca di stringhe da lei stessa accettate. Non appena ne troviamo una, accettiamo M. Se M è una TM con un linguaggio vuoto, non siamo mai in grado di stabilire con certezza che non sia in L_{ne}. Tuttavia non la accettiamo mai, ed è così che la TM deve reagire correttamente.

D'altra parte, per il problema complemento L_e, che non è RE, non c'è modo di accettare tutte le sue stringhe. Supponiamo di avere una stringa M che è una TM il cui linguaggio è vuoto. Possiamo saggiare gli input della TM M ed è possibile che non ne troviamo mai uno accettato da M. Eppure non possiamo mai essere certi che non esista un input che non abbiamo ancora provato ma che potrebbe essere accettato. Di conseguenza M non può mai essere accettata, anche se dovrebbe esserlo.

9.3.3 Il teorema di Rice e le proprietà dei linguaggi RE

Il fatto che linguaggi come L_e ed L_{ne} siano indecidibili è un caso speciale di un teorema molto più generale: tutte le proprietà non banali dei linguaggi RE sono indecidibili, nel senso che è impossibile riconoscere per mezzo di una macchina di Turing le stringhe binarie che rappresentano codici di una TM il cui linguaggio soddisfa la proprietà. Un esempio di proprietà dei linguaggi RE è "il linguaggio è libero dal contesto". Come caso speciale del principio generale che tutte le proprietà non banali dei linguaggi RE sono indecidibili, il problema se una data TM accetti un linguaggio libero dal contesto è indecidibile.

Una *proprietà* dei linguaggi RE è semplicemente un insieme di linguaggi RE. Di conseguenza la proprietà di "essere libero dal contesto" è in termini formali l'insieme di

tutti i CFL. La proprietà di essere vuoto è l'insieme $\{\emptyset\}$, che consiste del solo linguaggio vuoto.

Una proprietà è *banale* se è vuota (ossia non viene soddisfatta da nessun linguaggio) o comprende tutti i linguaggi RE. Altrimenti è *non banale*.

- Osserviamo che la proprietà vuota, $\emptyset$, è diversa dalla proprietà di essere un linguaggio vuoto $\{\emptyset\}$.

Non possiamo riconoscere un insieme di linguaggi come i linguaggi stessi. La ragione è che il tipico linguaggio, essendo infinito, non può essere espresso come una stringa di lunghezza finita che possa essere l'input di una TM. Dobbiamo piuttosto riconoscere le macchine di Turing che accettano quei linguaggi; il codice della TM è finito anche se il linguaggio che accetta è infinito. Di conseguenza, se $\mathcal{P}$ è una proprietà dei linguaggi RE, il linguaggio $L_{\mathcal{P}}$ è l'insieme dei codici di macchine di Turing M_i tali che $L(M_i)$ è un linguaggio in $\mathcal{P}$. Quando parliamo di decidibilità di una proprietà $\mathcal{P}$, intendiamo la decidibilità del linguaggio $L_{\mathcal{P}}$.

Teorema 9.11 (Teorema di Rice) Ogni proprietà non banale dei linguaggi RE è indecidibile.

DIMOSTRAZIONE Sia $\mathcal{P}$ una proprietà non banale dei linguaggi RE. Per cominciare, supponiamo che $\emptyset$, il linguaggio vuoto, non sia in $\mathcal{P}$; ci occuperemo più tardi del caso opposto. Dato che $\mathcal{P}$ è non banale, deve esistere un linguaggio non vuoto L che sia in $\mathcal{P}$. Sia M_L una TM che accetta L.

Ridurremo L_u a $L_{\mathcal{P}}$, dimostrando in questo modo che $L_{\mathcal{P}}$ è indecidibile, dal momento che L_u è indecidibile. L'algoritmo per la riduzione riceve in input una coppia (M, w) e produce una TM M'. Lo schema di M' è illustrato nella Figura 9.10; $L(M')$ è $\emptyset$ se M non accetta w, e $L(M') = L$ se M accetta w.

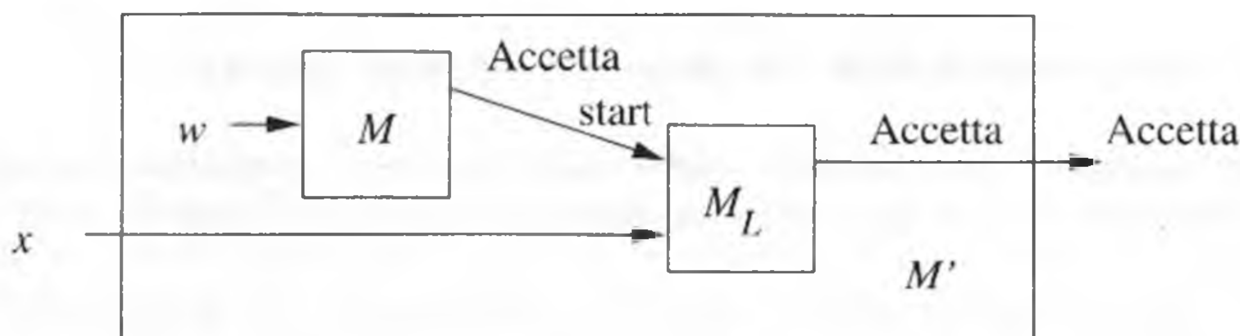


Figura 9.10 Costruzione di M' per la dimostrazione del teorema di Rice.

M' è una TM a due nastri. Un nastro è usato per simulare M su w. Ricordiamo che l'algoritmo che effettua la riduzione riceve come input M e w, e li usa nella definizione

delle transizioni di M'. Di conseguenza la simulazione di M su w è incorporata in M'; la seconda TM non deve leggere le transizioni di M sui suoi nastri.

Se necessario, l'altro nastro di M' viene usato per simulare M_L sull'input x di M'. Anche qui le transizioni di M_L sono note all'algoritmo di riduzione e possono essere incorporate nelle transizioni di M'. La TM M' è costruita per operare nel modo seguente.

1. Simula M su input w. Osserviamo che w non è l'input di M'; M' scrive invece M e w su uno dei due nastri e simula la TM universale U su tale coppia come nella dimostrazione del Teorema 9.8.

2. Se M non accetta w, allora M' non fa nient'altro. M' non accetta mai il proprio input x, per cui $L(M') = \emptyset$. Poiché assumiamo che $\emptyset$ non sia nella proprietà $\mathcal{P}$, ciò significa che il codice di M' non è in $L_{\mathcal{P}}$.

3. Se M accetta w, allora M' comincia a simulare M_L sul proprio input x. Perciò M' accetterà esattamente il linguaggio L. Poiché L è in $\mathcal{P}$, il codice di M' è in $L_{\mathcal{P}}$.

Il lettore può notare che la costruzione di M' da M e w può essere realizzata da un algoritmo. Dato che tale algoritmo trasforma (M, w) in una M' che è in $L_{\mathcal{P}}$ se e solo se (M, w) è in L_u, esso è una riduzione di L_u a $L_{\mathcal{P}}$, e dimostra che la proprietà $\mathcal{P}$ è indecidibile.

Non abbiamo ancora finito. Ci resta da considerare il caso in cui $\emptyset$ *è* in $\mathcal{P}$. Esaminiamo allora la proprietà complemento $\overline{\mathcal{P}}$, l'insieme dei linguaggi RE che non hanno la proprietà $\mathcal{P}$. Per quanto abbiamo visto sopra, $\overline{\mathcal{P}}$ è indecidibile. Ma poiché ogni TM accetta un linguaggio RE, $\overline{L_{\mathcal{P}}}$, l'insieme di (codici per) macchine di Turing che non accettano un linguaggio in $\mathcal{P}$ è uguale a $L_{\overline{\mathcal{P}}}$, l'insieme di TM che accettano un linguaggio in $\overline{\mathcal{P}}$. Supponiamo che $L_{\mathcal{P}}$ sia decidibile. Allora lo sarebbe anche $L_{\overline{\mathcal{P}}}$, perché il complemento di un linguaggio ricorsivo è ricorsivo (Teorema 9.3). $\square$

9.3.4 Problemi sulle specifiche di macchine di Turing

Per il Teorema 9.11 tutti i problemi sulle macchine di Turing che toccano solo i linguaggi accettati sono indecidibili. Alcuni di questi problemi sono interessanti di per sé. Per esempio sono indecidibili i seguenti problemi.

1. Il linguaggio accettato da una TM è vuoto (come abbiamo appreso dai Teoremi 9.9 e 9.3)?

2. Il linguaggio accettato da una TM è finito?

3. Il linguaggio accettato da una TM è un linguaggio regolare?

4. Il linguaggio accettato da una TM è un linguaggio libero dal contesto?

Il Teorema di Rice non implica che tutto ciò che riguarda le TM sia indecidibile. Questioni concernenti gli stati di una TM, e non il linguaggio da essa accettato, potrebbero per esempio essere decidibili.

Esempio 9.12 La questione se una TM abbia cinque stati è decidibile. L'algoritmo per deciderla non fa altro che esaminare il codice della TM e contare il numero degli stati che compaiono nelle transizioni.

Come ulteriore esempio, è decidibile l'esistenza di un input tale che una TM compia almeno cinque mosse. Considerando che, se una TM fa cinque mosse, le uniche celle che può guardare sono le nove intorno alla posizione iniziale della testina, l'algoritmo risulta evidente. Possiamo simulare la TM per cinque mosse su tutti i nastri (in numero finito) formati da non più di cinque simboli di input, preceduti e seguiti da blank. Se una qualsiasi di queste simulazioni non raggiunge una situazione di arresto, concludiamo che la TM compie almeno cinque mosse su un input. □

9.3.5 Esercizi

* **Esercizio 9.3.1** Mostrate che l'insieme dei codici di macchine di Turing che accettano tutti gli input palindromi (eventualmente insieme con altri input) è indecidibile.

Esercizio 9.3.2 La Big Computer Corp. ha deciso di incrementare la sua fetta di mercato in crisi fabbricando una versione high-tech della macchina di Turing, chiamata BWTM, fornita di *bells*, campanelli, e *whistles*, fischietti. Fondamentalmente la BWTM è una comune macchina di Turing, salvo il fatto che ogni stato della macchina è etichettato come "stato campanello" o come "stato fischietto". Ogni volta che la BWTM entra in un nuovo stato suona il campanello o fischia, a seconda del tipo di stato. Dimostrate che è indecidibile se una data BWTM M fischia su un dato input w.

Esercizio 9.3.3 Mostrate che il linguaggio dei codici per le TM M che, partite con il nastro bianco, prima o poi scrivono un 1 sul nastro è indecidibile.

! **Esercizio 9.3.4** Dal teorema di Rice sappiamo che nessuno dei problemi che seguono è decidibile. Sono però ricorsivamente enumerabili oppure non RE?

a) $L(M)$ contiene almeno due stringhe?

b) $L(M)$ è infinito?

c) $L(M)$ è un linguaggio libero dal contesto?

* d) $L(M) = \left(L(M)\right)^R$?

! **Esercizio 9.3.5** Sia L il linguaggio formato dalle coppie di codici di TM più un intero, (M_1, M_2, k), tali che $L(M_1) \cap L(M_2)$ contiene almeno k stringhe. Dimostrate che L è RE ma non ricorsivo.

Esercizio 9.3.6 Dimostrate che le seguenti questioni sono decidibili.

* a) L'insieme dei codici di TM M tali che, una volta partita con un nastro bianco, M scrive prima o poi un simbolo diverso dal blank sul nastro. *Suggerimento*: se M ha m stati, considerate le sue prime m transizioni.

! b) L'insieme dei codici di TM che non compiono mai una mossa verso sinistra.

! c) L'insieme delle coppie (M, w) tali che la TM M, partita con input w, non visita mai una cella di nastro più di una volta.

! **Esercizio 9.3.7** Dimostrate che i seguenti problemi non sono ricorsivamente enumerabili.

* a) L'insieme delle coppie (M, w) tali che la TM M, partita con input w, non si arresta.

b) L'insieme delle coppie (M_1, M_2) tali che $L(M_1) \cap L(M_2) = \emptyset$.

c) L'insieme delle triple (M_1, M_2, M_3) tali che $L(M_1) = L(M_2)L(M_3)$, cioè tali che il linguaggio della prima TM è la concatenazione dei linguaggi delle altre due.

!! **Esercizio 9.3.8** Dite se i seguenti insiemi sono ricorsivi, RE ma non ricorsivi, o non RE.

* a) L'insieme di tutti i codici di TM che si arrestano su ogni input.

b) L'insieme di tutti i codici di TM che non si arrestano su nessun input.

c) L'insieme di tutti i codici di TM che si arrestano su almeno un input.

* d) L'insieme di tutti i codici di TM che non si arrestano su almeno un input.

9.4 Il problema di corrispondenza di Post

In questo paragrafo cominciamo a ridurre problemi indecidibili sulle macchine di Turing a problemi indecidibili su oggetti "reali", cioè svincolati dall'astrazione della macchina di Turing. Partiamo dal cosiddetto "problema di corrispondenza di Post" (PCP, *Post's Correspondence Problem*), che è ancora un problema astratto, ma si riferisce a stringhe anziché a macchine di Turing. Intendiamo dimostrare che questo problema è indecidibile e poi sfruttare il risultato per dimostrare che altri problemi sono indecidibili per riduzione di PCP.

Dimostriamo che PCP è indecidibile riducendo L_u a PCP. Per semplificare la prova definiamo un PCP modificato e lo riduciamo al PCP originale. Quindi riduciamo L_u al PCP modificato. La catena di riduzioni è illustrata nella Figura 9.11. Poiché sappiamo che L_u è indecidibile, concludiamo che PCP è indecidibile.

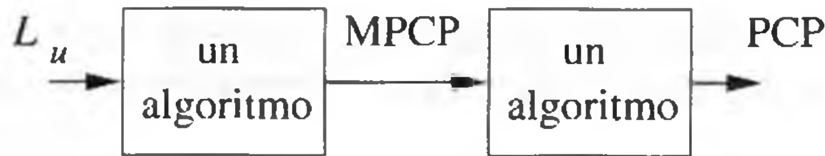


Figura 9.11 Riduzioni che dimostrano l'indecidibilità del problema di corrispondenza di Post.

9.4.1 Definizione del problema di corrispondenza di Post

Un'istanza del *problema di corrispondenza di Post* (PCP) consiste in due liste di stringhe sullo stesso alfabeto Σ; le due liste devono avere la stessa lunghezza. Generalmente parleremo delle liste A e B, e scriveremo $A = w_1, w_2, \ldots, w_k$ e $B = x_1, x_2, \ldots, x_k$, per un intero k. Per ogni i la coppia (w_i, x_i) si dice coppia *corrispondente*.

Diciamo che l'istanza di PCP *ha soluzione* se esiste una sequenza di uno o più interi $i_1, i_2, \ldots, i_m$ che, interpretati come indici per le stringhe di A e B, producono la stessa stringa. Formalmente $w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}$. In questo caso diciamo che la sequenza $i_1, i_2, \ldots, i_m$ è una *soluzione* dell'istanza di PCP. Il problema di corrispondenza di Post è definito così:

- data un'istanza di PCP, dire se ha una soluzione.

	Lista A	Lista B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Figura 9.12 Un'istanza di PCP.

Esempio 9.13 Sia $\Sigma = \{0, 1\}$, e siano A e B le liste definite come nella Figura 9.12. In questo caso PCP ha una soluzione. Per esempio, siano $m = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$ e $i_4 = 3$; la soluzione è allora la lista $2, 1, 1, 3$. Verifichiamo che sia una soluzione concatenando le stringhe corrispondenti delle due liste nell'ordine indicato: $w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3 = 101111110$. Osserviamo che la soluzione non è unica: per esempio $2, 1, 1, 3, 2, 1, 1, 3$ è un'altra soluzione. □

Il PCP come linguaggio

Per trattare la questione se un'istanza di PCP abbia una soluzione, dobbiamo esprimere il problema come linguaggio. Poiché le istanze di PCP hanno alfabeto arbitrario, il linguaggio corrispondente è di fatto una stringa su un alfabeto fissato, che codifica le istanze di PCP così come nel Paragrafo 9.1.2 abbiamo codificato macchine di Turing con insiemi arbitrari di stati e di simboli di nastro. In particolare, se l'alfabeto di un'istanza di PCP ha meno di 2^k simboli, possiamo usare codici di k bit per ogni simbolo.

Poiché le istanze di PCP hanno tutte alfabeto finito, per ciascuna possiamo fissare il valore di k. A questo punto possiamo codificare un'istanza impiegando un alfabeto di tre simboli: 0, 1 e un simbolo "virgola" per separare le stringhe. Collochiamo all'inizio del codice il numero k in binario seguito dalla virgola e dalle coppie di stringhe, separate da virgole e codificate secondo un codice a k bit.

	Lista A	Lista B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

Figura 9.13 Un'altra istanza di PCP.

Esempio 9.14 Ecco un'istanza senza soluzione. Prendiamo ancora $\Sigma = \{0, 1\}$, ma le due liste sono quelle della Figura 9.13.

Supponiamo che l'istanza di PCP della Figura 9.13 abbia la soluzione $i_1, i_2, \dots, i_m$, per un $m \geq 1$. Affermiamo che $i_1 = 1$. Infatti, se $i_1 = 2$, una stringa che comincia per $w_2 = 011$ dovrebbe coincidere con una che comincia per $x_2 = 11$. Ma ciò è impossibile perché i primi simboli delle due stringhe sono, rispettivamente, 0 e 1. Analogamente non è possibile che $i_1 = 3$ perché in tal caso una stringa che comincia per $w_3 = 101$ dovrebbe coincidere con una che comincia per $x_3 = 011$.

Se $i_1 = 1$, le due stringhe corrispondenti nelle liste A e B dovrebbero cominciare così:

$$A\text{: } 10\cdots$$
$$B\text{: } 101\cdots$$

Consideriamo che cosa dev'essere i_2.

Soluzioni parziali

Nell'Esempio 9.14 abbiamo impiegato una tecnica molto comune per analizzare istanze di PCP. Abbiamo considerato le possibili *soluzioni parziali*, cioè sequenze di indici $i_1, i_2, \ldots, i_r$ tali che una fra $w_{i_1} w_{i_2} \cdots w_{i_r}$ e $x_{i_1} x_{i_2} \cdots x_{i_r}$ sia prefisso dell'altra, anche se le due stringhe non sono uguali. Notiamo che, se una sequenza di interi è una soluzione, ogni suo prefisso è una soluzione parziale. Perciò ragionare sulle soluzioni parziali permette di dedurre proprietà delle soluzioni globali.

Notiamo d'altra parte che, essendo PCP indecidibile, non esiste un algoritmo che calcoli tutte le soluzioni parziali. Esse sono in numero infinito e, soprattutto, non c'è un limite superiore alla differenza in lunghezza delle stringhe $w_{i_1} w_{i_2} \cdots w_{i_r}$ e $x_{i_1} x_{i_2} \cdots x_{i_r}$, anche se formano una soluzione parziale estendibile a globale.

1. Non può valere $i_2 = 1$ perché nessuna stringa che comincia per $w_1 w_1 = 1010$ può coincidere con una che comincia per $x_1 x_1 = 101101$: esse discordano alla quarta posizione.

2. Non può valere neppure $i_2 = 2$ perché nessuna stringa che comincia per $w_1 w_2 = 10011$ può coincidere con una che comincia per $x_1 x_2 = 10111$: esse differiscono alla terza posizione.

3. Solo $i_2 = 3$ è possibile.

Se poniamo $i_2 = 3$, le stringhe corrispondenti, formate dalla lista di interi i_1, i_3, sono le seguenti:

$$A\text{: } 10101 \cdots$$
$$B\text{: } 101011 \cdots$$

Nulla in queste stringhe indica chiaramente che la lista $1, 3$ non si possa estendere a una soluzione. Eppure siamo in grado di provare che non è possibile farlo perché siamo nella stessa situazione di quando abbiamo scelto $i_1 = 1$. La stringa presa dalla lista B è la stessa della lista A, ma con un 1 in più alla fine. Siamo perciò obbligati a scegliere $i_3 = 3$, $i_4 = 3$, e così via, per mantenere la corrispondenza. La stringa di A non può mai raggiungere quella di B, e quindi non possiamo raggiungere una soluzione. □

9.4.2 Il PCP modificato

È facile ridurre L_u a PCP se introduciamo prima una versione intermedia di PCP, che chiamiamo *problema di corrispondenza di Post modificato*, o MPCP. In questa variante imponiamo il vincolo aggiuntivo che la prima coppia delle liste A e B dev'essere la prima anche nella soluzione. Un'istanza di MPCP è data formalmente da due liste, $A = w_1, w_2, \dots, w_k$ e $B = x_1, x_2, \dots, x_k$, mentre una soluzione è una sequenza di 0 o più interi $i_1, i_2, \dots, i_m$ tali che

$$w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$$

Notiamo che la coppia (w_1, x_1) deve trovarsi all'inizio delle due stringhe, anche se l'indice 1 non è menzionato in testa alla sequenza che forma la soluzione. Inoltre, a differenza di PCP, in cui la soluzione deve contenere almeno un intero, la sequenza vuota può essere soluzione di MPCP se $w_1 = x_1$ (un caso del genere non è però molto interessante, e non ne incontreremo nel seguito).

Esempio 9.15 Possiamo interpretare le liste della Figura 9.12 come un'istanza di MPCP, che però non ha soluzioni. Per dimostrarlo osserviamo che ogni soluzione parziale deve cominciare con l'indice 1 e le due stringhe relative devono cominciare così:

$$\begin{aligned} A&: 1 \cdots \\ B&: 111 \cdots \end{aligned}$$

L'intero successivo non può essere né 2 né 3 perché sia w_2 sia w_3 cominciano per 10 e si avrebbe una discordanza alla terza posizione. L'indice successivo deve quindi essere 1:

$$\begin{aligned} A&: 11 \cdots \\ B&: 111111 \cdots \end{aligned}$$

Possiamo proseguire così indefinitamente. Solo un altro 1 nella soluzione evita la discordanza. Tuttavia se possiamo scegliere solo l'indice 1, la stringa di B sarà sempre tre volte più lunga della stringa di A, ed esse non coincideranno mai. □

Un passo importante nella prova che PCP risulta indecidibile è la riduzione di MPCP a PCP. Dimostreremo poi che MPCP è indecidibile riducendo L_u a MPCP. Avremo allora la prova che anche PCP è indecidibile; se fosse decidibile, lo sarebbe anche MPCP, e quindi anche L_u.

Data un'istanza di MPCP con alfabeto Σ, costruiamo un'istanza di PCP. Introduciamo un nuovo simbolo, $*$, che nell'istanza di PCP si interpone fra i simboli delle stringhe dell'istanza di MPCP. Ma nelle stringhe di A gli $*$ seguono i simboli di Σ, mentre in quelle di B li precedono. C'è un'eccezione: una nuova coppia, formata a partire dalla prima

coppia dell'istanza di MPCP; questa coppia ha un $*$ in più all'inizio di w_1, e possiamo usarla per avviare la soluzione di PCP. All'istanza di PCP aggiungiamo un'ultima coppia, $(\$, *\$)$, che farà da coppia terminale nelle soluzioni di PCP che imitano soluzioni di MPCP.

Definiamo ora formalmente la costruzione. Abbiamo un'istanza di MPCP con sequenze $A = w_1, w_2, \ldots, w_k$ e $B = x_1, x_2, \ldots, x_k$. Supponiamo che $*$ e \$ non siano elementi dell'alfabeto Σ di questa istanza. Costruiamo un'istanza di PCP $C = y_0, y_1, \ldots, y_{k+1}$ e $D = z_0, z_1, \ldots, z_{k+1}$.

1. Per $i = 1, 2, \ldots, k$ sia y_i uguale a w_i, ma con un $*$ dopo ogni simbolo, e sia z_i uguale a x_i, ma con un $*$ prima di ogni simbolo.

2. $y_0 = *y_1$ e $z_0 = z_1$. La coppia di posto 0 è uguale a quella di posto 1, salvo che per il simbolo $*$ supplementare davanti alla stringa della prima lista. Notiamo che la coppia di posto 0 è l'unica, nell'istanza di PCP, in cui le due stringhe cominciano per lo stesso simbolo, così che ogni soluzione deve cominciare dall'indice 0.

3. $y_{k+1} = \$$ e $z_{k+1} = *\$$.

Esempio 9.16 Supponiamo che la Figura 9.12 sia un'istanza di MPCP. L'istanza di PCP costruita come descritto è illustrata nella Figura 9.14. □

	Lista C	Lista D
i	y_i	z_i
0	$*1*$	$*1*1*1$
1	$1*$	$*1*1*1$
2	$1*0*1*1*1*$	$*1*0$
3	$1*0*$	$*0$
4	\$	$*\$$

Figura 9.14 Costruzione di un'istanza di PCP da un'istanza di MPCP.

Teorema 9.17 MPCP si riduce a PCP.

DIMOSTRAZIONE Il nucleo della prova è la costruzione definita sopra. Supponiamo per prima cosa che $i_1, i_2, \ldots, i_m$ sia una soluzione all'istanza assegnata di MPCP con liste A e B. Sappiamo allora che $w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$. Sostituendo le w con le y e le x con le z otteniamo due stringhe quasi uguali: $y_1 y_{i_1} y_{i_2} \cdots y_{i_m}$ e $z_1 z_{i_1} z_{i_2} \cdots z_{i_m}$. Alla prima stringa manca un $*$ all'inizio; alla seconda ne manca uno alla fine. Quindi

$$*y_1 y_{i_1} y_{i_2} \cdots y_{i_m} = z_1 z_{i_1} z_{i_2} \cdots z_{i_m} *$$

Poiché $y_0 = *y_1$ e $z_0 = z_1$ possiamo rimediare alla mancanza del simbolo $*$ iniziale sostituendo il primo indice con 0. Abbiamo ora

$$y_0 y_{i_1} y_{i_2} \cdots y_{i_m} = z_0 z_{i_1} z_{i_2} \cdots z_{i_m} *$$

Per il simbolo $*$ finale possiamo accodare l'indice $k+1$. Poiché $y_{k+1} = \$$ e $z_{k+1} = *\$$ abbiamo:

$$y_0 y_{i_1} y_{i_2} \cdots y_{i_m} y_{k+1} = z_0 z_{i_1} z_{i_2} \cdots z_{i_m} z_{k+1}$$

Abbiamo così dimostrato che $0, i_1, i_2, \ldots, i_m, k+1$ è una soluzione dell'istanza di PCP.

Dobbiamo ora dimostrare che, se l'istanza del PCP derivato ha una soluzione, anche l'istanza dell'MPCP originale ha una soluzione. Osserviamo che una soluzione dell'istanza di PCP deve cominciare con l'indice 0 e finire con l'indice $k+1$, perché solo la coppia di posto 0 è formata da stringhe che cominciano per lo stesso simbolo, e solo la coppia di posto $(k+1)$ è formata da stringhe che finiscono con lo stesso simbolo. Perciò possiamo scrivere così la soluzione di PCP: $0, i_1, i_2, \ldots, i_m, k+1$.

Affermiamo che $i_1, i_2, \ldots, i_m$ è una soluzione dell'istanza di MPCP. Infatti se eliminiamo gli $*$ e il \$ finale dalla stringa $y_0 y_{i_1} y_{i_2} \cdots y_{i_m} y_{k+1}$ otteniamo $w_1 w_{i_1} w_{i_2} \cdots w_{i_m}$. Inoltre se eliminiamo gli $*$ e il \$ da $z_0 z_{i_1} z_{i_2} \cdots z_{i_m} z_{k+1}$ otteniamo $x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$. Sappiamo che

$$y_0 y_{i_1} y_{i_2} \cdots y_{i_m} y_{k+1} = z_0 z_{i_1} z_{i_2} \cdots z_{i_m} z_{k+1}$$

perciò deduciamo che

$$w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$$

Quindi da una soluzione dell'istanza di PCP ricaviamo una soluzione dell'istanza di MPCP.

Ora è chiaro che la costruzione descritta prima del teorema è un algoritmo per convertire un'istanza di MPCP con una soluzione in un'istanza di PCP con una soluzione, e un'istanza di MPCP senza soluzioni in un'istanza di PCP senza soluzioni. Esiste dunque una riduzione di MPCP a PCP, a conferma che se PCP fosse decidibile lo sarebbe anche MPCP. □

9.4.3 Dimostrazione di indecidibilità di PCP: finale

Completiamo la catena di riduzioni della Figura 9.11 riducendo L_u a MPCP. Data una coppia (M, w) dobbiamo cioè costruire un'istanza (A, B) di MPCP tale che la TM M accetta l'input w se e solo se (A, B) ha una soluzione.

Fulcro del ragionamento è il fatto che l'istanza (A, B) di MPCP simula, nelle soluzioni parziali, la computazione di M su input w. Le soluzioni parziali sono dunque formate

da prefissi della sequenza di ID di M: $\#\alpha_1\#\alpha_2\#\alpha_3\#\cdots$, dove α_1 è la ID iniziale di M con input w, e $\alpha_i \vdash \alpha_{i+1}$ per ogni i. La stringa presa dalla lista B sarà sempre di una ID più avanti rispetto a quella presa da A, salvo che M entri in uno stato accettante. In quel caso saranno disponibili le coppie con cui A può "raggiungere" B e produrre infine una soluzione. Se però M non entra in uno stato accettante, non c'è modo di servirsene, e non ci sono soluzioni.

Per semplificare la costruzione dell'istanza di MPCP ricorriamo al Teorema 8.12, secondo il quale possiamo assumere che la TM non stampa mai blank e non si sposta mai a sinistra della posizione iniziale della testina. In questo modo ogni ID della macchina di Turing è una stringa della forma $\alpha q\beta$, dove α e β sono stringhe di simboli di nastro diversi dal blank, e q è uno stato. β può essere vuoto se la testina si trova sul blank immediatamente a destra di α; evitiamo così di collocare un blank a destra dello stato. I simboli di α e β corrispondono quindi esattamente al contenuto delle celle in cui si trovava l'input, più tutte le celle verso destra già visitate dalla testina.

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ una TM che soddisfa il Teorema 8.12, e sia $w \in \Sigma^*$, una stringa di input. Costruiamo un'istanza di MPCP. Per capire i motivi della scelta delle coppie ricordiamo che la prima lista deve sempre trovarsi di una ID dietro la seconda, a meno che M accetti.

1. La prima coppia è:

Lista A	Lista B
$\#$	$\#q_0w\#$

   Questa coppia, da cui deve partire ogni soluzione, secondo le regole di MPCP, avvia la simulazione di M su input w. All'inizio B precede A di una ID.

2. Ad ambedue le liste possiamo accodare simboli di nastro e il separatore #. Le coppie

Lista A	Lista B	
X	X	per ogni X in Γ
$\#$	$\#$	

   permettono di "copiare" simboli che non contengono lo stato. La scelta di queste coppie ci consente in effetti di estendere la stringa di A in accordo con quella di B, copiando al contempo parti della ID precedente in coda alla stringa di B. Possiamo in questo modo formare la ID successiva nella sequenza di mosse di M, in coda alla stringa di B.

3. Per simulare le mosse di M disponiamo di coppie che le rispecchiano. Per ogni q in $Q - F$ (cioè q è uno stato non accettante), p in Q, e X, Y e Z in Γ abbiamo:

Lista A	Lista B	
qX	Yp	se $\delta(q, X) = (p, Y, R)$
ZqX	pZY	se $\delta(q, X) = (p, Y, L)$; Z è un simbolo di nastro arbitrario
$q\#$	$Yp\#$	se $\delta(q, B) = (p, Y, R)$
$Zq\#$	$pZY\#$	se $\delta(q, B) = (p, Y, L)$; Z è un simbolo di nastro

Come quelle del punto (2), queste coppie permettono di estendere la stringa di B con la ID successiva, e di estendere la stringa di A in accordo con quella di B. Le coppie si servono dello stato per stabilire come cambia la ID corrente e generare la successiva. I cambiamenti – nuovo stato, simbolo di nastro e spostamento della testina – si riflettono nella ID accodata alla stringa di B.

4. Se la ID in coda alla stringa di B ha uno stato accettante, dobbiamo completare la soluzione parziale. A tale scopo la estendiamo con "ID" che non sono vere ID di M, ma rappresentano quanto accadrebbe se lo stato accettante potesse consumare i simboli di nastro ai suoi due lati. Se q è uno stato accettante, per ogni X e Y, simboli di nastro, esistono coppie:

Lista A	Lista B
XqY	q
Xq	q
qY	q

5. Infine, dopo aver consumato tutti i simboli di nastro, lo stato accettante resta da solo come ultima ID nella stringa di B. Il *residuo* delle due stringhe (il suffisso della stringa di B che si deve accodare alla stringa di A per uguagliarla) è $q\#$. Per completare la soluzione usiamo l'ultima coppia:

Lista A	Lista B
$q\#\#$	$\#$

Nel seguito scriveremo "coppie della regola (1), della regola (2)", e così via, per riferirci ai cinque tipi di coppie fin qui generate.

Esempio 9.18 Convertiamo in un'istanza di MPCP la TM

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_3\})$$

dove δ è definita da:

q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
q_1	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
q_3	—	—	—

e la stringa di input è $w = 01$. Per semplificare, osserviamo che M non scrive mai blank, e quindi B non compare mai nelle ID. Possiamo perciò tralasciare le coppie relative a B. La lista completa delle coppie è riportata nella Figura 9.15, corredata della spiegazione dell'origine di ogni coppia.

Regola	Lista A	Lista B	Origine
(1)	#	$\#q_1 01\#$	
(2)	0	0	
	1	1	
	#	#	
(3)	$q_1 0$	$1q_2$	da $\delta(q_1, 0) = (q_2, 1, R)$
	$0q_1 1$	$q_2 00$	da $\delta(q_1, 1) = (q_2, 0, L)$
	$1q_1 1$	$q_2 10$	da $\delta(q_1, 1) = (q_2, 0, L)$
	$0q_1\#$	$q_2 01\#$	da $\delta(q_1, B) = (q_2, 1, L)$
	$1q_1\#$	$q_2 11\#$	da $\delta(q_1, B) = (q_2, 1, L)$
	$0q_2 0$	$q_3 00$	da $\delta(q_2, 0) = (q_3, 0, L)$
	$1q_2 0$	$q_3 10$	da $\delta(q_2, 0) = (q_3, 0, L)$
	$q_2 1$	$0q_1$	da $\delta(q_2, 1) = (q_1, 0, R)$
	$q_2\#$	$0q_2\#$	da $\delta(q_2, B) = (q_2, 0, R)$
(4)	$0q_3 0$	q_3	
	$0q_3 1$	q_3	
	$1q_3 0$	q_3	
	$1q_3 1$	q_3	
	$0q_3$	q_3	
	$1q_3$	q_3	
	$q_3 0$	q_3	
	$q_3 1$	q_3	
(5)	$q_3\#\#$	#	

Figura 9.15 Istanza di MPCP costruita dalla TM M dell'Esempio 9.18.

Osserviamo che M accetta l'input 01 tramite la sequenza di mosse

$$q_1 01 \vdash 1q_2 1 \vdash 10q_1 \vdash 1q_2 01 \vdash q_3 101$$

Consideriamo ora la sequenza di soluzioni parziali che imita questa computazione di M

e porta a una soluzione. Come per ogni soluzione di MPCP dobbiamo partire dalla prima coppia:

$$A: \#$$
$$B: \#q_1 01\#$$

C'è un solo modo di estendere la soluzione parziale: la stringa presa da A dev'essere un prefisso del residuo $q_1 01\#$. Dobbiamo quindi scegliere la coppia $(q_1 0, 1q_2)$, una delle coppie che simulano una mossa prese dalla regola (3). Ecco la soluzione parziale:

$$A: \#q_1 0$$
$$B: \#q_1 01\#1q_2$$

Possiamo estendere ulteriormente la soluzione parziale mediante le coppie della regola (2) fino ad avere lo stato nella seconda ID. La soluzione parziale diventa:

$$A: \#q_1 01\#1$$
$$B: \#q_1 01\#1q_2 1\#1$$

A questo punto possiamo usare un'altra coppia della regola (3) per simulare una mossa; la coppia appropriata è $(q_2 1, 0q_1)$. Se ne ricava la soluzione parziale:

$$A: \#q_1 01\#1q_2 1$$
$$B: \#q_1 01\#1q_2 1\#10q_1$$

Ora potremmo usare le coppie della regola (2) per "copiare" i tre simboli successivi: #, 1 e 0. Ma spingersi tanto in là sarebbe un errore perché la mossa successiva di M sposta la testina a sinistra, e lo 0 che precede lo stato serve alla successiva coppia della regola (3). Perciò "copiamo" solo i due simboli successivi, ottenendo una nuova soluzione parziale:

$$A: \#q_1 01\#1q_2 1\#1$$
$$B: \#q_1 01\#1q_2 1\#10q_1\#1$$

La coppia della regola (3) da usare qui è $(0q_1\#, q_2 01\#)$, che dà la soluzione parziale

$$A: \#q_1 01\#1q_2 1\#10q_1\#$$
$$B: \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#$$

Possiamo ora usare un'altra coppia della regola (3), $(1q_2 0, q_3 10)$, che porta all'accettazione:

$$A: \#q_1 01\#1q_2 1\#10q_1\#1q_2 0$$
$$B: \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 10$$

A questo punto ci serviamo di coppie della regola (4) per eliminare dalla ID tutti i simboli tranne q_3. Ci servono anche coppie della regola (2) per copiare simboli. La soluzione parziale prosegue così:

$$A: \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#$$
$$B: \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#$$

Con il solo q_3 nella ID possiamo usare la coppia $(q_3\#\#, \#)$ dalla regola (5) per completare la soluzione:

$$A: \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$$
$$B: \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$$

□

Teorema 9.19 Il problema di corrispondenza di Post è indecidibile.

DIMOSTRAZIONE La catena di riduzioni illustrata nella Figura 9.11 è quasi completa. La riduzione di MPCP a PCP è stata dimostrata nel Teorema 9.17. La costruzione di questo paragrafo spiega come ridurre L_u a MPCP. Completiamo la prova di indecidibilità di PCP dimostrando la correttezza della costruzione, espressa dal seguente enunciato.

- M accetta w se e solo se l'istanza derivata di MPCP ha una soluzione.

(Solo se) L'Esempio 9.18 presenta l'idea fondamentale. Se w è in $L(M)$ possiamo partire dalla coppia della regola (1) e simulare la computazione di M su w. Ci serviamo di una coppia della regola (3) per copiare lo stato da ogni ID e simulare una mossa di M, e, se necessario, delle coppie della regola (2) per copiare simboli di nastro e il separatore #. Se M raggiunge uno stato accettante, attraverso le coppie della regola (4) e infine attraverso la coppia della regola (5), la stringa di A raggiunge quella di B e forma una soluzione.

(Se) Dobbiamo provare che, se l'istanza di MPCP ha soluzione, allora M accetta w. Poiché lavoriamo su MPCP ogni soluzione deve cominciare dalla prima coppia; una soluzione parziale comincia così:

$$A: \#$$
$$B: \#q_0 w\#$$

Finché non ci sono stati accettanti nella soluzione parziale, le coppie prese dalle regole (4) e (5) sono inutili. Possiamo trattare gli stati e uno o due simboli adiacenti in una ID solo con le coppie della regola (3); tutti gli altri simboli di nastro e # sono trattati da coppie della regola (2). Perciò, finché M non raggiunge uno stato accettante, tutte le soluzioni parziali sono della forma

$$A: x$$
$$B: xy$$

dove x è una sequenza di ID di M che rappresentano una computazione di M su input w, eventualmente seguita da # e dall'inizio della ID successiva α. Il residuo y è il completamento di α, un altro # e l'inizio della ID che segue α, fino al punto in cui x termina in α.

In particolare, finché M non entra in uno stato accettante, la soluzione parziale non è una soluzione: la stringa di B è più lunga di quella di A. Quindi, se c'è una soluzione, M deve a un certo punto entrare in uno stato accettante, cioè M accetta w. □

9.4.4 Esercizi

Esercizio 9.4.1 Indicate quali, fra le seguenti istanze di PCP, hanno una soluzione. Ciascuna è presentata sotto forma delle due liste A e B; le i-esime stringhe di due liste si corrispondono per $i = 1, 2, \ldots$

* a) $A = (01, 001, 10)$; $B = (011, 10, 00)$.

b) $A = (01, 001, 10)$; $B = (011, 01, 00)$.

c) $A = (ab, a, bc, c)$; $B = (bc, ab, ca, a)$.

! **Esercizio 9.4.2** Abbiamo dimostrato che PCP è indecidibile sotto l'ipotesi che l'alfabeto Σ sia arbitrario. Dimostrate che PCP è indecidibile anche limitando l'alfabeto a $\Sigma = \{0, 1\}$, riducendolo a questo caso particolare.

! **Esercizio 9.4.3** Supponiamo di limitare il PCP a un alfabeto di un simbolo come $\Sigma = \{0\}$. Anche questo caso ristretto del PCP è indecidibile?

! **Esercizio 9.4.4** Un *Post tag system* è formato da un insieme di coppie di stringhe su un alfabeto finito Σ e da una stringa iniziale. Se (w, x) è una coppia e y è una stringa qualsiasi su Σ, scriviamo $wy \vdash yx$. In altri termini, in una mossa eliminiamo un prefisso w dalla stringa "corrente" wy e aggiungiamo un suffisso x abbinato a w. Come per le derivazioni nelle grammatiche libere dal contesto, definiamo $\overset{*}{\vdash}$ come zero o più passi di $\vdash$. Dimostrate che, dato un insieme di coppie P e una stringa iniziale z, è indecidibile se $z \overset{*}{\vdash} \epsilon$. *Suggerimento*: per ogni TM M e input w, sia z la ID iniziale di M con input w, seguita da un simbolo separatore #. Scegliete le coppie P tali che ogni ID di M diventi prima o poi la ID che segue dopo una mossa di M. Se M entra in uno stato accettante, fate in modo che la stringa corrente possa essere cancellata, cioè ridotta a ϵ.

9.5 Altri problemi indecidibili

Consideriamo ora alcuni altri problemi di cui possiamo dimostrare l'indecidibilità. La tecnica principale è la riduzione di PCP al problema che vogliamo dimostrare indecidibile.

9.5.1 Problemi relativi a programmi

Partiamo da un'osservazione: si può scrivere un programma, in un normale linguaggio di programmazione, che accetta in ingresso un'istanza di PCP e ne cerca le soluzioni sistematicamente (per esempio per ordine di lunghezza – numero di coppie – della soluzione). Poiché PCP ammette alfabeti arbitrari dobbiamo codificare i simboli dell'alfabeto in binario o in un altro alfabeto fissato, come descritto nel riquadro "Il PCP come linguaggio" del Paragrafo 9.4.1.

Siamo liberi di scegliere che cosa fa il programma, per esempio arrestarsi o stampare un frase, se e quando trova una soluzione. In caso contrario il programma non esegue mai l'azione prescelta. Dunque stabilire se un programma stampa `Ciao, mondo`, se si arresta, se chiama una certa funzione, se emette il "bip" della console, o se esegue una qualsiasi azione non banale, sono tutti problemi indecidibili. Esiste in effetti un analogo del teorema di Rice per i programmi: qualsiasi proprietà non banale relativa alle operazioni di un programma (in contrasto con le proprietà lessicali o sintattiche del programma stesso) è indecidibile.

9.5.2 Indecidibilità dell'ambiguità delle CFG

Se consideriamo le analogie fra programmi e macchine di Turing, le affermazioni del Paragrafo 9.5.1 non sono sorprendenti. Spieghiamo ora come ridurre PCP a un problema del tutto estraneo ai calcolatori: la questione se una grammatica libera dal contesto sia ambigua.

L'idea di base è di considerare stringhe che rappresentano una sequenza di indici (interi), al contrario, insieme con le stringhe corrispondenti costruite secondo una delle liste di un'istanza di PCP. Queste stringhe possono essere generate da una grammatica. Anche l'analogo insieme di stringhe per l'altra lista nell'istanza di PCP può essere generato da una grammatica. L'unione delle due grammatiche, con gli ovvi aggiustamenti, genera una stringa usando le produzioni di ciascuna grammatica se e solo se l'istanza di PCP ha una soluzione. Perciò c'è una soluzione se e solo se l'unione delle grammatiche è ambigua.

Precisiamo l'idea intuitiva. Supponiamo che l'istanza di PCP sia formata dalle liste $A = w_1, w_2, \ldots, w_k$ e $B = x_1, x_2, \ldots, x_k$. Per la lista A costruiamo una CFG con A come unica variabile. I terminali sono tutti i simboli dell'alfabeto Σ impiegati nell'istanza di PCP, oltre a un insieme distinto di *simboli indice* $a_1, a_2, \ldots, a_k$, che rappresentano le scelte di coppie di stringhe in una soluzione dell'istanza di PCP. Il simbolo indice a_i

rappresenta quindi la scelta di w_i dalla lista A o di x_i dalla lista B. Le produzioni della CFG per la lista A sono:

$$\begin{array}{rcl} A & \rightarrow & w_1Aa_1 \mid w_2Aa_2 \mid \cdots \mid w_kAa_k \mid \\ & & w_1a_1 \mid w_2a_2 \mid \cdots \mid w_ka_k \end{array}$$

Denoteremo con G_A questa grammatica e con L_A il suo linguaggio. Ci riferiremo a un simile linguaggio L_A come al *linguaggio per la lista A*.

Osserviamo che le stringhe terminali derivate da G_A sono tutte quelle della forma $w_{i_1}w_{i_2}\cdots w_{i_m}a_{i_m}\cdots a_{i_2}a_{i_1}$ per un $m \geq 1$ e una lista di interi $i_1, i_2, \ldots, i_m$; ogni intero varia tra 1 e k. Tutte le forme sentenziali di G_A hanno una sola A fra le stringhe (le w) e i simboli indice (gli a) fino al momento in cui usiamo una produzione scelta fra le ultime k, che non hanno A nel corpo. Gli alberi sintattici hanno la forma illustrata nella Figura 9.16.

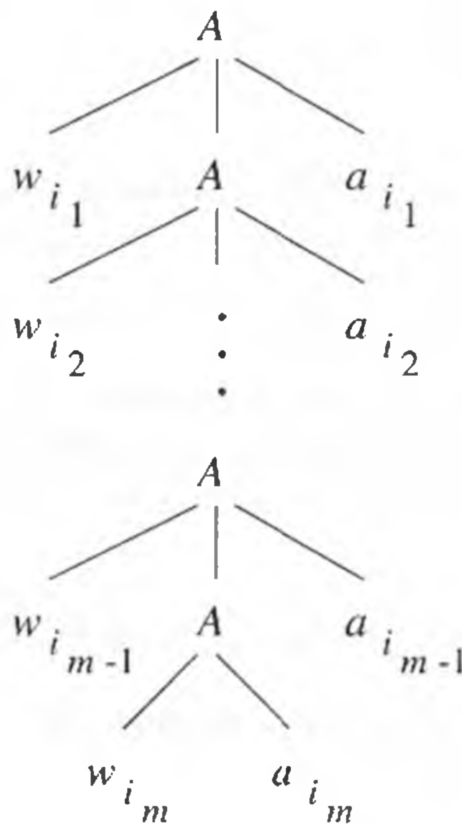


Figura 9.16 La forma degli alberi sintattici della grammatica G_A.

Possiamo anche osservare che ogni stringa terminale derivabile da A in G_A ha una sola derivazione. I simboli indice al termine della stringa determinano univocamente quale produzione applicare a ogni passo. Solamente due corpi di produzioni terminano infatti con un dato simbolo indice a_i: $A \rightarrow w_iAa_i$ e $A \rightarrow w_ia_i$. Dobbiamo usare la prima produzione se il passo di derivazione non è l'ultimo, e la seconda se è l'ultimo.

Consideriamo l'altra parte dell'istanza assegnata di PCP, la lista $B = x_1, x_2, \ldots, x_k$. Per questa lista sviluppiamo un'altra grammatica, G_B:

$$\begin{array}{rcl} B & \rightarrow & x_1 B a_1 \mid x_2 B a_2 \mid \cdots \mid x_k B a_k \mid \\ & & x_1 a_1 \mid x_2 a_2 \mid \cdots \mid x_k a_k \end{array}$$

Denoteremo il linguaggio di questa grammatica con L_B. Le osservazioni già fatte per G_A si applicano anche a G_B. In particolare una stringa terminale di L_B ha una sola derivazione, determinata dai simboli indice in coda alla stringa.

Combiniamo infine i linguaggi e le grammatiche delle due liste a formare la grammatica G_{AB} per l'istanza di PCP. Ecco i componenti di G_{AB}.

1. Le variabili A, B ed S; S è il simbolo iniziale.

2. Le produzioni $S \rightarrow A \mid B$.

3. Tutte le produzioni di G_A.

4. Tutte le produzioni di G_B.

Sosteniamo che G_{AB} è ambigua se e solo se l'istanza (A, B) di PCP ha una soluzione; questa tesi è il nucleo del prossimo teorema.

Teorema 9.20 Il problema dell'ambiguità di una CFG è indecidibile.

DIMOSTRAZIONE Abbiamo già quasi compiuto la riduzione di PCP al problema se una CFG sia ambigua; questa riduzione dimostra che il problema dell'ambiguità di una CFG è indecidibile perché PCP è indecidibile. Dobbiamo solo provare che la costruzione è corretta, cioè:

- G_{AB} è ambigua se e solo se l'istanza (A, B) di PCP ha una soluzione.

(Se) Supponiamo che $i_1, i_2, \ldots, i_m$ sia una soluzione di questa istanza di PCP. Consideriamo le due seguenti derivazioni in G_{AB}:

$$\begin{array}{l} S \Rightarrow A \Rightarrow w_{i_1} A a_{i_1} \Rightarrow w_{i_1} w_{i_2} A a_{i_2} a_{i_1} \Rightarrow \cdots \Rightarrow \\ \quad w_{i_1} w_{i_2} \cdots w_{i_{m-1}} A a_{i_{m-1}} \cdots a_{i_2} a_{i_1} \Rightarrow w_{i_1} w_{i_2} \cdots w_{i_m} a_{i_m} \cdots a_{i_2} a_{i_1} \end{array}$$

$$\begin{array}{l} S \Rightarrow B \Rightarrow x_{i_1} B a_{i_1} \Rightarrow x_{i_1} x_{i_2} B a_{i_2} a_{i_1} \Rightarrow \cdots \Rightarrow \\ \quad x_{i_1} x_{i_2} \cdots x_{i_{m-1}} B a_{i_{m-1}} \cdots a_{i_2} a_{i_1} \Rightarrow x_{i_1} x_{i_2} \cdots x_{i_m} a_{i_m} \cdots a_{i_2} a_{i_1} \end{array}$$

Poiché $i_1, i_2, \ldots, i_m$ è una soluzione, sappiamo che $w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}$. Queste due derivazioni generano la stessa stringa terminale. Trattandosi di due derivazioni a sinistra, e distinte, della stessa stringa terminale deduciamo che G_{AB} è ambigua.

(Solo se) Abbiamo già osservato che una stringa terminale non può avere più di una derivazione, né in G_A né in G_B. C'è quindi un solo caso in cui una stringa terminale ha due derivazioni a sinistra in G_{AB}: una di esse comincia da $S \Rightarrow A$ e continua con una derivazione in G_A, mentre l'altra comincia da $S \Rightarrow B$ e continua con una derivazione della stessa stringa in G_B.

La stringa con due derivazioni ha una coda di indici $a_{i_m} \cdots a_{i_2} a_{i_1}$ per un $m \geq 1$. Questa coda è una soluzione dell'istanza di PCP perché ciò che precede la coda nella stringa con due derivazioni è sia $w_{i_1} w_{i_2} \cdots w_{i_m}$ sia $x_{i_1} x_{i_2} \cdots x_{i_m}$. □

9.5.3 Il complemento di un linguaggio associato a una lista

Grazie a linguaggi liberi dal contesto come L_A per una lista A possiamo dimostrare l'indecidibilità di alcuni problemi relativi ai CFL. Ulteriori prove di indecidibilità per CFL si ricavano dallo studio del linguaggio complemento $\overline{L_A}$. Osserviamo che $\overline{L_A}$ è formato da tutte le stringhe sull'alfabeto $\Sigma \cup \{a_1, a_2, \ldots, a_k\}$ che non sono in L_A, dove Σ è l'alfabeto di un'istanza di PCP e gli a_i sono simboli distinti che rappresentano gli indici delle coppie nell'istanza di PCP.

Gli elementi interessanti di $\overline{L_A}$ sono le stringhe formate da un prefisso in Σ^* ottenuto per concatenazione di stringhe della lista A, seguito da un suffisso di simboli indice che *non* combaciano con le stringhe scelte da A. Ma in $\overline{L_A}$ ci sono anche molte stringhe che sono semplicemente malformate, nel senso che non appartengono al linguaggio dell'espressione regolare $\Sigma^*(a_1 + a_2 + \cdots + a_k)^*$.

Sosteniamo che $\overline{L_A}$ è un CFL. A differenza di L_A, non è facile scrivere una grammatica per $\overline{L_A}$, ma possiamo definire un PDA deterministico che lo accetta. La costruzione è spiegata nel prossimo teorema.

Teorema 9.21 Se L_A è il linguaggio per la lista A, $\overline{L_A}$ è un linguaggio libero dal contesto.

DIMOSTRAZIONE Sia Σ l'alfabeto delle stringhe nella lista $A = w_1, w_2, \ldots, w_k$ e sia I l'insieme dei simboli indice $I = \{a_1, a_2, \ldots, a_k\}$. Descriviamo le operazioni del DPDA P destinato ad accettare $\overline{L_A}$.

1. Finché legge simboli in Σ, P li mette sullo stack. Poiché tutte le stringhe in Σ^* sono in $\overline{L_A}$, P resta in stati accettanti.

2. Quando legge un simbolo indice in I, per esempio a_i, P toglie dallo stack per verificare se i simboli alla sommità formano w_i^R, cioè la stringa corrispondente rovesciata.

   (a) Se la risposta è negativa, l'input letto fin qui, e ogni sua continuazione, è in $\overline{L_A}$. P si sposta in uno stato accettante in cui consuma il resto dell'input senza mutare lo stack.

(b) Se w_i^R è stato tolto dallo stack, ma l'indicatore di fondo stack non è visibile, P accetta ricordando nello stato che ora si aspetta solo simboli di I e che potrebbe ancora leggere una stringa in L_A, che *non* deve accettare. P ripete il passo (2) finché la questione se l'input sia in L_A non è risolta.

(c) Se w_i^R è stato tolto dallo stack e l'indicatore di fondo stack è visibile, P ha letto un elemento di L_A, che non accetta. Ma poiché nessuna continuazione dell'input può essere in L_A, P si sposta in uno stato in cui accetta ogni input futuro senza modificare lo stack.

3. Se, dopo aver letto uno o più simboli di I, P legge un altro simbolo di Σ, l'input non è di forma tale da poter essere in L_A. Dunque P si sposta in uno stato in cui accetta l'input corrente e ogni input futuro senza modificare lo stack.

□

Per dimostrare proprietà di indecidibilità relative ai linguaggi liberi dal contesto, possiamo servirci di L_A, L_B e dei loro complementi in vario modo. Nel prossimo teorema sono riassunte alcune di queste proprietà.

Teorema 9.22 Siano G_1 e G_2 grammatiche libere dal contesto e sia R un'espressione regolare. I problemi seguenti sono indecidibili.

a) $L(G_1) \cap L(G_2) = \emptyset$?

b) $L(G_1) = L(G_2)$?

c) $L(G_1) = L(R)$?

d) $L(G_1) = T^*$ per un alfabeto T?

e) $L(G_1) \subseteq L(G_2)$?

f) $L(R) \subseteq L(G_1)$?

DIMOSTRAZIONE Ogni dimostrazione è una riduzione da PCP. Mostriamo come trasformare un'istanza (A, B) di PCP in una questione relativa alle CFG e alle espressioni regolari la cui risposta è "sì" se e solo se l'istanza di PCP ha una soluzione. In alcuni casi riduciamo PCP alla questione così come l'abbiamo enunciata nel teorema; in altri casi al complemento. I due modi si equivalgono perché se il complemento di un problema è indecidibile, il problema stesso non può essere decidibile, dal momento che i linguaggi ricorsivi sono chiusi rispetto al complemento (Teorema 9.3).

Denoteremo con Σ l'alfabeto delle stringhe di questa istanza e con I l'alfabeto dei simboli indice. Le riduzioni dipendono dal fatto che L_A, L_B, $\overline{L_A}$ ed $\overline{L_B}$ hanno ciascuno

una CFG. Costruiremo le CFG o direttamente, come nel Paragrafo 9.5.2, o tramite il PDA per i linguaggi complemento descritto nel Teorema 9.21 e la successiva trasformazione del PDA in CFG (Teorema 6.14).

a) Siano $L(G_1) = L_A$ e $L(G_2) = L_B$. Allora $L(G_1) \cap L(G_2)$ è l'insieme delle soluzioni di questa istanza di PCP. L'intersezione è vuota se e solo se non ci sono soluzioni. Tecnicamente abbiamo ridotto PCP al linguaggio delle coppie di CFG con intersezione vuota; abbiamo cioè dimostrato che il problema "l'intersezione di due CFG è non vuota?" è indecidibile. Ma, come abbiamo detto introducendo la dimostrazione, mostrare che il complemento di un problema è indecidibile equivale a dimostrare che il problema stesso è indecidibile.

b) Dato che le CFG sono chiuse per unione possiamo costruire una CFG G_1 per $\overline{L_A} \cup \overline{L_B}$. Poiché $(\Sigma \cup I)^*$ è un insieme regolare, possiamo senz'altro costruire una CFG G_2 che lo genera. Vale $\overline{L_A} \cup \overline{L_B} = \overline{L_A \cap L_B}$. Quindi a $L(G_1)$ mancano solo le stringhe che rappresentano soluzioni dell'istanza di PCP. A $L(G_2)$ non manca nessuna stringa in $(\Sigma \cup I)^*$. Dunque i loro linguaggi sono uguali se e solo se l'istanza di PCP non ha soluzioni.

c) Si ragiona come in (b), ma R è l'espressione regolare $(\Sigma \cup I)^*$.

d) Si deduce da (c) perché $\Sigma \cup I$ è l'unico alfabeto di cui $\overline{L_A} \cup \overline{L_B}$ può essere la chiusura.

e) Sia G_1 una CFG per $(\Sigma \cup I)^*$ e G_2 una CFG per $\overline{L_A} \cup \overline{L_B}$. Allora $L(G_1) \subseteq L(G_2)$ se e solo se $\overline{L_A} \cup \overline{L_B} = (\Sigma \cup I)^*$, cioè se e solo se l'istanza di PCP non ha soluzioni.

f) Si ragiona come in (e), ma R è l'espressione regolare $(\Sigma \cup I)^*$ e $L(G_1)$ è $\overline{L_A} \cup \overline{L_B}$.

□

9.5.4 Esercizi

* **Esercizio 9.5.1** Sia L l'insieme delle grammatiche libere dal contesto (codificate) G tali che $L(G)$ contiene almeno una palindroma. Dimostrate che L è indecidibile. *Suggerimento*: riducete PCP a L costruendo per ogni istanza di PCP una grammatica il cui linguaggio contiene una palindroma se e solo se l'istanza di PCP ha una soluzione.

! **Esercizio 9.5.2** Dimostrate che il linguaggio $\overline{L_A} \cup \overline{L_B}$ è regolare se e solo se è l'insieme di tutte le stringhe sul suo alfabeto, cioè se e solo se l'istanza (A, B) di PCP non ha soluzioni. Provate così che è indecidibile stabilire se una CFG genera un linguaggio

regolare. *Suggerimento*: supponete che PCP abbia una soluzione e che la stringa wx manchi da $\overline{L_A} \cup \overline{L_B}$, dove w è una stringa sull'alfabeto Σ di questa istanza di PCP e x è l'inverso della relativa stringa di simboli indice. Definite l'omomorfismo $h(0) = w$ e $h(1) = x$. Che cos'è $h^{-1}(\overline{L_A} \cup \overline{L_B})$? Per dimostrare che $\overline{L_A} \cup \overline{L_B}$ non è regolare, sfruttate il fatto che gli insiemi regolari sono chiusi rispetto a omomorfismo inverso e complementazione insieme al *pumping lemma* degli insiemi regolari.

!! **Esercizio 9.5.3** È indecidibile stabilire se il complemento di un CFL è un CFL. Dall'Esercizio 9.5.2 si ricava che la questione se il complemento di un CFL sia regolare è indecidibile, ma non è la stessa cosa. Per provare l'ipotesi dobbiamo definire un nuovo linguaggio che rappresenta le non-soluzioni di un'istanza (A, B) di PCP. Sia L_{AB} l'insieme delle stringhe della forma $w\#x\#y\#z$ tali che valgano le condizioni 1–3 e almeno una delle condizioni 4–7:

1. w e x sono stringhe sull'alfabeto Σ dell'istanza di PCP
2. y e z sono stringhe sull'alfabeto degli indici I di questa istanza
3. $\#$ è un simbolo assente sia da Σ sia da I
4. $w \neq x^R$
5. $y \neq z^R$
6. x^R *non* è ciò che la stringa di indici y genera secondo la lista B
7. w non è ciò che la stringa di indici z^R genera secondo la lista A.

Osservate che L_{AB} è formato da tutte le stringhe in $\Sigma^*\#\Sigma^*\#I^*\#I^*$ salvo che l'istanza (A, B) abbia una soluzione, ma L_{AB} è un CFL in ogni caso. Dimostrate che $\overline{L_{AB}}$ è un CFL se e solo se non ci sono soluzioni. *Suggerimento*: per imporre l'uguaglianza delle lunghezze di certe sottostringhe come suggerito nell'Esercizio 7.2.5(b), applicate la tecnica dell'omomorfismo inverso dell'Esercizio 9.5.2 e il lemma di Ogden.

9.6 Riepilogo

- ✦ *Linguaggi ricorsivi e ricorsivamente enumerabili*: i linguaggi accettati dalle macchine di Turing sono detti ricorsivamente enumerabili (RE). I linguaggi RE accettati da una TM che si arresta sempre sono detti ricorsivi.

- ✦ *Complementi di linguaggi ricorsivi e RE*: i linguaggi ricorsivi sono chiusi rispetto alla complementazione, e se un linguaggio e il suo complemento sono entrambi RE, allora entrambi sono ricorsivi. Di conseguenza il complemento di un linguaggio RE ma non ricorsivo non può essere RE.

- *Decidibilità e indecidibilità*: "decidibile" è sinonimo di "ricorsivo", anche se si tende a usare "ricorsivo" per i linguaggi e "decidibile" per i problemi (che sono linguaggi interpretati come domande). Se un linguaggio non è ricorsivo, diciamo "indecidibile" il problema espresso da tale linguaggio.

- *Il linguaggio L_d*: è l'insieme delle stringhe di 0 e di 1 che, interpretate come TM, *non* sono nel linguaggio della TM stessa. Il linguaggio L_d è un buon esempio di linguaggio non RE, vale a dire non accettato da alcuna macchina di Turing.

- *Il linguaggio universale*: il linguaggio L_u consiste delle stringhe che vengono interpretate come una TM seguita da un input per la TM. Una stringa è in L_u se la TM accetta l'input. L_u è un buon esempio di un linguaggio RE ma non ricorsivo.

- *Teorema di Rice*: qualunque proprietà non banale dei linguaggi accettati dalle macchine di Turing è indecidibile. Un esempio: per il teorema di Rice è indecidibile l'insieme dei codici di macchine di Turing il cui linguaggio è vuoto. Si tratta in effetti di un linguaggio non RE, sebbene il suo complemento, ossia l'insieme di codici di TM che accettano almeno una stringa, sia RE ma non ricorsivo.

- *Problema della corrispondenza di Post*: si chiede se, date due liste con lo stesso numero di stringhe, sia possibile scegliere una sequenza di stringhe corrispondenti dalle due liste e formare per concatenazione la stessa stringa. Il PCP è un importante esempio di problema indecidibile. È inoltre utile come problema da ridurre a un altro per provarne l'indecidibilità.

- *Problemi indecidibili sui linguaggi liberi dal contesto*: per riduzione da PCP possiamo mostrare l'indecidibilità di un numero di questioni sui CFL o le loro grammatiche. Per esempio è indecidibile se una CFG è ambigua, se un CFL è contenuto in un altro, oppure se l'intersezione di due CFL è vuota.

9.7 Bibliografia

L'indecidibilità del linguaggio universale è essenzialmente un risultato di Turing [9], anche se nel suo lavoro è espressa in termini di computazione di funzioni aritmetiche e arresto anziché in termini di linguaggi e accettazione per stato finale. Il Teorema di Rice è tratto da [8].

L'indecidibilità del problema di corrispondenza di Post è dimostrata in [7], sebbene la dimostrazione fosse già stata tratteggiata da R.W. Floyd in una memoria inedita. L'indecidibilità dei *Post tag system* (definiti nell'Esercizio 9.4.4) è tratta da [6].

Gli scritti fondamentali sull'indecidibilità di questioni concernenti i linguaggi liberi dal contesto sono [1] e [5]. L'indecidibilità della questione se una CFG sia ambigua è

stata scoperta in studi indipendenti da Cantor [2], Floyd [4], e Chomsky e Schutzenberger [3].

1. Y. Bar-Hillel, M. Perles, E. Shamir, "On formal properties of simple phrase-structure grammars," *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), pp. 143–172.

2. D. C. Cantor, "On the ambiguity problem in Backus systems," *J. ACM* **9**:4 (1962), pp. 477–479.

3. N. Chomsky, M. P. Schutzenberger, "The algebraic theory of context-free languages," *Computer Programming and Formal Systems* (1963), North Holland, Amsterdam, pp. 118–161.

4. R. W. Floyd, "On ambiguity in phrase structure languages," *Communications of the ACM* **5**:10 (1962), pp. 526–534.

5. S. Ginsburg, G. F. Rose, "Some recursively unsolvable problems in ALGOL-like languages," *J. ACM* **10**:1 (1963), pp. 29–47.

6. M. L. Minsky, "Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines," *Annals of Mathematics* **74**:3 (1961), pp. 437–455.

7. E. Post, "A variant of a recursively unsolvable problem," *Bulletin of the AMS* **52** (1946), pp. 264–268.

8. H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the AMS* **89** (1953), pp. 25–59.

9. A. M. Turing, "On computable numbers with an application to the Entscheidungsproblem," *Proc. London Math. Society* **2**:42 (1936), pp. 230–265.

Capitolo 10

Problemi intrattabili

Dopo aver visto che cosa è calcolabile, focalizziamo l'attenzione su che cosa lo è in modo efficiente. Ci occupiamo di problemi decidibili e ci chiediamo quali possono essere computati da macchine di Turing che impiegano tempo polinomiale nella dimensione dell'input. Riprendiamo due punti importanti già introdotti nel Paragrafo 8.6.3.

- I problemi risolvibili in tempo polinomiale su un tipico computer coincidono con quelli risolvibili in tempo polinomiale su una macchina di Turing.
- L'esperienza ha mostrato che la distinzione tra problemi risolvibili in tempo polinomiale e problemi che richiedono un tempo esponenziale o maggiore è fondamentale. I problemi concreti che richiedono tempo polinomiale sono quasi sempre risolvibili in un tempo accettabile, mentre quelli che richiedono tempo esponenziale non possono generalmente essere risolti, fatta eccezione per istanze piccole.

In questo capitolo presentiamo la teoria dell'"intrattabilità", ossia le tecniche per dimostrare che un problema non è risolvibile in tempo polinomiale. Partiamo da un problema specifico: la questione se un'espressione booleana possa essere *soddisfatta*, cioè se ha valore vero per un certo assegnamento dei valori di verità TRUE (vero) e FALSE (falso) alle sue variabili. Questo problema svolge per i problemi intrattabili la stessa funzione che L_u o PCP svolgono per i problemi indecidibili. Partiremo dal "teorema di Cook", secondo il quale la soddisfacibilità delle formule booleane non può essere decisa in tempo polinomiale. Spiegheremo poi come ridurre questo problema a molti altri, di cui si dimostra così l'intrattabilità.

Poiché il tema è se i problemi possono essere risolti in tempo polinomiale, bisogna cambiare il concetto di riduzione. Non basta più, infatti, un algoritmo che trasforma le istanze di un problema in istanze di un altro. L'algoritmo stesso deve impiegare al massimo un tempo polinomiale, altrimenti la riduzione non permette di concludere che il

problema di destinazione è intrattabile, per quanto il problema sorgente lo sia. Nel primo paragrafo introdurremo perciò la nozione di "riduzione polinomiale".

C'è un'altra distinzione importante tra gli enunciati che abbiamo ricavato nella teoria dell'indecidibilità e quelli che ricaveremo con la teoria dell'intrattabilità. Le dimostrazioni di indecidibilità illustrate nel Capitolo 9 sono incontrovertibili; dipendono unicamente dalla definizione di macchina di Turing e dalla matematica usuale. Per contro, i risultati sui problemi intrattabili che daremo qui si fondano tutti su un'ipotesi non dimostrata, ma fortemente creduta, la cosiddetta ipotesi $\mathcal{P} \neq \mathcal{NP}$.

Essa afferma che la classe dei problemi risolvibili da TM non deterministiche che operano in tempo polinomiale include almeno alcuni problemi che non possono essere risolti da TM deterministiche operanti in tempo polinomiale (per quanto si accettino gradi elevati del polinomio). Esistono letteralmente migliaia di problemi che *sembrano* appartenere a questa categoria, visto che possono essere risolti facilmente da una NTM in tempo polinomiale, mentre non conosciamo nessuna DTM (o, in modo equivalente, nessun programma tradizionale) che li risolva in tempo polinomiale. La teoria dell'intrattabilità ha una conseguenza importante: o quei problemi hanno tutti soluzioni polinomiali deterministiche, e la cosa ci è sfuggita per secoli, oppure nessuno ne ha, cioè richiedono veramente un tempo esponenziale.

10.1 Le classi $\mathcal{P}$ ed $\mathcal{NP}$

In questo paragrafo presentiamo i fondamenti della teoria dell'intrattabilità: le classi $\mathcal{P}$ ed $\mathcal{NP}$ dei problemi risolvibili in tempo polinomiale rispettivamente da TM deterministiche e non deterministiche, e la tecnica della riduzione polinomiale. Definiamo inoltre la nozione di "NP-completezza", una proprietà di alcuni problemi in $\mathcal{NP}$ che sono almeno tanto *ardui* (a meno di un polinomio rispetto al tempo) quanto ogni altro problema in $\mathcal{NP}$.

10.1.1 Problemi risolvibili in tempo polinomiale

Diremo che una macchina di Turing M ha *complessità in tempo* $T(n)$ (o che ha tempo di esecuzione $T(n)$) se, a fronte di un input w di lunghezza n, M si arresta dopo aver fatto al massimo $T(n)$ mosse, a prescindere dal fatto che accetti o no. La definizione si applica a qualunque funzione $T(n)$, come $T(n) = 50n^2$ o $T(n) = 3^n + 5n^4$; ci occuperemo prevalentemente del caso in cui $T(n)$ è un polinomio in n. Diremo che un linguaggio L è nella classe $\mathcal{P}$ se esiste un polinomio $T(n)$ tale che $L = L(M)$ per una TM deterministica M di complessità in tempo $T(n)$.

Tra polinomiale ed esponenziale

Nella trattazione introduttiva, e anche successivamente, ipotizziamo che tutti i programmi impieghino tempo polinomiale (cioè $O(n^k)$ per un intero k), o tempo esponenziale (cioè $O(2^{cn})$ per una costante $c > 0$), o superiore. Nella pratica gli algoritmi conosciuti per i problemi comuni rientrano effettivamente in una delle due categorie. Esistono però tempi di esecuzione compresi tra polinomiale ed esponenziale. Quando parliamo di esponenziale, intendiamo "qualunque tempo di esecuzione più grande di qualsiasi polinomio".

Un esempio di funzione tra i polinomi e gli esponenziali è $n^{\log_2 n}$. Si tratta di una funzione che cresce più velocemente di qualsiasi polinomio in n, dato che $\log n$ diventa (per valori grandi di n) maggiore di qualunque costante k. D'altra parte $n^{\log_2 n} = 2^{(\log_2 n)^2}$; per convincersene basta prendere il logaritmo di entrambi i membri. Questa funzione cresce più lentamente di 2^{cn} per qualunque $c > 0$. In altre parole, per quanto piccola sia la costante positiva c, cn finirà per essere maggiore di $(\log_2 n)^2$.

10.1.2 Un esempio: l'algoritmo di Kruskal

Il lettore conosce probabilmente diversi problemi che hanno soluzioni efficienti, magari per averli studiati in un corso di algoritmi e strutture dati. Questi problemi sono generalmente in $\mathcal{P}$. Ne esamineremo uno: trovare un albero di copertura di peso minimo (*MWST, Minimum-Weight Spanning Tree*) in un grafo.

In termini informali consideriamo i grafi come diagrammi del tipo rappresentato nella Figura 10.1. Ci sono *nodi*, qui numerati da 1 a 4, e *lati* tra alcune coppie di nodi. Ogni lato ha un *peso*, che è un intero. Un *albero di copertura* è un sottoinsieme di lati tali da connettere tutti i nodi, ma senza formare cicli. Un esempio di albero di copertura è illustrato nella Figura 10.1; si tratta dei tre lati disegnati con un tratto più spesso. Un albero di copertura si dice *di peso minimo* se il peso totale dei suoi lati è il più piccolo fra tutti gli alberi di copertura.

Un noto algoritmo *greedy*, detto *algoritmo di Kruskal*[1], serve a trovare un MWST. Ne riassumiamo le caratteristiche fondamentali.

1. Per ciascun nodo si conserva la *componente connessa* di cui fa parte rispetto ai lati dell'albero selezionati finora. Inizialmente nessun lato è selezionato, per cui ogni nodo forma da solo una componente connessa.

[1] J. B. Kruskal Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proc. AMS* 7:1 (1956), pp. 48–50.

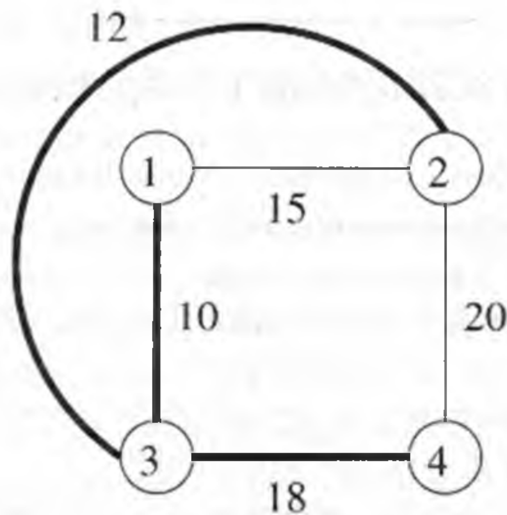


Figura 10.1 Un grafo; i tratti più spessi indicano un albero di copertura di peso minimo.

2. Si considera uno dei lati di peso minimo non ancora esaminati, scegliendolo arbitrariamente. Se questo lato unisce due nodi in componenti connesse diverse:

   (a) si seleziona il lato, che farà parte dell'albero di copertura

   (b) si riuniscono le due componenti connesse coinvolte, associando a ogni loro nodo lo stesso numero identificativo della componente connessa.

   Se invece il lato selezionato unisce due nodi della stessa componente, esso non appartiene all'albero di copertura perché darebbe vita a un ciclo.

3. Si continua a considerare nuovi lati finché si sono esaminati tutti, o fino a quando il numero di lati selezionati è pari al numero di nodi meno uno. Nel secondo caso tutti i nodi devono già trovarsi in una componente connessa, e si può evitare di esaminare altri lati.

Esempio 10.1 Nel grafo della Figura 10.1 consideriamo in primo luogo il lato $(1, 3)$, che ha il peso più basso, 10. Poiché 1 e 3 sono inizialmente in componenti diverse, lo accettiamo e facciamo sì che 1 e 3 abbiano lo stesso numero di componente, per esempio "componente 1". Il lato successivo nell'ordine di peso è $(2, 3)$, con peso 12. Poiché 2 e 3 sono in componenti diverse, lo accettiamo e uniamo il nodo 2 alla "componente 1". Il terzo lato è $(1, 2)$, con peso 15. 1 e 2 però si trovano ora nella stessa componente, per cui lo rifiutiamo e procediamo con il quarto, $(3, 4)$. Dato che 4 non si trova nella "componente 1", lo accettiamo. A questo punto abbiamo tre lati nell'albero di copertura di un grafo di 4 nodi e possiamo quindi fermarci. □

È possibile implementare quest'algoritmo (usando un computer, non una macchina di Turing) e avere, per un grafo di m nodi ed e lati, un tempo di esecuzione $O(m +$

$e \log e$). L'implementazione più semplice procede in e "turni". Una tabella fornisce la componente di ogni nodo a ogni istante. In tempo $O(e)$ scegliamo fra i lati restanti quello di peso più piccolo, e in tempo $O(1)$ troviamo le componenti dei due nodi uniti dal lato. Se si trovano in componenti diverse, in tempo $O(m)$ riuniamo tutti i nodi che recano quei valori, percorrendo la tabella dei nodi. Il tempo totale impiegato dall'algoritmo è $O\big(e(e+m)\big)$. Questo tempo di esecuzione è polinomiale nella "dimensione" dell'input, che possiamo informalmente considerare come la somma di e ed m.

Se proviamo a tradurre queste idee nei termini di una macchina di Turing, ci troviamo di fronte ad alcuni problemi.

- Nello studio di algoritmi ci imbattiamo in "problemi" che richiedono una varietà di forme di output, come la lista dei lati di un MWST. Trattando macchine di Turing, possiamo considerare i problemi solo come linguaggi, e l'unico output è "sì" o "no", ossia accettazione o rifiuto. Per esempio il problema MWST potrebbe essere espresso così: "dato il grafo G e l'intero W, G ha un albero di copertura di peso non superiore a W?" Dato che non chiede di determinare l'albero di copertura, questo problema sembra più facile rispetto alla versione di MWST che abbiamo studiato. Ma nella teoria dell'intrattabilità vogliamo in genere dimostrare che un problema è difficile, non facile, e il fatto che la versione sì/no di un problema sia difficile implica che è difficile anche la versione in cui si deve calcolare una risposta completa.

- Se possiamo considerare la "dimensione" di un grafo come il numero dei suoi nodi e lati, l'input di una TM è una stringa su un alfabeto finito. Di conseguenza gli elementi del problema, come nodi e lati, devono essere adeguatamente codificati. Ne segue che gli input di una macchina di Turing sono di solito più lunghi rispetto alla "dimensione" intuitiva dell'input. Ci sono però due ragioni per cui la differenza non è rilevante.

  1. La differenza tra la dimensione come stringa di input di una TM e la dimensione come input di un problema informale non è mai superiore a un fattore piccolo, solitamente il logaritmo della dimensione dell'input. Di conseguenza quanto può essere fatto in tempo polinomiale rispetto a una misura può essere fatto in tempo polinomiale anche rispetto all'altra.

  2. La lunghezza di una stringa che rappresenta l'input è effettivamente una misura più precisa del numero di byte che un computer deve leggere per ottenere l'input. Per esempio, se un nodo è rappresentato da un intero, allora il numero di byte necessari per rappresentarlo è proporzionale al logaritmo della dimensione dell'intero, e non è "un byte per qualsiasi nodo", come potremmo immaginare in una spiegazione informale della dimensione dell'input.

Esempio 10.2 Esaminiamo una possibile codifica per i grafi e i limiti di peso che potrebbe costituire l'input per il problema MWST. La codifica usa cinque simboli: 0, 1, le parentesi sinistra e destra, e la virgola.

1. Attribuiamo ai nodi gli interi da 1 a m.
2. Iniziamo la codifica con il valore di m in binario e il limite di peso W in binario separati dalla virgola.
3. Se esiste un lato tra i nodi i e j con peso w, collochiamo (i, j, w) nella codifica. Gli interi i, j e w sono rappresentati in binario. L'ordine di i e j entro un lato e l'ordine dei lati all'interno della codifica sono irrilevanti.

Una possibile codifica per il grafo della Figura 10.1 e il limite $W = 40$ è

$$100, 101000(1, 10, 1111)(1, 11, 1010)(10, 11, 1100)(10, 100, 10100)(11, 100, 10010)$$

□

Se rappresentiamo gli input del problema MWST come nell'Esempio 10.2, un input di lunghezza n può rappresentare al massimo $O(n/\log n)$ lati. Se i lati sono pochi, il numero di nodi m può essere esponenziale in n. Ma se il numero di lati e non è almeno $m - 1$, il grafo non può essere connesso, e dunque non avrà alcun MWST, indipendentemente dai lati. Di conseguenza, se il numero di nodi non è almeno una certa frazione di $n/\log n$, non c'è bisogno di eseguire l'algoritmo di Kruskal; diciamo semplicemente "no, non esiste alcun albero di copertura di questo peso".

Perciò, se abbiamo un limite superiore, espresso come funzione di m ed e, al tempo di esecuzione dell'algoritmo di Kruskal, per esempio il limite $O\big(e(m + e)\big)$ sviluppato in precedenza, possiamo sostituire per prudenza sia m sia e con n, e dire che il tempo di esecuzione, come funzione della lunghezza di input n, è $O\big(n(n + n)\big)$, cioè $O(n^2)$. In effetti c'è un'implementazione migliore dell'algoritmo di Kruskal che richiede tempo $O(n \log n)$, ma in questa sede non ce ne occupiamo.

Dobbiamo tener conto che usiamo una macchina di Turing come modello di computazione, mentre l'algoritmo descritto è destinato a essere realizzato in un linguaggio di programmazione con strutture dati come gli array e i puntatori. Affermiamo però di poter implementare quella versione dell'algoritmo di Kruskal su una TM multinastro in $O(n^2)$ passi. Descriviamo l'impiego dei nastri supplementari.

1. Un nastro memorizza i nodi e i loro numeri di componente correnti. La lunghezza della tabella è $O(n)$.

2. Mentre percorriamo i lati sul nastro di input un altro nastro contiene il peso del lato che al momento è il più piccolo tra i lati che non sono marcati come "usati". Possiamo impiegare una seconda traccia del nastro di input per indicare i lati selezionati come lati residui di minor peso nei passi precedenti dell'algoritmo. La ricerca del lato residuo di peso minimo richiede tempo $O(n)$, perché ciascun lato viene preso in considerazione una sola volta e i confronti di peso possono essere fatti percorrendo, da destra a sinistra, i numeri binari.

3. Quando un lato viene selezionato in un passo, collochiamo i suoi due nodi su un nastro e ne cerchiamo le componenti nella tabella dei nodi e componenti. L'operazione richiede tempo $O(n)$.

4. Un nastro contiene le due componenti, i e j, che devono essere riunite quando si scopre che un lato collega due componenti precedentemente non connesse. Percorriamo la tabella di nodi e componenti, e a ogni nodo che si trova nella componente i associamo j come nuovo numero di componente. Anche quest'operazione richiede tempo $O(n)$.

Il lettore dovrebbe essere in grado di completare da solo il ragionamento per cui un passo può essere eseguito in tempo $O(n)$ su una TM multinastro. Poiché il numero di passi è al massimo n, concludiamo che un tempo $O(n^2)$ è sufficiente per una TM multinastro. Sfruttiamo ora il Teorema 8.10, secondo il quale tutto ciò che una TM multinastro può fare in s passi può essere fatto da una TM a nastro singolo in $O(s^2)$ passi. Di conseguenza, se la TM multinastro fa $O(n^2)$ passi, possiamo costruire una TM a nastro singolo che fa altrettanto in $O\big((n^2)^2\big) = O(n^4)$ passi. Concludiamo che la versione sì/no del problema MWST, ossia "il grafo ha un MWST di peso totale non superiore a W?", è in $\mathcal{P}$.

10.1.3 Tempo polinomiale non deterministico

Nello studio dell'intrattabilità una classe fondamentale di problemi è quella dei problemi risolvibili da una TM non deterministica in tempo polinomiale. In termini formali diremo che un linguaggio L è nella classe $\mathcal{NP}$ (polinomiale non deterministica) se esiste una TM non deterministica M e una complessità polinomiale in tempo $T(n)$ tale che $L = L(M)$ e che, quando a M viene dato un input di lunghezza n, in M non ci sono sequenze di mosse più lunghe di $T(n)$.

La prima osservazione è che $\mathcal{P} \subseteq \mathcal{NP}$, dato che ogni TM deterministica è una TM non deterministica senza possibilità di scelta fra mosse. Sembra però che $\mathcal{NP}$ contenga molti problemi non in $\mathcal{P}$. Intuitivamente questo accade perché una NTM che opera in tempo polinomiale ha la capacità di congetturare un numero esponenziale di soluzioni possibili al problema e controllarne ciascuna in tempo polinomiale, "in parallelo". Osserviamo tuttavia:

Una variante dell'accettazione non deterministica

Abbiamo imposto alle NTM di arrestarsi in tempo polinomiale in tutte le diramazioni, a prescindere dal fatto che accettino o no. Avremmo potuto imporre il limite di tempo polinomiale $T(n)$ solo sulle diramazioni che conducono all'accettazione; cioè avremmo potuto definire $\mathcal{NP}$ come insieme dei linguaggi accettati da NTM che accettano tramite almeno una sequenza di non più di $T(n)$ mosse, per un polinomio $T(n)$.

Avremmo comunque ottenuto la stessa classe di linguaggi. Sapendo infatti che, se M accetta, lo fa in $T(n)$ mosse, possiamo modificarla in modo che conti fino a $T(n)$ su una traccia distinta del nastro e si arresti senza accettare se supera $T(n)$. La M modificata può compiere $O(T^2(n))$ passi, ma $T^2(n)$ è un polinomio se lo è $T(n)$.

In effetti avremmo potuto anche definire $\mathcal{P}$ per accettazione da parte di TM che accettano in tempo non superiore a $T(n)$, per un polinomio $T(n)$, ma che possono non arrestarsi se non accettano. Con la stessa costruzione usata per le NTM possiamo infatti modificare la DTM in modo che conti fino a $T(n)$ e si arresti se si eccede il limite. La DTM opererebbe in tempo $O(T^2(n))$.

- la questione, tuttora aperta, se $\mathcal{P} = \mathcal{NP}$, ossia se effettivamente tutto ciò che una NTM può fare in tempo polinomiale possa essere fatto da una DTM in tempo polinomiale, eventualmente di grado più elevato, è una delle questioni più profonde della matematica.

10.1.4 Un esemplare di $\mathcal{NP}$: il problema del commesso viaggiatore

Per avere un'idea della portata di $\mathcal{NP}$, consideriamo un esempio di problema che sembra essere in $\mathcal{NP}$ ma non in $\mathcal{P}$: il *problema del commesso viaggiatore* (*TSP, Traveling Salesman Problem*). L'input di TSP è uguale a quello di MWST: un grafo con pesi interi sui lati, come nella Figura 10.1, e un limite di peso W. La domanda è se il grafo abbia un "circuito hamiltoniano" di peso totale non superiore a W. Un *circuito hamiltoniano* è un insieme di lati che connettono i nodi in un unico ciclo in cui ogni nodo figura una sola volta. Osserviamo che il numero dei lati di un circuito hamiltoniano deve coincidere con il numero di nodi del grafo.

Esempio 10.3 Il grafo della Figura 10.1 ha un solo circuito hamiltoniano: il ciclo $(1, 2, 4, 3, 1)$. Il suo peso totale è $15 + 20 + 18 + 10 = 63$. Pertanto, se W è 63 o più, la risposta è "sì"; se $W < 63$ la risposta è "no".

Il TSP per grafi di quattro nodi è ingannevolmente semplice, dato che non possono mai esistere più di due circuiti hamiltoniani diversi, salvo che per il nodo iniziale e per la direzione in cui li percorriamo. In grafi di m nodi il numero di cicli distinti cresce come $O(m!)$, il fattoriale di m, che è più di 2^{cm} per qualunque costante c. □

Qualsiasi modo di risolvere il TSP comporta apparentemente l'esame di tutti i cicli e il calcolo, per ognuno, del peso totale. In realtà si possono tralasciare alcune scelte palesemente sbagliate; in ogni caso, se siamo sfortunati nell'ordine in cui prendiamo in esame i cicli, sembra necessario esaminarne un numero esponenziale prima di poter concludere che nessun ciclo rispetta il limite di peso W o di trovarne uno.

D'altra parte, se avessimo un computer non deterministico, potremmo scegliere una permutazione dei nodi e computare il peso totale del ciclo corrispondente. Se l'input fosse di lunghezza n, nessuna diramazione richiederebbe più di $O(n)$ passi. Su una NTM multinastro è possibile scegliere una permutazione in $O(n^2)$ passi e calcolare il suo peso totale in un tempo analogo. Di conseguenza una NTM a nastro singolo può risolvere il TSP in un tempo massimo $O(n^4)$. La conclusione è che il TSP è in $\mathcal{NP}$.

10.1.5 Riduzioni polinomiali

La tecnica principale che adottiamo per dimostrare che un problema P_2 non può essere risolto in tempo polinomiale (cioè che P_2 non è in $\mathcal{P}$) è la riduzione di un problema P_1, che si sa non essere in $\mathcal{P}$, a P_2.[2] Il metodo è illustrato nella Figura 8.7, riprodotta come Figura 10.2.

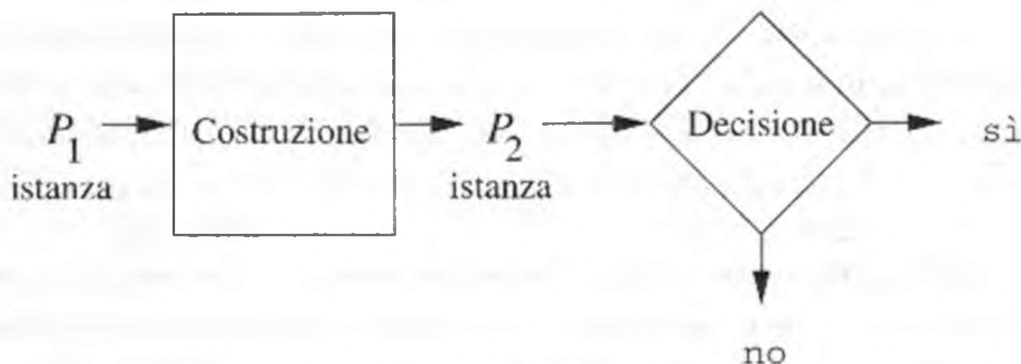


Figura 10.2 Schema di riduzione.

Supponiamo di voler dimostrare l'enunciato "se P_2 è in $\mathcal{P}$, allora lo è anche P_1". Poiché sosteniamo che P_1 *non* è in $\mathcal{P}$, potremmo allora affermare che neanche P_2 è in $\mathcal{P}$. La

[2] L'affermazione non è del tutto vera. In realtà *ipotizziamo* che P_1 non sia in $\mathcal{P}$ in base a un indizio molto forte: il fatto che P_1 è "NP-completo", un concetto che tratteremo nel Paragrafo 10.1.6. Dimostriamo che anche P_2 è "NP-completo", e abbiamo così l'indizio che P_1 non è in $\mathcal{P}$.

semplice esistenza dell'algoritmo etichettato "Costruzione" nella Figura 10.2 non basta però a dimostrare l'enunciato.

Supponiamo, per esempio, che quando riceve in input un'istanza di P_1 di lunghezza m, l'algoritmo produca una stringa di output di lunghezza 2^m, che viene passata all'algoritmo ipotetico in tempo polinomiale per P_2. Se l'algoritmo di decisione impiega un tempo $O(n^k)$, su un input di lunghezza 2^m il suo tempo di esecuzione sarebbe $O(2^{km})$, che è esponenziale in m. Di conseguenza, quando all'algoritmo di decisione per P_1 viene dato un input di lunghezza m, esso impiega un tempo esponenziale in m. Si tratta di dati perfettamente coerenti con la situazione in cui P_2 è in $\mathcal{P}$ e P_1 non è in $\mathcal{P}$.

Anche se l'algoritmo che costruisce un'istanza di P_2 da un'istanza di P_1 produce sempre un'istanza polinomiale nella dimensione dell'input, non è detto che si giunga alla conclusione desiderata. Per esempio supponiamo che l'istanza di P_2 costruita sia della stessa dimensione, m, dell'istanza P_1, ma che l'algoritmo di costruzione impieghi un tempo esponenziale in m, poniamo $O(2^m)$. Un algoritmo di decisione per P_2 che impiega tempo polinomiale $O(n^k)$ su input di lunghezza n comporta l'esistenza di un algoritmo di decisione per P_1 che impiega tempo $O(2^m + m^k)$ su input di lunghezza m. Questo limite sul tempo di esecuzione considera il fatto che bisogna compiere la traduzione a P_2 oltre a risolvere l'istanza di P_2. Anche qui è possibile che P_2 sia in $\mathcal{P}$ e P_1 no.

Il vincolo che dobbiamo porre alla traduzione da P_1 a P_2 è che richieda un tempo polinomiale nella lunghezza dell'input. Osserviamo che se la traduzione impiega tempo $O(m^j)$ su input di lunghezza m, l'istanza di P_2 non può essere più lunga del numero di passi compiuti, cioè al massimo cm^j per una costante c. Possiamo ora dimostrare che se P_2 è in $\mathcal{P}$, lo è anche P_1.

Per la dimostrazione supponiamo di poter decidere l'appartenenza a P_2 di una stringa di lunghezza n in tempo $O(n^k)$. Allora possiamo decidere l'appartenenza a P_1 di una stringa di lunghezza m in tempo $O\big(m^j + (cm^j)^k\big)$; il termine m^j corrisponde al tempo per la traduzione e il termine $(cm^j)^k$ al tempo per decidere l'istanza risultante di P_2. Semplificando l'espressione vediamo che P_1 può essere risolto in tempo $O(m^j + cm^{jk})$. Poiché c, j e k sono costanti, il tempo è polinomiale in m, e deduciamo che P_1 è in $\mathcal{P}$.

Per questi motivi, nella teoria dell'intrattabilità useremo solo *riduzioni polinomiali*. Una riduzione da P_1 a P_2 è polinomiale se il tempo impiegato è un polinomio nella lunghezza dell'istanza di P_1. Ne segue che la lunghezza dell'istanza di P_2 è un polinomio nella lunghezza dell'istanza di P_1.

10.1.6 Problemi NP-completi

Ci occupiamo ora di una famiglia di problemi composta dai più noti candidati ad appartenere a $\mathcal{NP}$ ma non a $\mathcal{P}$. Sia L un linguaggio (problema). Diciamo che L è *NP-completo* se i seguenti enunciati sono veri.

1. L è in $\mathcal{NP}$.

2. Per ogni linguaggio L' in $\mathcal{NP}$ esiste una riduzione polinomiale di L' a L.

Come vedremo, un esempio di problema NP-completo è il problema del commesso viaggiatore, introdotto nel Paragrafo 10.1.4. Poiché si ritiene che $\mathcal{P} \neq \mathcal{NP}$, e che in particolare tutti i problemi NP-completi sono in $\mathcal{NP} - \mathcal{P}$, consideriamo la dimostrazione di NP-completezza di un problema come la dimostrazione che quel problema non è in $\mathcal{P}$.

Dimostreremo che un primo problema, detto SAT (abbreviazione di *boolean satisfiability*, soddisfacibilità booleana), è NP-completo provando che il linguaggio di ogni NTM con tempo polinomiale ha una riduzione polinomiale a SAT. Se disponiamo di alcuni problemi NP-completi, possiamo dimostrare l'NP-completezza di un nuovo problema riducendo a questo uno di quelli noti tramite una riduzione polinomiale. Il teorema che segue spiega perché una riduzione di questo tipo serve a dimostrare che il problema d'arrivo è NP-completo.

Teorema 10.4 Sia P_1 un problema NP-completo e P_2 un problema in $\mathcal{NP}$. Se esiste una riduzione polinomiale di P_1 a P_2, allora P_2 è NP-completo.

DIMOSTRAZIONE Dobbiamo dimostrare che ogni linguaggio L in $\mathcal{NP}$ si riduce in tempo polinomiale a P_2. Sappiamo che esiste una riduzione polinomiale di L a P_1 che impiega un tempo polinomiale $p(n)$. Di conseguenza una stringa w in L di lunghezza n viene convertita in una stringa x in P_1 di lunghezza massima $p(n)$.

Sappiamo inoltre che esiste una riduzione polinomiale di P_1 a P_2; supponiamo che essa impieghi un tempo polinomiale $q(m)$. La riduzione trasforma dunque x in una stringa y in P_2, impiegando un tempo massimo $q\big(p(n)\big)$. Pertanto la trasformazione di w in y impiega un tempo massimo $p(n) + q\big(p(n)\big)$, che è polinomiale. Concludiamo che L è riducibile in tempo polinomiale a P_2. Dato che L è un linguaggio arbitrario in $\mathcal{NP}$, abbiamo dimostrato che l'intera classe $\mathcal{NP}$ si riduce polinomialmente a P_2, cioè che P_2 è NP-completo. □

C'è un altro importante teorema da dimostrare in relazione ai problemi NP-completi: se uno qualunque di essi è in $\mathcal{P}$, allora tutta la classe $\mathcal{NP}$ è contenuta in $\mathcal{P}$. Poiché siamo convinti che molti problemi in $\mathcal{NP}$ *non* sono in $\mathcal{P}$, consideriamo la dimostrazione che un problema è NP-completo equivalente alla dimostrazione che esso non ha alcun algoritmo polinomiale, e quindi neppure una soluzione pratica basata su computer.

Teorema 10.5 Se un problema NP-completo P è in $\mathcal{P}$, allora $\mathcal{P} = \mathcal{NP}$.

DIMOSTRAZIONE Supponiamo che P sia NP-completo e si trovi in $\mathcal{P}$. Ogni linguaggio L in $\mathcal{NP}$ si riduce allora in tempo polinomiale a P. Se P è in $\mathcal{P}$, allora L è in $\mathcal{P}$, come discusso nel Paragrafo 10.1.5. □

Problemi NP-hard

Alcuni problemi L sono così "ardui" (*hard*) che, sebbene sia possibile dimostrare la condizione (2) della definizione di NP-completezza (ogni linguaggio in $\mathcal{NP}$ si riduce a L in tempo polinomiale), non possiamo dimostrare la condizione (1): che L è in $\mathcal{NP}$. In questo caso chiameremo L *NP-hard*. In precedenza abbiamo usato il termine informale "intrattabile" in relazione a problemi che sembrano richiedere tempo esponenziale. L'uso di "intrattabile" in luogo di "NP-hard" è generalmente accettabile, anche se in principio potrebbero esserci problemi che richiedono tempo esponenziale pur non essendo NP-hard in senso formale.

Una dimostrazione che L è NP-hard basta per indicare che L richiede molto probabilmente un tempo esponenziale o superiore. Se però L non è in $\mathcal{NP}$, la sua difficoltà non è utile a dimostrare che tutti i problemi NP-completi sono difficili. In altre parole può valere $\mathcal{P} = \mathcal{NP}$, anche se L richiede tempo esponenziale.

10.1.7 Esercizi

Esercizio 10.1.1 Determinate l'MWST del grafo ottenuto modificando i pesi nella Figura 10.1 come indicato qui sotto.

* a) Modificate da 10 a 25 il peso del lato $(1, 3)$.

b) Modificate in 16 il peso del lato $(2, 4)$.

Esercizio 10.1.2 Qual è il circuito hamiltoniano di peso minimo del grafo ottenuto aggiungendo alla Figura 10.1 un lato di peso 19 tra i nodi 1 e 4?

*! **Esercizio 10.1.3** Supponiamo che esista un problema NP-completo con una soluzione deterministica che impiega un tempo $O(n^{\log_2 n})$. Osserviamo che questa funzione si colloca tra i polinomi e gli esponenziali, e non si trova in nessuna delle due classi di funzioni. Che cosa potremmo dire del tempo di esecuzione di un problema arbitrario in $\mathcal{NP}$?

!! **Esercizio 10.1.4** Considerate i grafi i cui nodi sono i punti a coordinate intere di un cubo a n dimensioni di lato m, cioè i vettori $(i_1, i_2, \ldots, i_n)$, dove ogni i_j è compreso tra 1 ed m. Due nodi sono uniti da un lato se e solo se differiscono di uno in esattamente una dimensione. Per esempio il caso $n = 2$ e $m = 2$ è un quadrato, $n = 3$ e $m = 2$ è un cubo, mentre $n = 2$ e $m = 3$ è il grafo rappresentato nella Figura 10.3. Alcuni di questi grafi hanno un circuito hamiltoniano e altri no. Per esempio il quadrato ne ha evidentemente uno, così come il cubo, sebbene non tanto evidente. Un ciclo è $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 1)$,

Altre nozioni di NP-completezza

Il vero obiettivo dello studio dell'NP-completezza è il Teorema 10.5, cioè l'identificazione dei problemi la cui appartenenza a $\mathcal{P}$ implica $\mathcal{P} = \mathcal{NP}$. La definizione di "NP-completo" usata finora, detta spesso *Karp-completezza* perché venne utilizzata per la prima volta in un fondamentale studio di R. Karp sull'argomento, è adatta a cogliere ogni problema che possiamo ritenere soddisfi il Teorema 10.5. Altre nozioni più ampie di NP-completezza sono compatibili con il Teorema 10.5.

Per esempio Cook, nel suo studio originale sul tema, definiva un problema P come "NP-completo" se, dato un *oracolo* per il problema P, ossia un meccanismo che in una unità di tempo può rispondere a qualunque domanda sull'appartenenza di una data stringa a P, è possibile riconoscere qualunque linguaggio in $\mathcal{NP}$ in tempo polinomiale. Questo tipo di NP-completezza è detta *Cook-completezza*. In un certo senso la Karp-completezza è il caso speciale in cui si pone all'oracolo una sola domanda. La Cook-completezza però permette anche di "invertire" la risposta; per esempio si potrebbe porre all'oracolo una domanda e poi rispondere il contrario di quanto dice l'oracolo. Una conseguenza della definizione di Cook è che anche i complementi dei problemi NP-completi sarebbero NP-completi. Usando una nozione più ristretta della Karp-completezza, come effettivamente facciamo, nel Paragrafo 11.1 siamo in grado di operare un'importante distinzione tra i problemi NP-completi (nel senso di Karp) e i loro complementi.

$(0,1,0)$, $(1,1,0)$, $(1,1,1)$, $(1,0,1)$, $(1,0,0)$, e ritorno a $(0,0,0)$. La Figura 10.3 non ha alcun circuito hamiltoniano.

a) Dimostrate che la Figura 10.3 non ha alcun circuito hamiltoniano. *Suggerimento*: considerate che cosa succede quando un circuito hamiltoniano ipotetico passa attraverso il nodo centrale. Da dove può venire e verso dove può andare senza tagliar via una parte del grafo dal circuito hamiltoniano?

b) Per quali valori di n ed m esiste un circuito hamiltoniano?

! Esercizio 10.1.5 Supponete di avere una codifica delle grammatiche libere dal contesto in un alfabeto finito fissato. Considerate i due linguaggi seguenti.

1. $L_1 = \{(G, A, B) \mid G$ è una CFG (codificata), A e B sono variabili (codificate) di G e gli insiemi di stringhe terminali derivate da A e B sono identici$\}$.

2. $L_2 = \{(G_1, G_2) \mid G_1$ e G_2 sono CFG (codificate) e $L(G_1) = L(G_2)\}$.

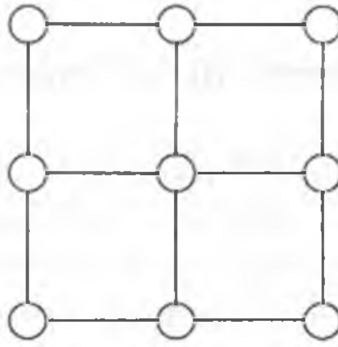


Figura 10.3 Un grafo con $n = 2$, $m = 3$.

Eseguite quanto segue e rispondente al quesito posto.

* a) Dimostrate che L_1 è riducibile in tempo polinomiale a L_2.

b) Dimostrate che L_2 è riducibile in tempo polinomiale a L_1.

* c) Che cosa dicono (a) e (b) circa la possibilità che L_1 ed L_2 siano NP-completi?

Esercizio 10.1.6 In quanto classi di linguaggi, $\mathcal{P}$ ed $\mathcal{NP}$ hanno ciascuno determinate proprietà di chiusura. Dimostrate che $\mathcal{P}$ è chiuso rispetto alle seguenti operazioni:

a) inversione

* b) unione

*! c) concatenazione

! d) chiusura (star)

e) omomorfismo inverso

* f) complementazione.

Esercizio 10.1.7 Anche $\mathcal{NP}$ è chiuso rispetto a ognuna delle operazioni elencate per $\mathcal{P}$ nell'Esercizio 10.1.6, con l'eccezione (presunta) della complementazione al punto (f). Non è noto se $\mathcal{NP}$ sia chiuso rispetto alla complementazione o no: questo punto verrà discusso più avanti nel Paragrafo 11.1. Dimostrate che i punti da (a) a (e) dell'Esercizio 10.1.6 valgono anche per $\mathcal{NP}$.

10.2 Un problema NP-completo

Presentiamo ora il primo problema NP-completo: decidere se un'espressione booleana è soddisfacibile. Ne dimostriamo l'NP-completezza riducendo esplicitamente il linguaggio di qualunque TM non deterministica polinomiale in tempo al problema della soddisfacibilità.

10.2.1 Il problema della soddisfacibilità

Le *espressioni booleane* sono costruite a partire da:

1. variabili a valori booleani, ossia 1 (vero) o 0 (falso)

2. gli operatori binari $\wedge$ e $\vee$, che significano AND e OR logico di due espressioni

3. l'operatore unario $\neg$, che significa la negazione logica

4. parentesi per raggruppare operatori e operandi in modo da modificare, se necessario, l'ordine di precedenza predefinito: prima $\neg$, poi $\wedge$, infine $\vee$.

Esempio 10.6 Un esempio di espressione booleana è $x \wedge \neg(y \vee z)$. La sottoespressione $y \vee z$ è vera se la variabile y o la variabile z ha valore vero, ed è falsa se le due variabili sono false. La sottoespressione più ampia $\neg(y \vee z)$ è vera quando $y \vee z$ è falsa, ossia quando sia y sia z sono false. Se y o z oppure entrambe sono vere, allora $\neg(y \vee z)$ è falsa.

Infine consideriamo l'intera espressione. In quanto AND logico di due sottoespressioni, essa è vera quando entrambe le sottoespressioni sono vere. Quindi $x \wedge \neg(y \vee z)$ è vera quando x è vera, y è falsa e z è falsa. □

Un *assegnamento di valori di verità* per una data espressione booleana E assegna i valori vero o falso a ognuna delle variabili presenti in E. Il *valore* dell'espressione E rispetto a un assegnamento di valori di verità T, denotato come $E(T)$, è il risultato della valutazione di E con ciascuna variabile x sostituita dal valore $T(x)$ (vero o falso) che T assegna a x.

Un assegnamento di valori di verità T *soddisfa* l'espressione booleana E se $E(T) = 1$, cioè se rende vera l'espressione E. Un'espressione booleana E si dice *soddisfacibile* se esiste almeno un assegnamento di valori di verità T che soddisfa E.

Esempio 10.7 L'espressione $x \wedge \neg(y \vee z)$ dell'Esempio 10.6 è soddisfacibile. Abbiamo visto che l'assegnamento di valori di verità T definito da $T(x) = 1$, $T(y) = 0$ e $T(z) = 0$ la soddisfa, in quanto rende vero (1) il valore dell'espressione. Abbiamo inoltre osservato

che T è l'*unico* assegnamento di valori di verità che soddisfa l'espressione, perché le altre sette combinazioni di valori per le tre variabili danno all'espressione il valore falso (0).

Come ulteriore esempio consideriamo l'espressione $E = x \wedge (\neg x \vee y) \wedge \neg y$. Sosteniamo che E non è soddisfacibile. Poiché esistono solo due variabili, il numero di assegnamenti di valori di verità è $2^2 = 4$; il lettore può facilmente provarli tutti e verificare che E ha sempre valore 0. Possiamo però anche ragionare in altro modo. E è vera solo se tutti e tre i termini composti per mezzo di $\wedge$ sono veri. Ciò significa che x dev'essere vera (per il primo termine) e y dev'essere falsa (per l'ultimo termine). Ma rispetto a questo assegnamento di valori di verità il termine mediano $\neg x \vee y$ è falso. Di conseguenza E non può essere verificata ed è quindi insoddisfacibile.

Abbiamo visto un esempio in cui un'espressione ha un solo assegnamento di valori di verità soddisfacente e un esempio in cui non ne ha nessuno. Ci sono anche molti esempi in cui un'espressione ha più di un assegnamento soddisfacente. Per esemplificare consideriamo $F = x \vee \neg y$. Il valore di F è 1 per tre assegnamenti:

1. $T_1(x) = 1, T_1(y) = 1$

2. $T_2(x) = 1, T_2(y) = 0$

3. $T_3(x) = 0, T_3(y) = 0$.

F ha valore 0 solo per il quarto assegnamento, in cui $x = 0$ e $y = 1$. Quindi F è soddisfacibile. □

Il *problema della soddisfacibilità* è così definito:

- un'espressione booleana assegnata è soddisfacibile?

In genere denoteremo con SAT il problema della soddisfacibilità. Enunciato come linguaggio, il problema SAT è l'insieme delle espressioni booleane (codificate) soddisfacibili. Le stringhe che non sono codifiche valide per un'espressione booleana o che sono codifiche per espressioni booleane non soddisfacibili non appartengono a SAT.

10.2.2 Rappresentazione di istanze di SAT

I simboli in un'espressione booleana sono $\wedge$, $\vee$, $\neg$, le parentesi aperta e chiusa, e i simboli che rappresentano le variabili. La soddisfacibilità di un'espressione non dipende dai nomi delle variabili, ma solo dal fatto che due occorrenze di variabili rappresentino la stessa variabile o due variabili diverse. Di conseguenza possiamo presumere che le variabili siano $x_1, x_2, \ldots$, anche se negli esempi continueremo a usare nomi di variabili come y o z, oltre a x. Assumiamo inoltre di rinominare le variabili in modo da usare gli indici più

bassi. Per esempio non useremo x_5 se nella stessa espressione non compaiono anche le variabili da x_1 a x_4.

Dal momento che in un'espressione booleana può esserci, in linea di principio, un numero infinito di simboli, dobbiamo ideare un codice con un alfabeto finito per rappresentare espressioni con un numero arbitrariamente grande di variabili. Solo allora possiamo parlare del SAT come di un "problema", ossia come di un linguaggio su un alfabeto fisso formato dalle codifiche delle espressioni booleane soddisfacibili. Descriviamo il codice di cui faremo uso.

1. I simboli $\wedge$, $\vee$, $\neg$, (, e) sono rappresentati da se stessi.

2. La variabile x_i è rappresentata dal simbolo x seguito dalla rappresentazione binaria di i.

L'alfabeto per il problema/linguaggio SAT ha quindi solo otto simboli. Tutte le istanze di SAT sono stringhe su questo alfabeto finito.

Esempio 10.8 Consideriamo l'espressione $x \wedge \neg(y \vee z)$ tratta dall'Esempio 10.6. Il primo passo da compiere nella codifica è sostituire le variabili con x dotati di indici. Essendoci tre variabili, adoperiamo x_1, x_2 e x_3. Possiamo scegliere liberamente quale fra x, y e z sostituire con ognuna delle x_i. Poniamo $x = x_1$, $y = x_2$ e $z = x_3$. L'espressione diventa dunque $x_1 \wedge \neg(x_2 \vee x_3)$. L'espressione codificata è:

$$x1 \wedge \neg(x10 \vee x11)$$

□

Osserviamo che la lunghezza dell'espressione booleana codificata è pari a circa il numero di posizioni nell'espressione, contando 1 per ogni occorrenza di variabile. La differenza si spiega con il fatto che se l'espressione ha m posizioni, può avere $O(m)$ variabili; le variabili richiedono quindi $O(\log m)$ per la codifica. Quindi un'espressione la cui lunghezza è di m posizioni può avere un codice lungo $n = O(m \log m)$ simboli.

La differenza tra m e $m \log m$ è senz'altro limitata da un polinomio. Perciò, finché trattiamo la questione se un problema possa essere risolto in tempo polinomiale nella lunghezza del suo input, non c'è bisogno di distinguere tra la lunghezza della codifica di un'espressione e il numero di posizioni nell'espressione stessa.

10.2.3 NP-completezza del problema SAT

Dimostriamo ora il teorema di Cook, secondo il quale SAT è NP-completo. Per dimostrare che un problema è NP-completo, dobbiamo in primo luogo provare che è in $\mathcal{NP}$. Dobbiamo poi dimostrare che ogni linguaggio in $\mathcal{NP}$ si riduce al problema in questione.

Per la seconda parte ci serviamo prima di una riduzione polinomiale da un altro problema NP-completo, poi del Teorema 10.5. Ma per ora non conosciamo alcun problema NP-completo da ridurre a SAT. Pertanto l'unica strategia disponibile consiste nel ridurre ogni problema in $\mathcal{NP}$ a SAT.

Teorema 10.9 (Teorema di Cook) SAT è NP-completo.

DIMOSTRAZIONE La prima parte della dimostrazione, cioè provare che SAT è in $\mathcal{NP}$, è facile.

1. Sfruttiamo il non determinismo delle NTM per tentare assegnamenti di valori di verità per l'espressione data E. Se la E codificata ha lunghezza n, allora il tempo $O(n)$ è sufficiente su una NTM multinastro. Osserviamo che la NTM ha diverse scelte e alla fine del tentativo può raggiungere fino a 2^n ID distinte. Ogni diramazione rappresenta il tentativo di un diverso assegnamento di valori di verità.

2. Valutiamo E per l'assegnamento di valori di verità T. Se $E(T) = 1$, accettiamo. Osserviamo che questa parte è deterministica. Il fatto che altre diramazioni della NTM possano non condurre all'accettazione non ha effetto sul risultato: per accettare basta un solo assegnamento soddisfacente di valori di verità.

La valutazione può essere fatta facilmente in tempo $O(n^2)$ su una NTM multinastro. Di conseguenza l'intero riconoscimento di SAT da parte della NTM multinastro richiede tempo $O(n^2)$. La conversione in una NTM a nastro singolo può elevare il tempo al quadrato, per cui un tempo $O(n^4)$ è sufficiente su una NTM a nastro singolo.

Dobbiamo ora dimostrare la parte difficile: se L è un qualsiasi linguaggio in $\mathcal{NP}$, esiste una riduzione polinomiale di L a SAT. Possiamo assumere che esistano una NTM a nastro singolo M e un polinomio $p(n)$ tale che M non impiega più di $p(n)$ passi su un input di lunghezza n, lungo qualunque diramazione. Oltre a ciò, le restrizioni del Teorema 8.12, che abbiamo dimostrato per le DTM, valgono anche per le NTM. Quindi possiamo assumere che M non scriva mai un simbolo blank e non muova mai la testina a sinistra della sua posizione iniziale.

Perciò, se M accetta un input w e $|w| = n$, esiste una sequenza di mosse di M tale che:

1. α_0 è la ID iniziale di M con input w

2. $\alpha_0 \vdash \alpha_1 \vdash \cdots \vdash \alpha_k$, dove $k \leq p(n)$

3. α_k è una ID con uno stato accettante

4. ciascun α_i è formato soltanto da simboli diversi dal blank (tranne il caso in cui α_i finisce con uno stato e un blank) e si estende dalla posizione iniziale della testina (cioè il simbolo di input più a sinistra) verso destra.

La strategia adottata può essere riassunta come segue.

a) Ciascun α_i può essere scritto come una sequenza di simboli $X_{i0}X_{i1}\cdots X_{i,p(n)}$. Uno di questi è uno stato e gli altri sono simboli di nastro. Come sempre assumiamo che gli stati e i simboli di nastro siano disgiunti, per cui possiamo stabilire quale X_{ij} è lo stato, e dunque dove si trova la testina. Osserviamo che non c'è motivo di rappresentare i simboli a destra dei primi $p(n)$ simboli sul nastro (contando lo stato abbiamo così una ID di lunghezza $p(n)+1$) perché questi non possono influenzare una mossa di M se questa si arresta entro $p(n)$ mosse.

b) Descriviamo la sequenza di ID in termini di variabili booleane, creando la variabile y_{ijA} per rappresentare la proposizione $X_{ij} = A$. Qui i e j sono interi nell'intervallo da 0 a $p(n)$ e A è un simbolo di nastro oppure uno stato.

c) Esprimiamo la condizione che la sequenza di ID rappresenta l'accettazione di un input w scrivendo un'espressione booleana soddisfacibile se e solo se M accetta w con una sequenza di non più di $p(n)$ mosse. L'assegnamento soddisfacente è quello che "dice la verità" sulle ID; in altre parole y_{ijA} è vero se e solo se $X_{ij} = A$. Per assicurarci della correttezza della riduzione di $L(M)$ a SAT, scriviamo l'espressione in modo tale che indichi le caratteristiche della computazione.

   i. Avvio corretto. La ID iniziale è q_0w seguito da blank.

   ii. Mossa successiva lecita (cioè la mossa rispetta le regole della TM). Ogni ID successiva segue dalla precedente tramite una delle mosse lecite di M.

   iii. Terminazione corretta. Esiste una ID con stato accettante.

Prima di definire come si costruisce l'espressione booleana, dobbiamo precisare alcuni punti.

- In primo luogo abbiamo specificato che una ID termina dove comincia la coda infinita di blank. Quando si simula una computazione polinomiale è più opportuno considerare tutte le ID come aventi la stessa lunghezza, $p(n)+1$. Perciò in una ID può essere presente una coda di blank.

- In secondo luogo è opportuno assumere che tutte le computazioni durino esattamente $p(n)$ mosse (e abbiano perciò $p(n)+1$ ID) anche se l'accettazione avviene prima. Dunque facciamo sì che ogni ID con uno stato accettante sia il proprio successore. In altre parole, se α ha uno stato accettante, permettiamo la "mossa" $\alpha \vdash \alpha$. Perciò possiamo assumere che, se esiste una computazione accettante, allora $\alpha_{p(n)}$ avrà una ID accettante. Questo è ciò che si deve verificare per la condizione "Terminazione corretta".

La Figura 10.4 illustra una computazione polinomiale di M. Le righe corrispondono alla sequenza di ID; le colonne sono le celle del nastro usato nella computazione. Osserviamo che il numero di celle nella Figura 10.4 è $\big(p(n)+1\big)^2$. Inoltre il numero di variabili che rappresentano ciascuna cella è finito e dipende solo da M; è infatti la somma del numero degli stati e dei simboli di nastro di M.

ID	0	1	...				...	$p(n)$
α_0	X_{00}	X_{01}						$X_{0,p(n)}$
α_1	X_{10}	X_{11}						$X_{1,p(n)}$
α_i				$X_{i,j-1}$	$X_{i,j}$	$X_{i,j+1}$		
α_{i+1}				$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$		
$\alpha_{p(n)}$	$X_{p(n),0}$	$X_{p(n),1}$						$X_{p(n),p(n)}$

Figura 10.4 Costruzione dell'array di celle/ID.

Scriviamo ora un algoritmo per costruire un'espressione booleana $E_{M,w}$ da M e w. La forma complessiva di $E_{M,w}$ è $S \wedge N \wedge F$, dove S, N ed F sono espressioni indicanti che M comincia, si muove e termina correttamente.

Avvio corretto

X_{00} dev'essere lo stato iniziale q_0 di M, i simboli da X_{01} a X_{0n} devono formare w (dove n è la lunghezza di w), e i rimanenti X_{0j} devono essere il blank, B. Quindi se $w = a_1a_2\cdots a_n$:

$$S = y_{00q_0} \wedge y_{01a_1} \wedge y_{02a_2} \wedge \cdots \wedge y_{0na_n} \wedge y_{0,n+1,B} \wedge y_{0,n+2,B} \wedge \cdots \wedge y_{0,p(n),B}$$

Data la codifica di M e dato w, possiamo senz'altro scrivere S in un tempo $O(p(n))$ sul secondo nastro di una TM multinastro.

Terminazione corretta

Avendo ipotizzato che una ID accettante si ripeta per sempre, l'accettazione da parte di M equivale a trovare uno stato accettante in $\alpha_{p(n)}$. Ricordiamo che per ipotesi la NTM M, se accetta, lo fa entro $p(n)$ passi. Di conseguenza F è l'OR delle espressioni F_j, per $j = 0, 1, \ldots, p(n)$, dove F_j afferma che $X_{p(n),j}$ è uno stato accettante. In altre parole F_j è $y_{p(n),j,a_1} \vee y_{p(n),j,a_2} \vee \cdots \vee y_{p(n),j,a_k}$, dove $a_1, a_2, \ldots, a_k$ sono tutti gli stati accettanti di M. Quindi

$$F = F_0 \vee F_1 \vee \cdots \vee F_{p(n)}$$

Osserviamo che ogni F_i impiega un numero costante di simboli, che dipende da M, ma non dalla lunghezza n dell'input w. Perciò F ha lunghezza $O(p(n))$. Inoltre il tempo per scrivere F, data una codifica di M e l'input w, è polinomiale in n; in realtà F può essere scritta in un tempo $O(p(n))$ su una TM multinastro.

Mossa successiva lecita

Assicurarsi che le mosse di M siano corrette è la parte di gran lunga più complicata. L'espressione N sarà l'AND delle espressioni N_i, per $i = 0, 1, \ldots, p(n) - 1$, e ogni N_i sarà definita in modo da garantire che α_{i+1} sia una delle ID che possono seguire α_i in M. Per spiegare come si scrive N_i, osserviamo anzitutto il simbolo $X_{i+1,j}$ nella Fig. 10.4. È sempre possibile determinare $X_{i+1,j}$ dai seguenti dati.

1. I tre simboli sopra di esso: $X_{i,j-1}$, X_{ij} e $X_{i,j+1}$.
2. La mossa scelta da M se uno dei simboli è lo stato di α_i.

Scriveremo N_i come l'$\wedge$ delle espressioni $A_{ij} \vee B_{ij}$, dove $j = 0, 1, \ldots, p(n)$.

- L'espressione A_{ij} indica che:
  a) lo stato di α_i è alla posizione j (cioè X_{ij} è lo stato)
  b) M può scegliere una mossa, dove X_{ij} è lo stato e $X_{i,j+1}$ è il simbolo guardato, che trasforma la sequenza di simboli $X_{i,j-1}X_{ij}X_{i,j+1}$ nella sequenza $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$. Osserviamo che, se X_{ij} è uno stato accettante, esiste la "scelta" di non fare alcuna mossa, per cui tutte le ID successive coincidono con quella che per prima ha condotto all'accettazione.
- L'espressione B_{ij} indica che:

a) lo stato di α_i non è nella posizione j (cioè X_{ij} non è uno stato);

b) se lo stato di α_i non è adiacente alla posizione j (cioè $X_{i,j-1}$ e $X_{i,j+1}$ non sono degli stati), allora $X_{i+1,j} = X_{ij}$.

Quando lo stato è adiacente alla posizione j, la correttezza della posizione j è garantita da $A_{i,j-1}$ o da $A_{i,j+1}$.

B_{ij} è la più facile da scrivere. Siano $q_1, q_2, \ldots, q_m$ gli stati di M, e siano $Z_1, Z_2, \ldots, Z_r$ i simboli di nastro. Allora:

$$\begin{aligned} B_{ij} \quad = \quad & (y_{i,j-1,q_1} \vee y_{i,j-1,q_2} \vee \cdots \vee y_{i,j-1,q_m}) \vee \\ & (y_{i,j+1,q_1} \vee y_{i,j+1,q_2} \vee \cdots \vee y_{i,j+1,q_m}) \vee \\ & \big((y_{i,j,Z_1} \vee y_{i,j,Z_2} \vee \cdots \vee y_{i,j,Z_r}) \wedge \\ & \big((y_{i,j,Z_1} \wedge y_{i+1,j,Z_1}) \vee (y_{i,j,Z_2} \wedge y_{i+1,j,Z_2}) \vee \cdots \vee (y_{i,j,Z_r} \wedge y_{i+1,j,Z_r})\big)\big) \end{aligned}$$

Le prime due righe garantiscono che B_{ij} è vera quando lo stato di α_i è adiacente alla posizione j. Le prime tre righe insieme garantiscono che, se lo stato di α_i è alla posizione j, allora B_{ij} è falsa, e quindi N_i è vera solo se A_{ij} è vera, cioè se la mossa è lecita. Se lo stato dista almeno due posizioni da j, le due ultime righe indicano che il simbolo non deve cambiare. L'ultima riga afferma che $X_{ij} = X_{i+1,j}$ specificando che entrambi devono essere Z_1 o Z_2, e così via.

Esistono due importanti casi particolari: $j = 0$ e $j = p(n)$. Nel primo caso non ci sono variabili $y_{i,j-1,X}$, nell'altro non ci sono variabili $y_{i,j+1,X}$. Sappiamo però che la testina non si muove mai a sinistra della sua posizione iniziale, e sappiamo che non ha il tempo di raggiungere più di $p(n)$ celle a destra del punto da cui è partita. Di conseguenza possiamo eliminare certi termini da B_{i0} e $B_{i,p(n)}$. Lasciamo la semplificazione al lettore.

Consideriamo ora le espressioni A_{ij}, che rispecchiano le possibili relazioni tra gli elementi del rettangolo 2×3 di simboli nell'array della Figura 10.4: $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$, $X_{i+1,j-1}$, $X_{i+1,j}$ e $X_{i+1,j+1}$. Diremo che un assegnamento di simboli a queste variabili è *valido* se:

1. X_{ij} è uno stato, ma $X_{i,j-1}$ e $X_{i,j+1}$ sono simboli di nastro

2. c'è una mossa di M per effetto della quale $X_{i,j-1}X_{ij}X_{i,j+1}$ diventa

$$X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$$

Il numero di assegnamenti validi alle sei variabili è quindi finito. Sia A_{ij} un OR di termini, un termine per ogni insieme delle sei variabili che forma un assegnamento valido.

Per esempio supponiamo che una mossa di M derivi dal fatto che $\delta(q, A)$ contiene (p, C, L). Sia D un simbolo di nastro di M. Un assegnamento valido sarà dunque

$X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ e $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = pDC$. Osserviamo che l'assegnamento riflette il cambio di ID causato dalla mossa di M. Il termine che esprime questa possibilità è

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,p} \wedge y_{i+1,j,D} \wedge y_{i+1,j+1,C}$$

Se invece $\delta(q, A)$ contiene (p, C, R) (la mossa è la stessa, ma la testina si muove verso destra), l'assegnamento valido corrispondente è formato da $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ e $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = DCp$. Il termine per questo assegnamento valido è

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,D} \wedge y_{i+1,j,C} \wedge y_{i+1,j+1,p}$$

A_{ij} è l'OR di tutti i termini validi. Nei casi speciali $j = 0$ e $j = p(n)$ bisogna apportare alcune modifiche per tener conto che le variabili y_{ijZ} non esistono per $j < 0$ o $j > p(n)$, come abbiamo fatto per B_{ij}. Infine

$$N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \cdots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$$

e dunque

$$N = N_0 \wedge N_1 \wedge \cdots \wedge N_{p(n)-1}$$

Se M ha molti stati o simboli di nastro, A_{ij} e B_{ij} possono essere molto grandi, ma la loro dimensione è comunque una costante rispetto alla lunghezza dell'input w; in altre parole la loro dimensione è indipendente da n, la lunghezza di w. Di conseguenza la lunghezza di N_i è $O(p(n))$ e la lunghezza di N è $O(p^2(n))$. L'aspetto più importante è che possiamo scrivere N su un nastro di una TM multinastro in un tempo proporzionale alla sua lunghezza e polinomiale in n, la lunghezza di w.

Dimostrazione del teorema di Cook: conclusione

Abbiamo descritto la costruzione di

$$E_{M,w} = S \wedge N \wedge F$$

come una funzione sia di M sia di w, ma solo S (la parte "avvio corretto") dipende da w, e solo in modo diretto (w è sul nastro della ID iniziale). Le altre parti, N ed F, dipendono solo da M e da n, la lunghezza di w.

Quindi per ogni NTM M che gira in un tempo polinomiale $p(n)$ possiamo ideare un algoritmo che prende un input w di lunghezza n e produce $E_{M,w}$. Il tempo di esecuzione dell'algoritmo su una TM deterministica multinastro è $O(p^2(n))$, e possiamo convertire la TM in una TM a nastro singolo che impiega un tempo $O(p^4(n))$. L'output dell'algoritmo è un'espressione booleana $E_{M,w}$ soddisfacibile se e solo se M accetta w entro $p(n)$ mosse. □

Per sottolineare l'importanza del Teorema di Cook 10.9, vediamo come applicargli il Teorema 10.5. Supponiamo che esista una TM deterministica in grado di riconoscere le istanze di SAT in tempo polinomiale, poniamo $q(n)$. Allora ogni linguaggio accettato da una NTM M in tempo polinomiale $p(n)$ sarebbe accettato in tempo polinomiale dalla DTM schematizzata nella Figura 10.5. L'input w di M viene convertito in un'espressione booleana $E_{M,w}$, che viene passata al verificatore di SAT. Il nuovo algoritmo dà per w la stessa risposta del verificatore su $E_{M,w}$.

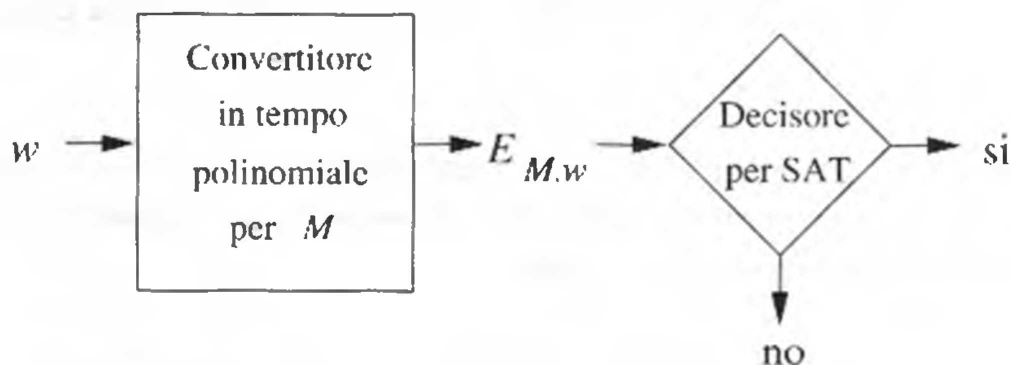


Figura 10.5 Se SAT è in $\mathcal{P}$, si può dimostrare che ogni linguaggio in $\mathcal{NP}$ si trova in $\mathcal{P}$ tramite una DTM definita in questo modo.

10.2.4 Esercizi

Esercizio 10.2.1 Quanti assegnamenti di valori di verità soddisfacenti hanno le seguenti espressioni booleane? Quali sono in SAT?

* a) $x \wedge (y \vee \neg x) \wedge (z \vee \neg y)$.

  b) $(x \vee y) \wedge \big(\neg(x \vee z) \vee (\neg z \wedge \neg y)\big)$.

Esercizio 10.2.2 Sia G un grafo di quattro nodi:1, 2, 3 e 4. Sia x_{ij}, per $1 \leq i < j \leq 4$, una variabile proposizionale la cui interpretazione è "esiste un lato tra i nodi i e j". Qualsiasi grafo su questi nodi può essere rappresentato da un assegnamento di valori di verità. Per esempio il grafo della Figura 10.1 è rappresentato dichiarando falsa x_{14} e vere le altre cinque variabili. Qualsiasi proprietà del grafo che riguardi solo l'esistenza o la non esistenza di lati si può esprimere come un'espressione booleana vera se e solo se l'assegnamento di valori di verità alle variabili descrive un grafo con quella proprietà. Scrivete le espressioni per le seguenti proprietà.

* a) G ha un circuito hamiltoniano.

  b) G è connesso.

c) G contiene una *clique* di dimensione 3, ossia un insieme di tre nodi tale che ogni coppia di quei nodi è unita da un lato (cioè un triangolo nel grafo).

d) G contiene almeno un nodo isolato, cioè un nodo senza lati adiacenti.

10.3 Un problema di soddisfacibilità vincolato

Intendiamo dimostrare l'NP-completezza di un'ampia gamma di problemi, tra cui il TSP, citato nel Paragrafo 10.1.4. In teoria procediamo per riduzione polinomiale dal problema SAT al problema in esame. Esiste però un importante problema intermedio, detto "3SAT", molto più facile da ridurre ai problemi tipici rispetto a SAT. Anche 3SAT è un problema di soddisfacibilità di espressioni booleane, ma per espressioni di forma molto regolare, formate dalla congiunzione logica di clausole, ognuna delle quali è la disgiunzione logica di esattamente tre variabili, eventualmente negate.

In questo paragrafo presentiamo la terminologia relativa alle espressioni booleane. Riduciamo poi la soddisfacibilità di un'espressione qualsiasi alla soddisfacibilità di espressioni nella forma di 3SAT. È interessante osservare che, se ogni espressione booleana E ha un'espressione equivalente F nella forma normale di 3SAT, la dimensione di F può essere esponenziale in quella di E. Una riduzione in tempo polinomiale di SAT a 3SAT deve quindi essere più "scaltra" di una semplice manipolazione algebrica. Dobbiamo convertire ogni espressione E di SAT in un'espressione F nella forma normale di 3SAT. Non occorre che F sia equivalente a E: basta che F sia soddisfacibile se e solo se E lo è.

10.3.1 Forme normali di espressioni booleane

Diamo tre definizioni fondamentali.

- Un *letterale* è una variabile o una variabile negata. Ne sono esempio x e $\neg y$. Per risparmiare spazio scriveremo spesso $\overline{y}$ in luogo di $\neg y$.

- Una *clausola* è la disgiunzione logica (OR) di uno o più letterali. Ne sono esempio x, $x \vee y$, e $x \vee \overline{y} \vee z$.

- Un'espressione booleana si dice in *forma normale congiuntiva*, o CNF (*Conjunctive Normal Form*), se è la congiunzione logica (AND) di una o più clausole.

Per abbreviare ulteriormente le espressioni adottiamo una notazione alternativa, in cui trattiamo $\vee$ come una somma, e scriviamo $+$, e $\wedge$ come un prodotto. Per il prodotto ricorriamo di solito alla giustapposizione, senza operatore, come per la concatenazione nelle espressioni regolari. Conformemente diremo che una clausola è una "somma di letterali" e un'espressione in CNF un "prodotto di clausole".

Esempio 10.10 Nella notazione compressa l'espressione $(x \lor \neg y) \land (\neg x \lor z)$ sarà scritta $(x+\overline{y})(\overline{x}+z)$. L'espressione è in forma normale congiuntiva perché è il prodotto (AND) delle clausole $(x+\overline{y})$ e $(\overline{x}+z)$.

L'espressione $(x+y\overline{z})(x+y+z)(\overline{y}+\overline{z})$ non è in CNF. È bensì il prodotto di tre sottoespressioni: $(x+y\overline{z})$, $(x+y+z)$ e $(\overline{y}+\overline{z})$, ma solo le ultime due sono clausole. La prima è la somma di un letterale e del prodotto di due letterali.

L'espressione xyz è in CNF. Ricordiamo che una clausola può contenere anche un solo letterale. L'espressione in esame è quindi il prodotto di tre clausole: (x), (y) e (z). □

Un'espressione si dice in *forma normale k-congiuntiva* (k-CNF) se è il prodotto di clausole, ognuna delle quali è la somma di k letterali distinti. Per esempio $(x+\overline{y})(y+\overline{z})(z+\overline{x})$ è in 2-CNF perché ognuna delle sue clausole ha esattamente due letterali.

Ogni vincolo sulle espressioni booleane dà luogo a un problema di soddisfacibilità delle espressioni che lo soddisfano. Affronteremo quindi i due problemi seguenti.

- CSAT: una data espressione booleana in CNF è soddisfacibile?
- kSAT: una data espressione booleana in k-CNF è soddisfacibile?

Vedremo che CSAT, 3SAT e kSAT per ogni k maggiore di 3 sono NP-completi. Esistono invece algoritmi in tempo lineare per 1SAT e 2SAT.

10.3.2 Conversione in CNF di espressioni booleane

Due espressioni booleane si dicono *equivalenti* se danno lo stesso risultato per ogni assegnamento di valori di verità alle variabili. Ovviamente, se due espressioni sono equivalenti, o sono entrambe soddisfacibili o nessuna delle due lo è. Perciò un metodo promettente per ricavare una riduzione polinomiale da SAT a CSAT consiste nel convertire espressioni arbitrarie in espressioni equivalenti in CNF. Questa riduzione dimostrerebbe che CSAT è NP-completo.

La questione non è però semplicissima. È vero che possiamo convertire in CNF qualsiasi espressione, ma la riduzione può richiedere un tempo più che polinomiale. In particolare può far crescere esponenzialmente la lunghezza dell'espressione, e quindi richiedere tempo esponenziale per generare il risultato.

Per fortuna convertire un'espressione booleana arbitraria in un'espressione in CNF è solo uno dei modi di ridurre SAT a CSAT, e così provare che CSAT è NP-completo. In realtà è sufficiente convertire un'istanza E di SAT in un'istanza F di CSAT in modo che F sia soddisfacibile se e solo se E lo è. Non è necessario che E ed F siano equivalenti. Non è neppure necessario che E ed F abbiano le stesse variabili; in generale, anzi, le variabili di E saranno un sottoinsieme di quelle di F.

Trattamento di input malformati

Tutti i linguaggi di cui abbiamo parlato – SAT, CSAT, 3SAT, e così via – sono linguaggi su un alfabeto fissato, di 8 simboli, le cui stringhe si possono a volte interpretare come espressioni booleane. Una stringa che non possa essere interpretata come espressione non può essere nel linguaggio di SAT. In modo simile, se consideriamo espressioni di forma vincolata, una stringa che sia un'espressione booleana ben formata, ma non della forma richiesta, non è nel linguaggio. Un algoritmo di decisione per il problema CSAT, per esempio, risponde "no" a fronte di un'espressione booleana soddisfacibile ma non in CNF.

La riduzione di SAT a CSAT si articola in due tempi. Dapprima tutti i $\neg$ vengono spinti verso il basso dell'albero dell'espressione finché le negazioni si applicano solo a singole variabili. L'espressione booleana diventa un AND e OR di letterali. Questa trasformazione produce un'espressione equivalente e richiede un tempo al più quadratico nella lunghezza dell'espressione. Un programma concreto, con strutture dati ben congegnate, impiega un tempo lineare.

In un secondo tempo scriviamo un'espressione composta di AND e OR di letterali come prodotto di clausole, cioè in CNF. Ricorrendo a nuove variabili possiamo compiere la trasformazione in un tempo polinomiale nella lunghezza dell'espressione data. In generale la nuova espressione F non è equivalente a quella originale E, ma è soddisfacibile se e solo se lo è E. Più esattamente, se T è un assegnamento di valori di verità che rende E vera, esiste un'*estensione* S di T che rende F vera. Diciamo che S è un'estensione di T se assegna gli stessi valori di T alle variabili di T: S può assegnare un valore anche a variabili assenti da T.

Il primo passo consiste nello spingere i $\neg$ sotto gli $\wedge$ e gli $\vee$. Ci servono tre regole.

1. $\neg(E \wedge F) \Rightarrow \neg(E) \vee \neg(F)$. Questa regola, una delle *leggi di DeMorgan*, permette di spostare i $\neg$ sotto gli $\wedge$, e ha l'effetto di mutare un $\wedge$ in $\vee$.

2. $\neg(E \vee F) \Rightarrow \neg(E) \wedge \neg(F)$. L'altra *legge di DeMorgan* sposta il $\neg$ sotto l'$\vee$, con l'effetto di mutare l'$\vee$ in $\wedge$.

3. $\neg\big(\neg(E)\big) \Rightarrow E$. La *legge della doppia negazione* cancella una coppia di $\neg$ applicati alla stessa espressione.

Esempio 10.11 Consideriamo l'espressione $E = \neg\Big(\big(\neg(x+y)\big)(\overline{x}+y)\Big)$. Abbiamo qui mescolato le due notazioni, con l'operatore $\neg$ esplicito quando l'espressione da negare

Espressione	Regola
$\neg\Big(\big(\neg(x+y)\big)(\overline{x}+y)\Big)$	
$\neg\big(\neg(x+y)\big)+\neg(\overline{x}+y)$	(1)
$x+y+\neg(\overline{x}+y)$	(3)
$x+y+\big(\neg(\overline{x})\big)\overline{y}$	(2)
$x+y+x\overline{y}$	(3)

Figura 10.6 Spostamento dei $\neg$ verso i letterali in fondo all'albero di un'espressione.

non è una semplice variabile. La Figura 10.6 illustra i passi che spostano in basso i $\neg$ dell'espressione E fino ad applicarli direttamente ai letterali.

L'ultima espressione, equivalente all'originale, è una combinazione di letterali con AND e OR. Potremmo semplificarla ancora e ridurla a $x + y$, ma questo non incide sull'ipotesi che ogni espressione si può riscrivere in modo che i $\neg$ figurino solo nei letterali. □

Teorema 10.12 Per ogni espressione booleana E esiste un'espressione equivalente F in cui le negazioni compaiono solo nei letterali, cioè si applicano direttamente alle variabili. Inoltre la lunghezza di F è lineare nel numero di simboli di E, ed F si costruisce da E in tempo polinomiale.

DIMOSTRAZIONE La dimostrazione è un'induzione sul numero di operatori ($\wedge$, $\vee$ e $\neg$) in E. Dimostriamo l'esistenza di un'espressione equivalente F con tutti i $\neg$ nei letterali. Se E contiene $n \geq 1$ operatori, F ne contiene al più $2n - 1$.

Poiché F si può scrivere usando non più di una coppia di parentesi per operatore, e il numero delle variabili di un'espressione può superare il numero di operatori al massimo di uno, concludiamo che la lunghezza di F è linearmente proporzionale a quella di E. Vedremo inoltre che la costruzione di F è tanto semplice da richiedere tempo proporzionale alla sua lunghezza, e quindi proporzionale alla lunghezza di E.

BASE Se contiene un solo operatore, E dev'essere della forma $\neg x$, $x \vee y$ o $x \wedge y$, con x e y variabili. In ogni caso E è già della forma desiderata e possiamo porre $F = E$. Sia E sia F hanno un operatore, quindi vale la relazione "il numero di operatori di F è al più doppio di quello di E, meno 1".

INDUZIONE Supponiamo che l'enunciato sia vero per tutte le espressioni con meno operatori di E. Se il suo operatore di livello più alto non è $\neg$, E dev'essere della forma $E_1 \vee E_2$ o $E_1 \wedge E_2$. In entrambi i casi l'ipotesi induttiva si applica a E_1 e a E_2; essa dice che esistono due espressioni equivalenti, rispettivamente F_1 ed F_2, in cui i $\neg$ figurano solo nei letterali. Quindi $F = F_1 \vee F_2$ o $F = (F_1) \wedge (F_2)$ è equivalente a E. Supponiamo che

E_1 ed E_2 abbiano rispettivamente a e b operatori. Allora E ne ha $a+b+1$. Per l'ipotesi induttiva F_1 ne ha al massimo $2a-1$ ed F_2 al massimo $2b-1$. Perciò F ha al massimo $2a+2b-1$ operatori, cioè non più di $2(a+b+1)-1$, ossia il doppio di E, meno 1.

Consideriamo ora il caso in cui E ha forma $\neg E_1$. A seconda dell'operatore più esterno di E_1 distinguiamo tre casi. Osserviamo che E_1 deve contenere un operatore; in caso contrario E rientrerebbe nei casi di base.

1. $E_1 = \neg E_2$. Per la legge della doppia negazione, $E = \neg(\neg E_2)$ equivale a E_2. Poiché E_2 ha meno operatori di E, si applica l'ipotesi di induzione. Possiamo quindi trovare una F equivalente a E_2, in cui i $\neg$ figurano solo nei letterali. La stessa F vale per E. Poiché il numero di operatori di F non supera il doppio del numero di operatori di E_2, meno 1, non può superare neppure il doppio di quello di E, diminuito di 1.

2. $E_1 = E_2 \vee E_3$. Per la legge di DeMorgan, $E = \neg(E_2 \vee E_3)$ equivale a $(\neg(E_2)) \wedge (\neg(E_3))$. Sia $\neg(E_2)$ sia $\neg(E_3)$ hanno meno operatori di E; per l'ipotesi induttiva esistono espressioni equivalenti F_2 ed F_3 con tutti i $\neg$ nei letterali. Quindi $F = (F_2) \wedge (F_3)$ serve da equivalente di E. Affermiamo inoltre che il numero di operatori di F non è "troppo elevato". Sia a il numero di operatori di E_2 e b quello di E_3. Allora E ha $a+b+2$ operatori. Poiché $\neg(E_2)$ e $\neg(E_3)$ hanno rispettivamente $a+1$ e $b+1$ operatori, ed F_2 ed F_3 sono ricavati da quelle espressioni, per ipotesi di induzione sappiamo che F_2 ed F_3 hanno al più rispettivamente $2(a+1)-1$ e $2(b+1)-1$ operatori. Quindi F ha non più di $2a+2b+3$ operatori. Questo numero è esattamente il doppio del numero di operatori di E, meno 1.

3. $E_1 = E_2 \wedge E_3$. In questo caso si procede come in (2) applicando la seconda legge di DeMorgan.

□

10.3.3 NP-completezza di CSAT

Dobbiamo ora convertire in CNF un'espressione E formata da AND e OR di letterali. Come già detto, per produrre in tempo polinomiale, a partire da E, un'espressione F soddisfacibile se e solo se E è soddisfacibile, dobbiamo rinunciare alle trasformazioni che preservano l'equivalenza e introdurre in F variabili che non figurano in E. Ci serviremo di questa tecnica per dimostrare che CSAT è NP-completo; per chiarire la costruzione, ne daremo poi un esempio.

Teorema 10.13 CSAT è NP-completo.

Descrivere algoritmi

Formalmente il tempo di esecuzione di una riduzione è quello necessario a eseguirla su una macchina di Turing mononastro, ma gli algoritmi di questo tipo sono inutilmente complessi. Sappiamo che i problemi risolvibili in tempo polinomiale su computer ordinari, su TM multinastro, o su TM mononastro sono gli stessi, anche se il grado dei polinomi può variare. Perciò, nel descrivere algoritmi sofisticati per ridurre un problema NP-completo a un altro, stabiliamo di misurare i tempi rispetto a implementazioni efficienti su un normale computer. In questo modo possiamo tralasciare dettagli sul trattamento dei nastri per mettere in luce le idee algoritmiche più importanti.

DIMOSTRAZIONE Spieghiamo come ridurre SAT a CSAT in tempo polinomiale. Applichiamo anzitutto il metodo del Teorema 10.12 per convertire un'istanza di SAT in un'espressione E con tutti i $\neg$ nei letterali. Spieghiamo poi come trasformare E in un'espressione F in CNF in tempo polinomiale e provare che F è soddisfacibile se e solo se lo è E. F si costruisce per induzione sulla lunghezza di E e soddisfa una proprietà più forte di quella che ci serve. Dimostriamo, per induzione sul numero di simboli presenti in E (la sua "lunghezza"), l'enunciato seguente.

- Esiste una costante c tale che, se E è un'espressione booleana di lunghezza n in cui i $\neg$ compaiono solo nei letterali, c'è un'espressione F con le seguenti proprietà.

  a) F è in CNF ed è formata da non più di n clausole.

  b) F si può costruire da E in tempo non superiore a $c|E|^2$.

  c) Un assegnamento di valori di verità T per E rende E vera se e solo se c'è un'estensione S di T che rende F vera.

BASE Se E consiste di uno o due simboli, è un letterale. Ma un letterale è una clausola, dunque E è già in CNF.

INDUZIONE Assumiamo che ogni espressione più breve di E si possa convertire in un prodotto di clausole in tempo non superiore a cn^2 per un'espressione di lunghezza n. A seconda dell'operatore principale di E distinguiamo due casi.

Caso 1: $E = E_1 \wedge E_2$. Per l'ipotesi di induzione esistono due espressioni in CNF, F_1 ed F_2, derivate rispettivamente da E_1 ed E_2. Tutti e soli gli assegnamenti soddisfacenti per E_1 si possono estendere ad assegnamenti soddisfacenti per F_1, e lo stesso vale per E_2 ed F_2. Senza perdere in generalità, assumiamo che le variabili di F_1 e quelle di F_2 siano

disgiunte, tranne che per le variabili che figurano in E. Se dobbiamo introdurre nuove variabili in F_1 o in F_2, le scegliamo diverse dalle altre.

Sia $F = F_1 \wedge F_2$. $F_1 \wedge F_2$ è chiaramente un'espressione in CNF se lo sono F_1 ed F_2. Dobbiamo dimostrare che un assegnamento di valori di verità T per E si può estendere a un assegnamento soddisfacente per F se e solo se T soddisfa E.

(Se) Supponiamo che T soddisfi E. Sia T_1 come T, ma limitato alle variabili presenti in E_1, e sia T_2 la stessa cosa riferita a E_2. Per ipotesi di induzione, T_1 e T_2 si possono estendere ad assegnamenti S_1 ed S_2 che soddisfano rispettivamente F_1 ed F_2. Definiamo S in accordo con S_1 ed S_2 sulle rispettive variabili. Poiché le sole variabili comuni a F_1 ed F_2 sono quelle di E, ed S_1 ed S_2 devono coincidere su di esse se sono entrambe definite, è sempre possibile costruire S. Dunque S è un'estensione di T che soddisfa F.

(Solo se) Nell'altra direzione, sia S un'estensione di T che soddisfa F. Sia T_1 (rispettivamente T_2) come T, ma limitato alle variabili di E_1 (risp. E_2). Con S_1 (risp. S_2) denotiamo S ristretto alle variabili di F_1 (risp. F_2). Allora S_1 è un'estensione di T_1 ed S_2 un'estensione di T_2. Poiché F è l'AND di F_1 ed F_2, necessariamente S_1 soddisfa F_1, ed S_2 soddisfa F_2. Per ipotesi di induzione, T_1 (risp. T_2) soddisfa E_1 (risp. E_2). Dunque T soddisfa E.

Caso 2: $E = E_1 \vee E_2$. Come nel caso 1, ricorriamo all'ipotesi di induzione per affermare l'esistenza di due espressioni CNF, F_1 ed F_2, con le seguenti proprietà.

1. Un assegnamento di valori di verità per E_1 (risp. E_2) soddisfa E_1 (risp. E_2) se e solo se si può estendere a un assegnamento che soddisfa F_1 (risp. F_2).

2. Le variabili di F_1 ed F_2 sono disgiunte, tranne quelle presenti in E.

3. F_1 ed F_2 sono in CNF.

Per costruire F non possiamo fare semplicemente l'OR di F_1 ed F_2 perché l'esito non sarebbe in CNF. Ricorriamo dunque a una costruzione più complicata, in cui sfruttiamo il fatto che vogliamo solo preservare la soddisfacibilità senza chiedere l'equivalenza. Supponiamo

$$F_1 = g_1 \wedge g_2 \wedge \cdots \wedge g_p$$

ed $F_2 = h_1 \wedge h_2 \wedge \cdots \wedge h_q$, dove le g e le h sono clausole. Introduciamo una nuova variabile y. Sia

$$F = (y + g_1) \wedge (y + g_2) \wedge \cdots \wedge (y + g_p) \wedge (\overline{y} + h_1) \wedge (\overline{y} + h_2) \wedge \cdots \wedge (\overline{y} + h_q)$$

Dobbiamo dimostrare che un assegnamento di valori di verità T per E soddisfa E se e solo se T si può estendere a un assegnamento S che soddisfa F.

(Solo se) Supponiamo che T soddisfi E. Come nel caso 1, sia T_1 (risp. T_2) come T, ma limitato alle variabili di E_1 (risp. E_2). Poiché $E = E_1 \vee E_2$, T soddisfa E_1 o T soddisfa E_2. Supponiamo che soddisfi E_1. Allora T_1, cioè T ristretto alle variabili di E_1, si può estendere a S_1, che soddisfa F_1. Costruiamo S, estensione di T, in modo che soddisfi l'espressione F definita sopra.

1. Per ogni variabile x in F_1, $S(x) = S_1(x)$.
2. $S(y) = 0$. Questa scelta rende vere tutte le clausole di F derivate da F_2.
3. Per ogni variabile x presente in F_2 ma non in F_1, $S(x)$ coincide con $T(x)$ se questo è definito; altrimenti può valere arbitrariamente 0 o 1.

Per la regola 1, S rende vere tutte le clausole derivate dalle g. Per la regola 2, cioè l'assegnamento per y, S rende vere tutte le clausole derivate dalle h. Quindi S soddisfa F.

Se T non soddisfa E_1, ma soddisfa E_2, si procede allo stesso modo, ponendo $S(y) = 1$ nella regola 2. Inoltre $S(x)$ deve coincidere con $S_2(x)$ dove questa è definita, mentre la scelta di $S(x)$ per variabili presenti solo in S_1 è arbitraria. Concludiamo che anche in questo caso S soddisfa F.

(Se) Supponiamo di estendere l'assegnamento di valori di verità T per E a un assegnamento S per F, e che S soddisfi F. Secondo il valore di verità assegnato a y, ci sono due casi. Poniamo anzitutto $S(y) = 0$. In questo caso tutte le clausole di F derivate dalle h sono vere. D'altra parte y non è d'aiuto per le clausole della forma $(y + g_i)$, e quindi S deve rendere vere tutte le g_i. In sostanza S rende vera F_1.

Più esattamente, sia S_1 come S, ma ristretto alle variabili di F_1. Allora S_1 soddisfa F_1. Per l'ipotesi di induzione T_1, cioè T ristretto alle variabili di E_1, deve soddisfare E_1. Infatti S_1 estende T_1, e T deve soddisfare E, cioè $E_1 \vee E_2$, perché T_1 soddisfa F_1.

Dobbiamo trattare anche il caso $S(y) = 1$, che però è simmetrico a quello appena svolto, ed è quindi lasciato al lettore. Concludiamo che T soddisfa E quando S soddisfa F.

Dobbiamo ora dimostrare che il tempo necessario per costruire F da E è al massimo quadratico in n, la lunghezza di E. In qualunque caso, sia la separazione di E in E_1 ed E_2 sia la costruzione di F da F_1 ed F_2 richiedono un tempo lineare rispetto alla dimensione di E. Sia dn un limite superiore al tempo richiesto per costruire E_1 ed E_2 da E sommato al tempo per costruire F da F_1 ed F_2, nel caso 1 o nel caso 2. Il tempo $T(n)$ richiesto per costruire F da un'espressione E di lunghezza n è quindi dato da un'equazione di ricorrenza:

$$T(1) = T(2) \leq e \text{ per una costante } e$$
$$T(n) \leq dn + c\max_{0<i<n-1}\big(T(i) + T(n-1-i)\big) \text{ per } n \geq 3$$

Dobbiamo ancora determinare il valore della costante c che consente di provare $T(n) \leq cn^2$. Le clausole di base per $T(1)$ e per $T(2)$ corrispondono ai casi in cui E consiste in uno o due simboli, cioè in un un letterale. Non si ha quindi ricorsione, e la procedura richiede tempo pari a e. La clausola ricorsiva sfrutta il fatto che E è composta da E_1 ed E_2, legate dall'operatore $\wedge$ o da $\vee$, e lunghe rispettivamente i e $n - i - 1$. Nel complesso la conversione di E in F comporta due passi: scomporre E in E_1 ed E_2, e trasformare E_1 ed E_2 in F. Sappiamo che ognuno di questi richiede al massimo un tempo dn, più le trasformazioni ricorsive di E_1 in F_1 e di E_2 in F_2.

Dobbiamo dimostrare per induzione su n l'esistenza di una costante c tale che, per ogni n, $T(n) \leq cn^2$.

BASE Per $n = 1$ scegliamo c maggiore o uguale a e.

INDUZIONE Supponiamo vero l'enunciato per lunghezze inferiori a n. Allora $T(i) \leq ci^2$ e $T(n - i - 1) \leq c(n - i - 1)^2$. Quindi

$$T(i) + T(n - i - 1) \leq n^2 - 2i(n - i) - 2(n - i) + 1 \tag{10.1}$$

Poiché $n \geq 3$ e $0 < i < n - 1$, $2i(n - i)$ è maggiore o uguale a n, e $2(n - i)$ è maggiore o uguale a 2. Quindi il membro destro di (10.1) è minore di $n^2 - n$ per ogni valore ammissibile di i. La parte ricorsiva della definizione di $T(n)$ afferma allora che $T(n) \leq dn + cn^2 - cn$. Scegliendo $c \geq d$ deduciamo che $T(n) \leq cn^2$ è vera per n, e questo completa l'induzione. Perciò la costruzione di F da E richiede tempo $O(n^2)$. □

Esempio 10.14 Applichiamo la costruzione del Teorema 10.13 a un'espressione semplice: $E = x\overline{y} + \overline{x}(y + z)$. L'albero sintattico dell'espressione è illustrato nella Figura 10.7. Presso ogni nodo è riportata l'espressione in CNF associata a quella rappresentata dal nodo.

Le foglie corrispondono ai letterali; per ogni letterale l'espressione in CNF è una clausola formata dal letterale stesso. Per esempio l'espressione in CNF associata alla foglia contrassegnata da $\overline{y}$ è $(\overline{y})$. Le parentesi sono superflue; le manteniamo nelle espressioni in CNF per ricordare che trattiamo con prodotti di clausole.

Per un nodo AND la costruzione dell'espressione in CNF consiste nel formare il prodotto (AND) di tutte le clausole delle due sottoespressioni. Quindi, per esempio, l'espressione in CNF associata al nodo della sottoespressione $\overline{x}(y + z)$ è il prodotto di una clausola per $\overline{x}$, cioè $(\overline{x})$, e delle due clausole per $y + z$, cioè $(v + y)(\overline{v} + z)$.[3]

Per un nodo OR dobbiamo introdurre una nuova variabile. La aggiungiamo a tutte le clausole dell'operando sinistro, e ne aggiungiamo la negazione alle clausole dell'operando destro. Consideriamo per esempio il nodo radice nella Figura 10.7. Esso è l'OR delle

[3]In questo caso particolare, in cui la sottoespressione $y + z$ è di per sé una clausola, non è necessario compiere la costruzione generale per l'OR di espressioni, e avremmo potuto scegliere $(y + z)$ come prodotto di clausole equivalente a $y + z$. In questo esempio ci atteniamo alla regola generale.

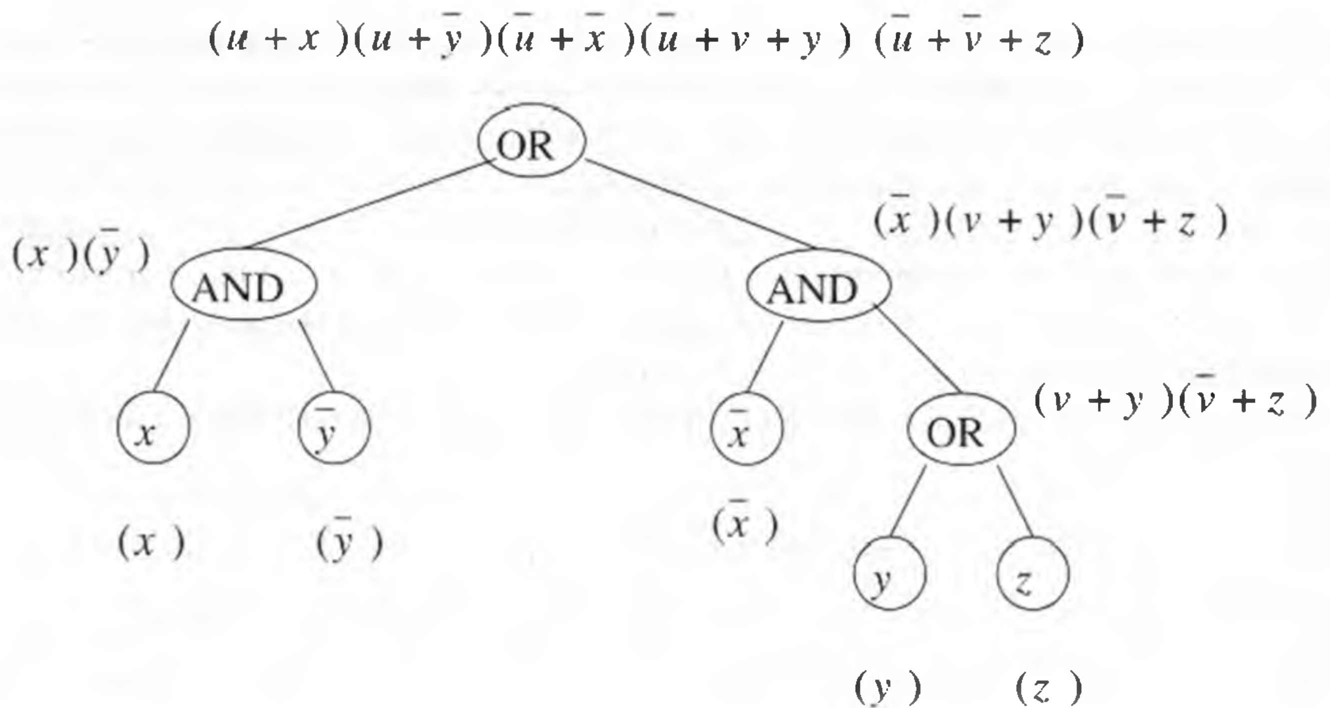


Figura 10.7 Trasformazione in CNF di un'espressione booleana.

espressioni $x\overline{y}$ e $\overline{x}(y + z)$, le cui espressioni in CNF sono state calcolate come $(x)(\overline{y})$ e $(\overline{x})(v + y)(\overline{v} + z)$. Introduciamo la nuova variabile u, aggiunta senza negazione al primo gruppo di clausole e negata al secondo. Il risultato è

$$F = (u + x)(u + \overline{y})(\overline{u} + \overline{x})(\overline{u} + v + y)(\overline{u} + \overline{v} + z)$$

Secondo il Teorema 10.13 ogni assegnamento di valori di verità T che soddisfa E si può estendere a un assegnamento S che soddisfa F. Per esempio l'assegnamento $T(x) = 0$, $T(y) = 1$ e $T(z) = 1$ soddisfa E. Possiamo estendere T a S aggiungendo $S(u) = 1$ e $S(v) = 0$ a $S(x) = 0$, $S(y) = 1$ e $S(z) = 1$ derivate da T. Si può verificare che S soddisfa F.

Osserviamo che nella scelta di S abbiamo dovuto porre $S(u) = 1$ perché T rende vera solo la seconda parte di E, cioè $\overline{x}(y + z)$. Il prodotto di clausole $(u + x)(u + \overline{y})$, derivato dalla prima parte di E, diventa vero se $S(u) = 1$. Possiamo invece scegliere qualsiasi valore per v, dal momento che entrambi i membri di OR in $y + z$ sono veri rispetto a T. □

10.3.4 NP-completezza di 3SAT

Definiamo ora una classe ancora più ristretta di espressioni booleane, per la quale il problema della soddisfacibilità è NP-completo. Ripetiamo l'enunciato di 3SAT.

- Data un'espressione booleana E in forma di prodotto di clausole, ognuna delle quali sia la somma di tre letterali distinti, E è soddisfacibile?

Pur essendo solo una piccola parte delle espressioni in CNF, quelle in 3-CNF sono abbastanza complesse da rendere NP-completa la verifica di soddisfacibilità, come dimostra il prossimo teorema.

Teorema 10.15 3SAT è NP-completo.

DIMOSTRAZIONE 3SAT è in $\mathcal{NP}$ perché SAT è in $\mathcal{NP}$. Per dimostrare che è NP-completo, riduciamo CSAT a 3SAT. Data un'espressione in CNF $E = e_1 \wedge e_2 \wedge \cdots \wedge e_k$, formiamo una nuova espressione F sostituendo ogni clausola e_i nel modo che descriveremo. Il tempo necessario per costruire F è lineare nella lunghezza di E, e vedremo che un assegnamento di valori di verità soddisfa E se e solo se possiamo estenderlo a un assegnamento che soddisfa F.

1. Se e_i è un letterale[4], poniamo x, introduciamo due nuove variabili u e v. Sostituiamo (x) con le quattro clausole $(x+u+v)(x+u+\overline{v})(x+\overline{u}+v)(x+\overline{u}+\overline{v})$. Poiché u e v compaiono in tutte le combinazioni, c'è un solo modo di soddisfare tutte le clausole: rendere vera x. Di conseguenza tutti gli assegnamenti che soddisfano E, e solo quelli, si possono estendere ad assegnamenti che soddisfano F.

2. Sia e_i la somma di due letterali: $(x + y)$. Introduciamo la nuova variabile z e sostituiamo e_i con il prodotto di due clausole $(x + y + z)(x + y + \overline{z})$. Come nel caso 1, il solo modo per soddisfarle è soddisfare $(x + y)$.

3. Se è la somma di tre letterali, e_i è già nella forma richiesta per 3-CNF e possiamo lasciarla in F così com'è.

4. Sia $e_i = (x_1 + x_2 + \cdots + x_m)$ per un $m \geq 4$. Introduciamo le nuove variabili $y_1, y_2, \ldots, y_{m-3}$ e sostituiamo e_i con il prodotto di clausole

$$\begin{gathered}(x_1 + x_2 + y_1)(x_3 + \overline{y_1} + y_2)(x_4 + \overline{y_2} + y_3) \cdots \\ (x_{m-2} + \overline{y_{m-4}} + y_{m-3})(x_{m-1} + x_m + \overline{y_{m-3}})\end{gathered} \tag{10.2}$$

Un assegnamento T che soddisfa E deve rendere vero almeno un letterale di e_i; sia x_j quel letterale (x_j può essere una variabile semplice o negata). Se dichiariamo $y_1, y_2, \ldots, y_{j-2}$ vere e $y_{j-1}, y_{j+1}, \ldots, y_{m-3}$ false, tutte le clausole di (10.2) risultano vere. Perciò T può essere estesa in modo da soddisfarle. Viceversa, se in T tutte le x sono false, non è possibile estendere T rendendo (10.2) vera. Infatti ci sono $m - 2$ clausole, e ognuna delle $m - 3$ variabili y, che sia vera o falsa, può rendere vera una sola clausola.

[4]Per comodità tratteremo i letterali come variabili non negate, per esempio x, ma le costruzioni si applicano anche se alcuni o tutti i letterali sono negati, come $\overline{x}$.

Abbiamo così mostrato come ridurre ogni istanza E di CSAT a un'istanza F di 3SAT in modo che F sia soddisfacibile se e solo se E è soddisfacibile. È chiaro che il tempo impiegato dalla costruzione è lineare rispetto alla lunghezza di E, perché in nessuno dei quattro casi svolti una clausola si espande di un fattore maggiore di 32/3 (il rapporto dei numeri di simboli nel caso 1) ed è facile determinare i simboli di F in tempo proporzionale al loro numero. Poiché CSAT è NP-completo, ne segue che lo stesso vale per 3-SAT. □

10.3.5 Esercizi

Esercizio 10.3.1 Trasformate in 3-CNF le seguenti espressioni booleane.

* a) $xy + \overline{x}z$.

b) $wxyz + u + v$.

c) $wxy + \overline{x}uv$.

Esercizio 10.3.2 Definiamo il problema 4TA-SAT: data un'espressione booleana E, stabilire se E è soddisfatta da almeno quattro assegnamenti di valori di verità. Dimostrate che 4TA-SAT è NP-completo.

Esercizio 10.3.3 In questo esercizio definiamo una famiglia di espressioni in 3-CNF. L'espressione E_n ha n variabili: $x_1, x_2, \ldots, x_n$. Per ogni insieme di tre interi distinti fra 1 ed n, E_n contiene le clausole $(x_1+x_2+x_3)$ e $(\overline{x_1}+\overline{x_2}+\overline{x_3})$. Stabilite se E_n è soddisfacibile per:

*! a) $n = 4$

!! b) $n = 5$.

Esercizio 10.3.4 Ideate un algoritmo polinomiale in tempo che risolva il problema 2SAT, cioè la soddisfacibilità di espressioni booleane in CNF con due letterali per clausola. *Suggerimento*: se uno dei due letterali di una clausola è falso, l'altro dev'essere vero. Partite da un'ipotesi sul valore di verità di una variabile ed esplorate le conseguenze per le altre.

10.4 Altri problemi NP-completi

Descriviamo ora schematicamente il processo per cui un problema NP-completo permette di dimostrare che altri problemi sono NP-completi. Questo modo di scoprire nuovi problemi NP-completi ha due effetti importanti.

- Scoprire che un problema è NP-completo indica che ci sono poche *chance* di trovare un algoritmo efficiente per risolverlo. Siamo invece incoraggiati a cercare per via euristica soluzioni parziali, approssimazioni o altri modi per aggirare il problema. Possiamo procedere così senza temere che la soluzione giusta ci sia sfuggita.

- Ogni nuovo problema NP-completo P che si aggiunge all'elenco avvalora la tesi che tutti i problemi NP-completi richiedono un tempo esponenziale. La fatica spesa nella ricerca di un algoritmo polinomiale per P è in realtà rivolta inconsciamente a dimostrare che $\mathcal{P} = \mathcal{NP}$. Proprio il peso crescente dei tentativi falliti da parte di tanti abili scienziati e matematici per dimostrare qualcosa di equivalente a $\mathcal{P} = \mathcal{NP}$ ci convince che questa uguaglianza è poco plausibile, e che invece ogni problema NP-completo richiede tempo esponenziale.

In questo paragrafo incontreremo diversi problemi NP-completi relativi ai grafi, spesso presenti nella soluzione di questioni di rilevanza pratica. Tratteremo del problema del commesso viaggiatore (TSP, *Traveling Salesman Problem*) che abbiamo già incontrato nel Paragrafo 10.1.4. Dimostreremo che una sua versione più semplice, ma altrettanto importante, detta problema del circuito hamiltoniano (HC), è NP-completa; questo dimostra anche che il più generale TSP è NP-completo. Presentiamo alcuni altri problemi di "copertura" di grafi, come il "problema della copertura per nodi", in cui si cerca il più piccolo insieme di nodi che "copre" tutti i lati, che comprende cioè almeno un estremo di ogni lato.

10.4.1 Descrivere problemi NP-completi

Nel presentare nuovi problemi NP-completi adottiamo il seguente schema di definizione.

1. Il *nome* del problema, di solito accompagnato da un'abbreviazione come 3SAT o TSP.

2. L'*input* del problema: che cosa si rappresenta, e come.

3. L'*output* richiesto: in quali circostanze l'output dev'essere "sì"?

4. Il problema da cui si compie la riduzione per dimostrare l'NP-completezza.

Esempio 10.16 Ecco come descriviamo il problema 3SAT e la dimostrazione di NP-completezza.

PROBLEMA: soddisfacibilità di espressioni in 3-CNF (3SAT).

INPUT: un'espressione booleana in 3-CNF.

OUTPUT: "sì" se e solo se l'espressione è soddisfacibile.

RIDUZIONE DA: CSAT. □

10.4.2 Il problema degli insiemi indipendenti

Sia G un grafo non orientato. Un sottoinsieme I di nodi di G si dice un *insieme indipendente* se nessuna coppia di nodi di I è collegata da un lato di G. Un insieme indipendente è *massimale* se ha almeno tanti nodi quanti ogni altro insieme indipendente.

Esempio 10.17 Nel grafo della Figura 10.1 (v. Paragrafo 10.1.2) $\{1, 4\}$ è un insieme indipendente massimale. Si tratta dell'unico insieme indipendente di cardinalità due, perché ogni altra coppia di nodi è unita da un lato. Perciò nessun insieme di tre o più elementi è indipendente; per esempio $\{1, 2, 4\}$ non è indipendente perché c'è un lato fra 1 e 2. Quindi $\{1, 4\}$ è un insieme indipendente massimale. È in effetti l'unico insieme indipendente massimale del grafo in esame; in generale un grafo può averne più di uno. Ancora un esempio per lo stesso grafo: $\{1\}$ è un insieme indipendente, ma non massimale. □

Nel campo dell'ottimizzazione combinatoria il problema dell'insieme indipendente massimale è spesso enunciato in questo modo: dato un grafo, trovare un insieme indipendente massimale. Noi invece, come d'abitudine nella teoria dei problemi intrattabili, vogliamo enunciarlo in termini "sì/no". Dobbiamo perciò incorporare nell'enunciato un limite inferiore ed esprimere il problema in questi termini: esiste nel grafo assegnato un insieme indipendente di dimensione pari almeno al limite dato? Ecco dunque la definizione formale del problema.

PROBLEMA: insieme indipendente (IS, *Independent Set*).

INPUT: un grafo G e un limite inferiore k, compreso fra 1 e il numero di nodi di G.

OUTPUT: "sì" se e solo se G ha un insieme indipendente di k nodi.

RIDUZIONE DA: 3SAT.

Come anticipato, dimostriamo che IS è NP-completo per riduzione polinomiale da 3SAT.

Teorema 10.18 Il problema dell'insieme indipendente è NP-completo.

DIMOSTRAZIONE È facile vedere che IS è in $\mathcal{NP}$: dato un grafo G e un intero k, scegliamo k nodi arbitrari e verifichiamo se sono indipendenti.

Spieghiamo ora come ridurre 3SAT a IS. Sia $E = (e_1)(e_2)\cdots(e_m)$ un'espressione in 3-CNF. A partire da E costruiamo un grafo G con $3m$ nodi, ai quali diamo i nomi $[i, j]$, con $1 \leq i \leq m$ e $j = 1, 2$ o 3. Il nodo $[i, j]$ rappresenta il j-esimo letterale della clausola e_i. La Figura 10.8 presenta il grafo G derivato dall'espressione in 3-CNF

$$(x_1 + x_2 + x_3)(\overline{x_1} + x_2 + x_4)(\overline{x_2} + x_3 + x_5)(\overline{x_3} + \overline{x_4} + \overline{x_5})$$

Le colonne rappresentano le clausole. Spiegheremo tra breve come sono stati scelti i lati.

La costruzione di G si fonda sull'impiego dei lati per imporre che ogni insieme indipendente di m nodi rappresenti un modo di soddisfare l'espressione E. Questa idea si articola in due punti fondamentali.

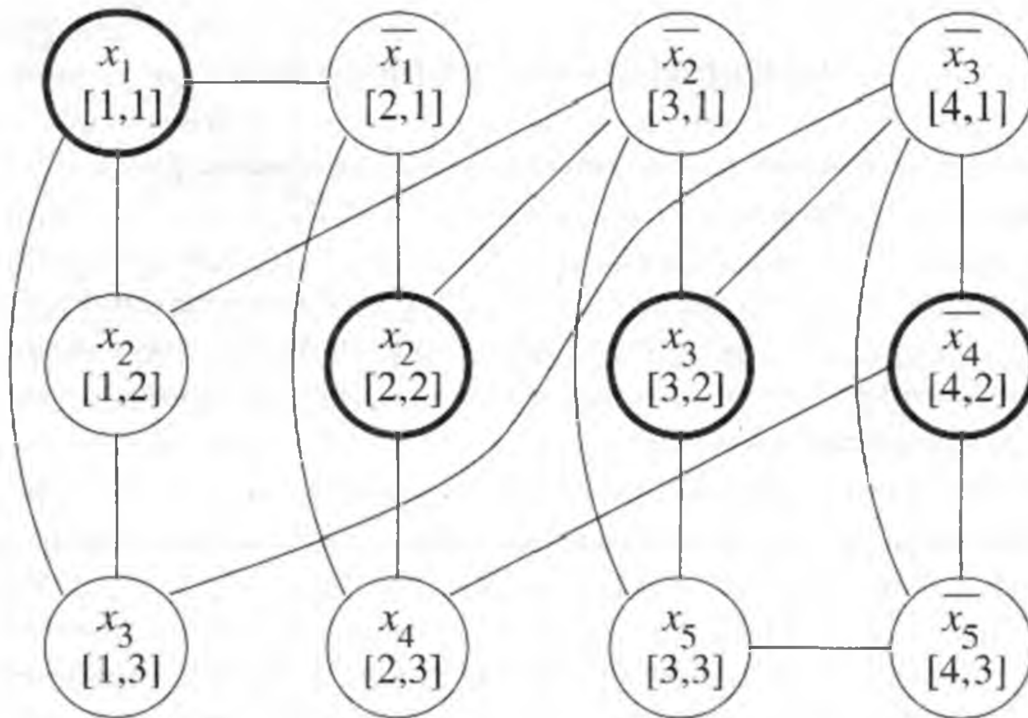


Figura 10.8 Costruzione di un insieme indipendente da un'espressione booleana soddisfacibile, in 3-CNF.

1. Vogliamo far sì che si possa scegliere un solo nodo per ogni clausola. A tal fine uniamo con un lato tutte le coppie di nodi nella stessa colonna. I lati di questo tipo sono $([i,1], [i,2])$, $([i,1], [i,3])$ e $([i,2], [i,3])$, per ogni i, come nella Figura 10.8.

2. Dobbiamo impedire che i nodi che rappresentano letterali complementari possano far parte di un insieme indipendente. Perciò uniamo con un lato due nodi $[i_1, j_1]$ e $[i_2, j_2]$ se uno di questi rappresenta la variabile x e l'altro la variabile $\overline{x}$. In questo modo non possiamo sceglierli per lo stesso insieme indipendente.

Il limite k per il grafo G costruito da queste due regole è m.

Non è difficile vedere che il grafo G e il limite k si possono determinare dall'espressione E in tempo proporzionale al quadrato della lunghezza di E. La trasformazione da E a G è quindi una riduzione polinomiale. Dobbiamo dimostrare che si tratta di una riduzione corretta di 3SAT a IS, cioè l'enunciato seguente.

- E è soddisfacibile se e solo se G ha un insieme indipendente di dimensione m.

(Se) Osserviamo in primo luogo che un insieme indipendente non può contenere due nodi della stessa clausola, cioè $[i, j_1]$ e $[i, j_2]$ per $j_1 \neq j_2$. Questo dipende dalla presenza di lati fra coppie di quei nodi, come si vede dalle colonne nella Figura 10.8. Se dunque esiste

Sulla facilità dei problemi sì/no

Potremmo sospettare che la versione sì/no di un problema sia più facile di quella di ottimizzazione. Per esempio, può essere difficile trovare un insieme indipendente di dimensione massima, mentre verificare che ci sia un insieme indipendente di dimensione k, per un valore piccolo di k, potrebbe essere semplice. Questo è vero, ma la costante k potrebbe essere proprio la dimensione massima di un insieme indipendente. In tal caso risolvere la versione sì/no equivale a trovare un insieme indipendente massimale.

In effetti, per tutti i comuni problemi NP-completi, la versione sì/no e quella di ottimizzazione sono equivalenti per complessità, a meno di un polinomio. In un caso tipico, come IS, se avessimo un algoritmo polinomiale in tempo per trovare gli insiemi indipendenti massimali, saremmo in grado di risolvere il problema sì/no trovando un insieme massimale e osservando se la sua dimensione è almeno k. Poiché dimostriamo che la versione sì/no è NP-completa, anche l'altra dev'essere intrattabile.

Possiamo fare il confronto anche nell'altro senso. Supponiamo che ci sia un algoritmo polinomiale in tempo per la versione sì/no di IS. Se il grafo ha n nodi, la dimensione dell'insieme indipendente massimale è compresa fra 1 ed n. Risolvendo IS per tutti i valori fra 1 ed n siamo certi di trovare la dimensione di un insieme indipendente massimale (anche se non l'insieme stesso) in un tempo pari a n volte quello necessario per risolvere il problema una volta. Ricorrendo alla ricerca binaria possiamo addirittura ridurre a $\log_2 n$ il fattore moltiplicativo.

un insieme indipendente di dimensione m, esso deve contenere esattamente un nodo per ogni clausola.

Inoltre un insieme indipendente non può contenere nodi corrispondenti a una variabile x e alla sua negata $\overline{x}$. C'è infatti un lato fra tutte le coppie di quei nodi. Pertanto un insieme indipendente I di dimensione m genera un assegnamento di valori di verità T che soddisfa E. Se il nodo corrispondente alla variabile x è in I, definiamo $T(x) = 1$; se il nodo associato a $\overline{x}$ è in I, definiamo $T(x) = 0$. Se nessun nodo in I corrisponde a x o a $\overline{x}$, definiamo arbitrariamente $T(x)$. Il punto (2) discusso sopra spiega perché non si può generare una contraddizione, cioè la compresenza in I dei nodi associati a x e a $\overline{x}$.

Sosteniamo che T soddisfa E. Infatti, per ogni clausola di E, I contiene un nodo corrispondente a uno dei suoi letterali, e T assegna il valore vero a quel letterale. Perciò, se esiste un insieme indipendente di dimensione m, E è soddisfacibile.

(Solo se) Supponiamo che esista un assegnamento T che soddisfa E. Poiché T rende vera ogni clausola di E, per ognuna di esse c'è almeno un letterale vero rispetto a T. Se ciò vale per due o tre letterali in una clausola, ne scegliamo uno in modo arbitrario. Formiamo un insieme I di m nodi scegliendo per ogni clausola il nodo abbinato al letterale prescelto.

Affermiamo che I è un insieme indipendente. Se un lato unisce due nodi derivati dalla stessa clausola (le colonne nella Figura 10.8), quei nodi non possono trovarsi entrambi in I perché abbiamo scelto un solo nodo per clausola. Se un lato unisce una variabile e la sua negata, i suoi estremi non possono trovarsi entrambi in I perché abbiamo scelto solo nodi corrispondenti a letterali veri rispetto a T. T può assegnare il valore vero a x o a $\overline{x}$, ma non a entrambi. Concludiamo che se E è soddisfacibile, G ha un insieme indipendente di dimensione m.

C'è quindi una riduzione polinomiale di 3SAT a IS. Per il Teorema 10.5, IS è NP-completo dato che lo stesso vale per 3SAT. □

Esempio 10.19 Applichiamo la costruzione del Teorema 10.18 al caso in cui

$$E = (x_1 + x_2 + x_3)(\overline{x_1} + x_2 + x_4)(\overline{x_2} + x_3 + x_5)(\overline{x_3} + \overline{x_4} + \overline{x_5})$$

Il grafo ricavato da questa espressione è illustrato nella Figura 10.8. I nodi sono disposti in quattro colonne corrispondenti alle quattro clausole. Per ogni nodo sono riportati il nome (una coppia di interi) e il letterale associato. Ogni coppia di nodi nella stessa colonna, corrispondenti a letterali in una clausola, è unita da un lato. Anche i nodi corrispondenti a una variabile e alla sua negata sono uniti da un lato. Per esempio il nodo $[3, 1]$, che corrisponde a $\overline{x_2}$, è unito da lati ai nodi $[1, 2]$ e $[2, 2]$, corrispondenti a occorrenze di x_2.

Un tratto più spesso indica i nodi, uno per colonna, che formano un insieme I. È chiaro che essi formano un insieme indipendente. I quattro letterali in I sono x_1, x_2, x_3 e $\overline{x_4}$. Possiamo definire un assegnamento di valori di verità T: $T(x_1) = 1$, $T(x_2) = 1$, $T(x_3) = 1$ e $T(x_4) = 0$. Dobbiamo assegnare un valore anche a x_5, e possiamo sceglierlo arbitrariamente, per esempio $T(x_5) = 0$. Così T soddisfa E, e l'insieme di nodi I indica un letterale per ogni clausola, al quale T assegna il valore vero. □

10.4.3 Il problema della copertura per nodi

Un'altra classe rilevante di problemi di ottimizzazione combinatoria riguarda la "copertura" di un grafo. Per esempio una *copertura per lati* è un insieme di lati tali che ogni nodo è l'estremo di almeno un lato dell'insieme. Una copertura per lati è *minimale* se ogni altra copertura dello stesso grafo ha un numero di lati uguale o maggiore. Stabilire se un grafo ha una copertura per lati di k elementi richiede tempo polinomiale, ma non lo dimostreremo.

Proveremo invece che il problema della *copertura per nodi* è NP-completo. Una copertura per nodi di un grafo è un insieme di nodi che contiene almeno un estremo di

Utilità degli insiemi indipendenti

Le applicazioni dei problemi NP-completi che presentiamo esulano dalla materia di questo libro. La scelta dei problemi trattati nel Paragrafo 10.4 si deve a un fondamentale lavoro di R. Karp sull'NP-completezza, nel quale l'autore esamina i problemi più importanti nel campo della ricerca operativa e dimostra che molti sono NP-completi. Di qui la chiara indicazione che certi problemi "reali" possono essere risolti mediante quelli astratti che abbiamo presentato.

Possiamo per esempio impiegare un algoritmo per la ricerca di insiemi indipendenti allo scopo di programmare una sessione di esami. A ogni corso associamo un nodo del grafo; uniamo due nodi con un lato se uno o più studenti seguono i due corsi corrispondenti. Gli esami per quei due corsi non possono aver luogo contemporaneamente. Se troviamo un insieme indipendente massimale, possiamo convocare gli esami dei corsi corrispondenti nella stessa data, senza causare problemi a nessuno studente.

ogni lato. Diciamo che è *minimale* se ogni altra copertura per nodi dello stesso grafo ha un numero di nodi uguale o maggiore.

Le coperture per nodi sono strettamente legate agli insiemi indipendenti. Il complemento di un insieme indipendente è una copertura per nodi, e viceversa. Perciò, a patto di formulare opportunamente la versione sì/no del problema della copertura per nodi, la riduzione da IS è molto semplice.

PROBLEMA: il problema della copertura per nodi (NC, *Node Cover*).

INPUT: un grafo G e un limite superiore k, compreso fra 0 e il numero di nodi di G meno uno.

OUTPUT: "sì" se e solo se G ha una copertura per nodi con non più di k nodi.

RIDUZIONE DA: insieme indipendente.

Teorema 10.20 Il problema della copertura per nodi è NP-completo.

DIMOSTRAZIONE NC è chiaramente in $\mathcal{NP}$. Scegliamo un insieme di k nodi e verifichiamo che contenga almeno un estremo di ogni lato di G.

Per completare la dimostrazione, riduciamo IS a NC. Ci basiamo sul fatto, illustrato dalla Figura 10.8, che il complemento di un insieme indipendente è una copertura per nodi. Per esempio l'insieme dei nodi di tratto più sottile nella figura è una copertura per nodi. Poiché i nodi con tratto spesso formano un insieme indipendente massimale, gli altri formano una copertura minimale.

Siano G e k gli elementi di un'istanza del problema dell'insieme indipendente. Se G ha n nodi, siano G e $n - k$ gli elementi dell'istanza di NC che vogliamo costruire. La trasformazione si può evidentemente compiere in tempo lineare. Sosteniamo che

- G ha un insieme indipendente di dimensione k se e solo se G ha una copertura per nodi di dimensione $n - k$.

(Se) Sia N l'insieme dei nodi di G, e C la copertura per nodi di dimensione $n - k$. Affermiamo che $N - C$ è un insieme indipendente. Supponiamo il contrario, cioè che ci siano due nodi v e w in $N - C$ uniti da un lato di G. Poiché né v né w sono in C, il lato (v, w) di G non è coperto da C. Abbiamo quindi dimostrato per assurdo che $N - C$ è un insieme indipendente. Questo insieme ha chiaramente k nodi, e questa parte della dimostrazione è conclusa.

(Solo se) Sia I un insieme indipendente di k nodi. Affermiamo che $N - I$ è una copertura per nodi con $n - k$ nodi. Ragioniamo per assurdo. Se un lato (v, w) non è coperto da $N - I$, sia v sia w sono in I; ma essi sono uniti da un lato, il che contraddice la definizione di insieme indipendente. □

10.4.4 Il problema del circuito hamiltoniano orientato

Vogliamo dimostrare che il problema del commesso viaggiatore (TSP) è NP-completo perché si tratta di un problema molto rilevante nell'ambito dell'ottimizzazione combinatoria. La migliore dimostrazione nota della sua NP-completezza è in realtà la prova che un problema più semplice, il "problema del circuito hamiltoniano (HC)", è NP-completo.

PROBLEMA: problema del circuito hamiltoniano (HC).

INPUT: un grafo non orientato G.

OUTPUT: "sì" se e solo se G ha un *circuito hamiltoniano*, cioè un ciclo che passa per ogni nodo di G esattamente una volta.

Osserviamo che HC è un caso speciale del TSP, in cui i pesi dei lati sono tutti pari a 1. Ridurre HC a TSP in tempo polinomiale è quindi molto semplice: assegniamo il peso 1 a ogni lato del grafo.

La dimostrazione di NP-completezza di HC è molto difficile. Introdurremo una variante di HC in cui i lati hanno una direzione (sono cioè orientati) e sono chiamati *archi*. Un circuito hamiltoniano deve percorrere gli archi nella giusta direzione. Riduciamo 3SAT alla variante orientata di HC; ridurremo questa alla versione non orientata in un secondo tempo.

PROBLEMA: problema del circuito hamiltoniano orientato (DHC, *Directed Hamiltonian Circuit*).

INPUT: un grafo orientato G.

OUTPUT: "sì" se e solo se esiste in G un ciclo orientato che passa per ogni nodo esattamente una volta.

RIDUZIONE DA: 3SAT.

Teorema 10.21 Il problema del circuito hamiltoniano orientato è NP-completo.

DIMOSTRAZIONE È facile provare che DHC è in $\mathcal{NP}$: si sceglie un ciclo potenziale e si verifica che tutti i suoi archi siano presenti nel grafo. Per ridurre 3SAT a DHC dobbiamo costruire un grafo complicato, con "blocchi", cioè sottografi specializzati, che rappresentano le variabili e le clausole dell'istanza di 3SAT.

Per avviare la costruzione di un'istanza di DHC a partire da un'espressione booleana in 3-CNF, poniamo $E = e_1 \wedge e_2 \wedge \cdots \wedge e_k$, dove ogni clausola e_i è la somma di tre letterali: $e_i = (\alpha_{i1} + \alpha_{i2} + \alpha_{i3})$. Siano $x_1, x_2, \ldots, x_n$ le variabili di E. Per ogni clausola e per ogni variabile costruiamo un blocco, secondo la Figura 10.9.

Per ogni variabile x_i definiamo il sottografo H_i illustrato nella Figura 10.9(a). Il numero m_i è il più grande fra il numero di occorrenze di x_i e quello di $\overline{x_i}$ in E. Fra le due colonne di nodi, denotate da b e c, poniamo archi, in entrambe le direzioni, fra b_{ij} e c_{ij}. Da ogni nodo b esce poi un arco verso il nodo c al livello sottostante. Da b_{ij} c'è quindi un arco verso $c_{i,j+1}$ se $j < m_i$. Simmetricamente un arco va da c_{ij} a $b_{i,j+1}$, per $j < m_i$. Infine ci sono un nodo sommitale a_i con archi verso i nodi b_{i0} e c_{i0}, e un nodo di fondo d_i al quale giungono due archi, da b_{im_i} e da c_{im_i}.

La struttura dell'intero grafo è delineata nella Figura 10.9(b). Ogni esagono rappresenta un blocco relativo a una variabile, strutturato come la Figura 10.9(a). I blocchi sono legati in forma di ciclo mediante archi dal nodo di fondo di un blocco al nodo sommitale del successivo.

Supponiamo che questo grafo abbia un circuito hamiltoniano orientato e che il circuito parta da a_1. Se il nodo successivo è b_{10}, esso dev'essere seguito da c_{10}, altrimenti c_{10} non potrebbe far parte del circuito. Infatti, se il ciclo andasse da a_1 a b_{10} a c_{11}, non potrebbe più raggiungere c_{10}, avendo già usato i suoi due predecessori.

Se il ciclo comincia con a_1 e b_{10}, deve scendere la "scala" alternando i due lati:

$$a_1, b_{10}, c_{10}, b_{11}, c_{11}, \ldots, b_{1m_1}, c_{1m_1}, d_1$$

Se il ciclo comincia con a_1, c_{10}, la scala viene percorsa in un ordine in cui, a ogni livello, il nodo c precede il nodo b:

$$a_1, c_{10}, b_{10}, c_{11}, b_{11}, \ldots, c_{1m_1}, b_{1m_1}, d_1$$

Dobbiamo fare un'osservazione cruciale: possiamo interpretare il primo ordine di discesa, dal c al b sottostante, come l'assegnazione del valore "vero" alla variabile del blocco del

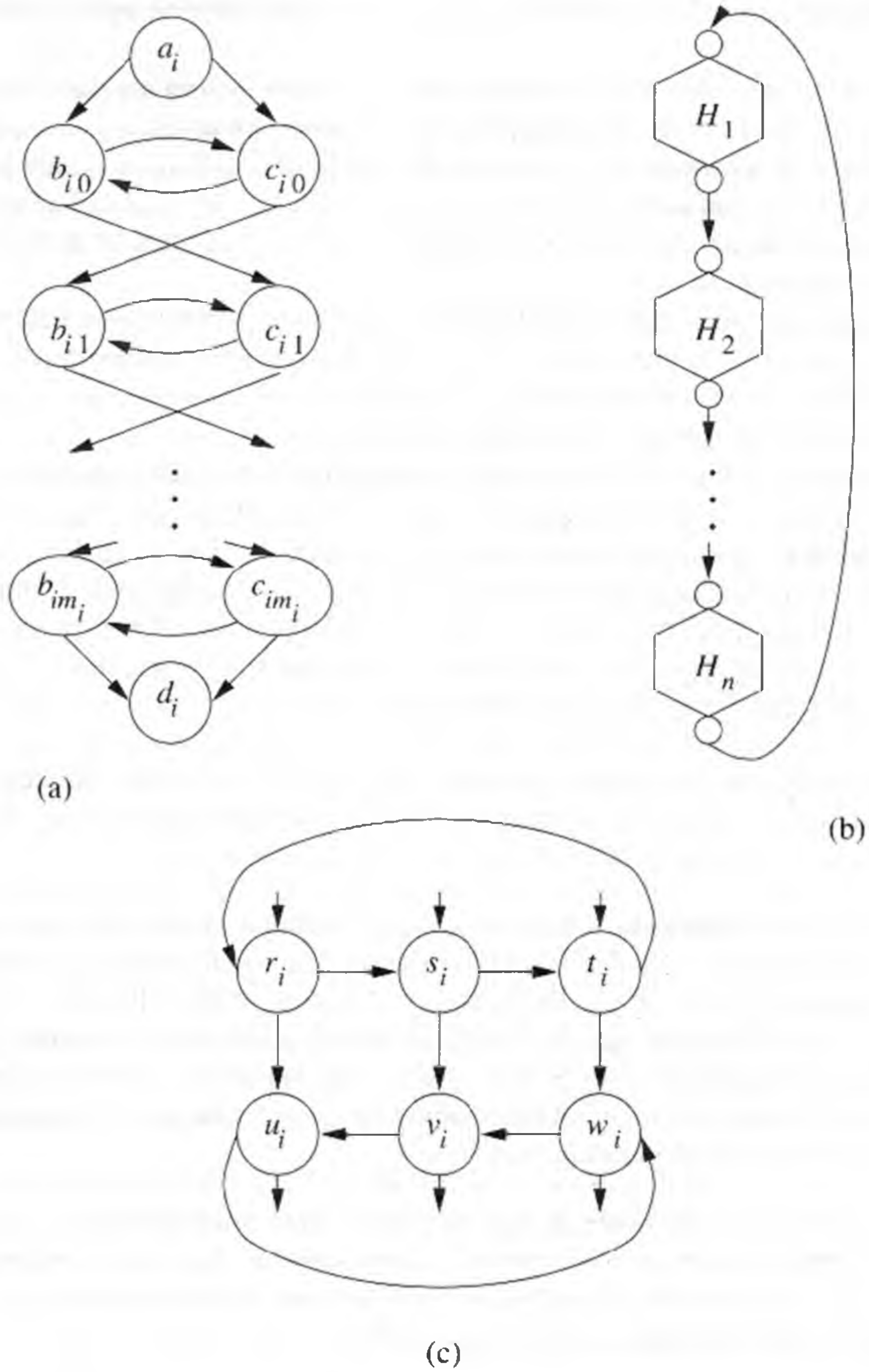


Figura 10.9 Elementi per la dimostrazione che il problema del circuito hamiltoniano è NP-completo.

grafo in esame, e il secondo ordine, da un b al c sottostante, come assegnazione del valore "falso".

Dopo aver attraversato H_1, il ciclo procede in a_2, dove si deve scegliere fra b_{20} e c_{20}. Come abbiamo spiegato per H_1, compiuta questa prima scelta, il resto del cammino in H_2 è obbligato. In generale, all'ingresso in H_i possiamo scegliere se andare a sinistra o a destra. In seguito non abbiamo scelta, se vogliamo evitare di rendere *inaccessibile* un nodo (cioè irraggiungibile nel circuito hamiltoniano orientato), perché i suoi predecessori nel grafo sono già stati usati.

Nel seguito conviene considerare la scelta di andare da a_i a b_{i0} come l'assegnamento di "vero" a x_i e quella di andare da a_i a c_{i0} come l'assegnarle il valore "falso". Perciò il grafo della Figura 10.9(b) ha esattamente 2^n circuiti hamiltoniani orientati, corrispondenti ai 2^n assegnamenti di valori di verità alle n variabili.

La Figura 10.9(b) è però solo lo scheletro del grafo abbinato all'espressione in 3-CNF E. Per ogni clausola e_j introduciamo il sottografo I_j riprodotto nella Figura 10.9(c). Se un ciclo entra in I_j da r_j, deve uscirne da u_j; se entra da s_j, deve uscire da v_j; se entra da t_j, deve uscire da w_j. Per provarlo osserviamo che se il ciclo, raggiunto I_j, non esce dal nodo sottostante quello di entrata, uno o più nodi risultano inaccessibili e non possono far parte del ciclo. Sfruttando la simmetria possiamo trattare solo il caso in cui r_j è il primo nodo di I_j nel ciclo. Distinguiamo tre situazioni.

1. I due vertici successivi nel ciclo sono s_j e t_j. Se il ciclo va a w_j ed esce, v_j risulta inaccessibile. Se attraversa w_j e v_j ed esce, u_j risulta inaccessibile. Perciò deve uscire da u_j dopo aver attraversato tutti i sei nodi del blocco.

2. I vertici che seguono r_j sono s_j e v_j. Se il ciclo non prosegue da u_j, questo risulta inaccessibile. Se dopo u_j il ciclo passa a w_j, non può più passare da t_j. Il ragionamento "inverte" quello fondato sull'inaccessibilità. Il nodo t_j può essere raggiunto dall'esterno, ma se il ciclo include t_j più avanti, non potrà proseguire perché i due successori di t_j sono già apparsi nel ciclo. Anche in questo caso, quindi, il ciclo esce da u_j. Osserviamo che t_j e w_j non sono stati attraversati, e dovranno comparire più avanti nel ciclo.

3. Il circuito va direttamente da r_j a u_j. Se il ciclo prosegue con w_j, t_j non può comparire nel ciclo perché i suoi successori sono già stati usati, come spiegato al punto (2). In questo caso, quindi, il ciclo deve uscire direttamente da u_j; restano quattro nodi da aggiungere al ciclo più tardi.

Per completare la costruzione del grafo G da un'espressione E, dobbiamo collegare gli I_j agli H_i. Sia x_i, variabile non negata, il primo letterale della clausola e_j. Scegliamo un nodo c_{ip}, con p compreso fra 0 e $m_i - 1$, non ancora usato per allacciarsi a uno dei blocchi I. Aggiungiamo gli archi da c_{ip} a r_j e da u_j a $b_{i,p+1}$. Se il primo letterale della clausola

e_j è $\overline{x_i}$, cioè una variabile negata, cerchiamo un b_{ip} non ancora usato. Colleghiamo b_{ip} a r_j e u_j a $c_{i,p+1}$.

Per il secondo e il terzo letterale di e_j operiamo allo stesso modo, con un'eccezione. Per il secondo usiamo i nodi s_j e v_j, per il terzo i nodi t_j e w_j. In questo modo ogni I_j ha tre collegamenti ai blocchi H che rappresentano le variabili della clausola e_j. Il collegamento deriva da un nodo c e ritorna al nodo b sottostante se il letterale non è negato, proviene da un nodo b e ritorna al nodo c sottostante se è negato. Affermiamo che

- il grafo G costruito in questo modo ha un circuito hamiltoniano orientato se e solo se l'espressione E è soddisfacibile.

(Se) Supponiamo che esista un assegnamento di valori di verità T che soddisfa E. Costruiamo un circuito hamiltoniano orientato.

1. Cominciamo dal cammino che attraversa solo gli H, cioè i grafi della Figura 10.9(b), secondo l'assegnamento T. Il ciclo va dunque da a_i a b_{i0} se $T(x_i) = 1$, e da a_i a c_{i0} se $T(x_i) = 0$.

2. Se il ciclo costruito fin qui percorre un arco da b_{ip} a $c_{i,p+1}$ ed esiste un altro arco da b_{ip} a un I_j non ancora incluso nel ciclo, introduciamo una "deviazione" che comprende i sei nodi di I_j e ritorna a $c_{i,p+1}$. Il nuovo ciclo non conterrà più l'arco $b_{ip} \to c_{i,p+1}$, ma percorre ancora i suoi estremi.

3. In modo simile, se il ciclo contiene un arco da c_{ip} a $b_{i,p+1}$, e un altro arco, non incorporato nel ciclo, esce da c_{ip} per andare a un I_j, modifichiamo il ciclo facendolo "deviare" attraverso tutti i sei nodi di I_j.

Poiché T soddisfa E, il cammino originale costruito al passo (1) deve includere almeno un arco che, al passo (2) o (3), permette di inserire il blocco I_j per ogni clausola e_j. Tutti gli I_j vengono così inclusi nel ciclo, che diventa un circuito hamiltoniano orientato.

(Solo se) Supponiamo che il grafo G abbia un circuito hamiltoniano orientato. Dobbiamo provare che E è soddisfacibile. Ripetiamo due punti importanti ricavati dall'analisi condotta fin qui.

1. Se un circuito hamiltoniano entra in I_j da r_j, s_j o t_j, deve uscirne, rispettivamente, da u_j, v_j o w_j.

2. Perciò, se pensiamo che il circuito segua il ciclo dei blocchi H, secondo lo schema della Figura 10.9(b), possiamo interpretare una deviazione verso I_j come se il ciclo seguisse un arco "in parallelo" con $b_{ip} \to c_{i,p+1}$ o con $c_{ip} \to b_{i,p+1}$.

Se ignoriamo le deviazioni negli I_j il circuito hamiltoniano dev'essere uno dei 2^n cicli possibili usando solo gli H_i, quelli ricavati dalla scelta di passare da ogni a_i a b_{i0} o a c_{i0}. Ogni scelta corrisponde a un assegnamento di valori di verità alle variabili di E. Se una di queste scelte produce un circuito hamiltoniano che comprende gli I_j, l'assegnamento corrispondente soddisfa E.

Infatti, se il ciclo va da a_i a b_{i0}, possiamo deviare verso I_j solo se nella j-esima clausola x_i compare come letterale. Se il ciclo va da a_i a c_{i0}, possiamo deviare verso I_j solo se $\overline{x_i}$ compare nella j-esima clausola come letterale. Perciò, se tutti i blocchi I_j possono essere inclusi, allora l'assegnamento rende vero almeno uno dei tre letterali di ogni clausola, ed E è soddisfacibile. □

Esempio 10.22 Diamo un esempio molto semplice della costruzione del Teorema 10.21 a partire dall'espressione in 3-CNF $E = (x_1 + x_2 + x_3)(\overline{x_1} + \overline{x_2} + x_3)$. Il grafo risultante è illustrato nella Figura 10.10. Per chiarezza gli archi che collegano i blocchi di tipo H a quelli di tipo I sono punteggiati, ma non c'è differenza con quelli a tratto continuo.

Il blocco in alto a sinistra corrisponde a x_1. Poiché x_1 compare una volta negata e una volta no, la "scala" ha un solo dislivello, e ci sono quindi due righe di b e c. In fondo a sinistra vediamo il blocco per x_3, che compare due volte, entrambe non negate. Occorrono quindi due archi distinti $c_{3p} \to b_{3,p+1}$, di cui ci serviamo per unire i blocchi I_1 e I_2, a rappresentare gli impieghi di x_3 nelle due clausole. Per questo motivo il blocco per x_3 vuole tre righe di b e c.

Consideriamo il blocco I_2, che corrisponde alla clausola $(\overline{x_1} + \overline{x_2} + x_3)$. Per il primo letterale, $\overline{x_1}$, uniamo b_{10} a r_2 e u_2 a c_{11}. Per il secondo, $\overline{x_2}$, facciamo lo stesso con b_{20}, s_2, v_2 e c_{21}. Il terzo letterale, non essendo negato, si lega a un c e al b sottostante. Uniamo cioè c_{31} a t_2 e w_2 a b_{32}.

Uno degli assegnamenti soddisfacenti è $x_1 = 1$, $x_2 = 0$, $x_3 = 0$. Con questo assegnamento la prima clausola è soddisfatta dal suo primo letterale x_1, la seconda dal secondo letterale, cioè $\overline{x_2}$. Possiamo individuare un circuito hamiltoniano che comprende gli archi $a_1 \to b_{10}$, $a_2 \to c_{20}$ e $a_3 \to c_{30}$. Il ciclo copre la prima clausola deviando da H_1 a I_1. Usa cioè l'arco $c_{10} \to r_1$, attraversa tutti i nodi di I_1 e ritorna a b_{11}. La seconda clausola è coperta dalla deviazione da H_2 a I_2, che parte dall'arco $b_{20} \to s_2$, attraversa I_2 e torna a c_{21}. L'intero ciclo hamiltoniano è evidenziato dai tratti più spessi (continui o punteggiati) e dalle frecce più grandi nella Figura 10.10. □

10.4.5 Circuiti hamiltoniani non orientati e TSP

Dimostrare che il problema del circuito hamiltoniano non orientato e il problema del commesso viaggiatore sono NP-completi è relativamente facile. Abbiamo già visto nel Paragrafo 10.1.4 che TSP è in $\mathcal{NP}$. Poiché HC è un caso speciale di TSP, anch'esso si trova in $\mathcal{NP}$. Effettuiamo quindi le riduzioni da DHC ad HC e da HC a TSP.

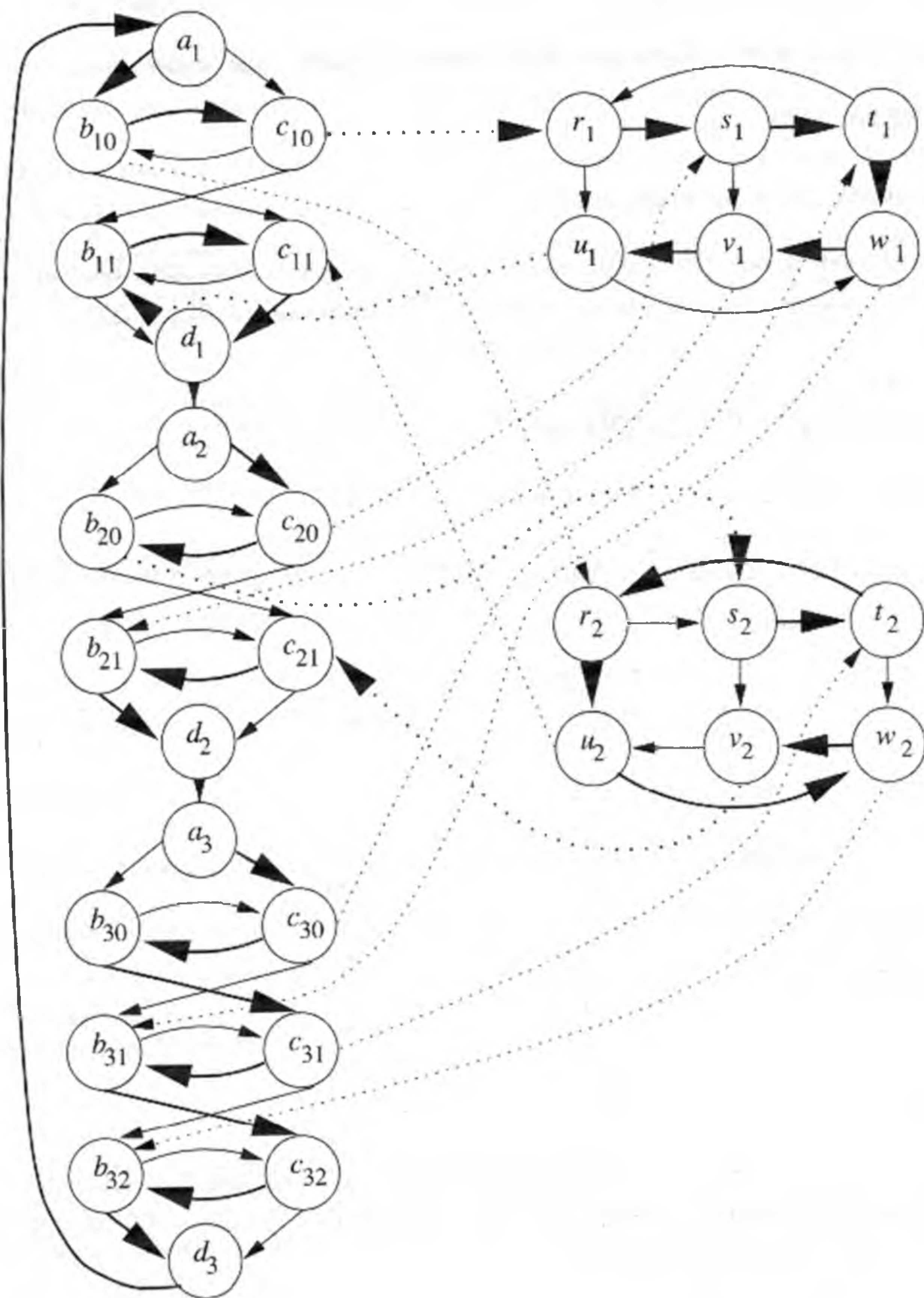


Figura 10.10 Esempio di costruzione di circuito hamiltoniano.

PROBLEMA: problema del circuito hamiltoniano non orientato.

INPUT: un grafo non orientato G.

OUTPUT: "sì" se e solo se G ha un circuito hamiltoniano.

RIDUZIONE DA: DHC.

Teorema 10.23 HC è NP-completo.

DIMOSTRAZIONE Ridurremo DHC ad HC. Sia G_d un grafo orientato. Denotiamo con G_u il grafo non orientato che dobbiamo costruire. Per ogni nodo v di G_d poniamo in G_u i tre nodi $v^{(0)}$, $v^{(1)}$ e $v^{(2)}$. I lati di G_u sono i seguenti.

1. Per ogni nodo v di G_d poniamo in G_u i lati $(v^{(0)}, v^{(1)})$ e $(v^{(1)}, v^{(2)})$.

2. Se in G_d c'è un arco $v \to w$, poniamo in G_u il lato $(v^{(2)}, w^{(0)})$.

Lo schema dei lati è illustrato nella Figura 10.11, che comprende il lato corrispondente all'arco $v \to w$.

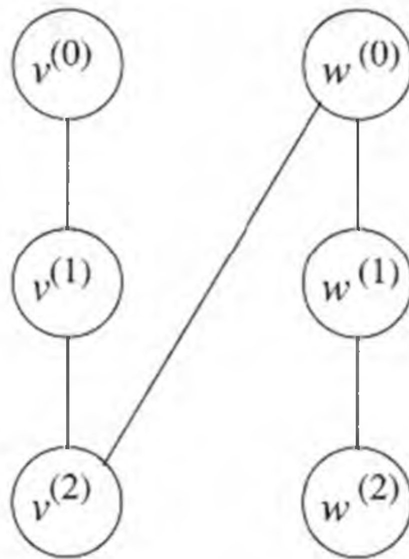


Figura 10.11 Gli archi in G_d sono sostituiti in G_u da lati che uniscono un nodo di indice 2 a un nodo di indice 0.

È evidente che la costruzione di G_u da G_d può compiersi in tempo polinomiale. Dobbiamo dimostrare che

- G_u ha un circuito hamiltoniano se e solo se G_d ha un circuito hamiltoniano orientato.

(Se) Sia $v_1, v_2, \ldots, v_n, v_1$ un circuito hamiltoniano orientato. Certamente

$$v_1^{(0)}, v_1^{(1)}, v_1^{(2)}, v_2^{(0)}, v_2^{(1)}, v_2^{(2)}, v_3^{(0)}, \ldots, v_n^{(0)}, v_n^{(1)}, v_n^{(2)}, v_1^{(0)}$$

è un circuito hamiltoniano non orientato di G_u, ottenuto percorrendo verso il basso ogni colonna e saltando alla sommità della successiva seguendo un arco di G_d.

(Solo se) Osserviamo che ogni nodo $v^{(1)}$ di G_u ha due soli lati adiacenti. In un circuito hamiltoniano esso deve quindi avere $v^{(0)}$ o $v^{(2)}$ come predecessore immediato e l'altro come successore immediato. Perciò gli apici dei nodi in un circuito hamiltoniano di G_u devono seguire lo schema $0, 1, 2, 0, 1, 2, \ldots$ o il suo opposto $2, 1, 0, 2, 1, 0, \ldots$ Poiché i due schemi corrispondono ad attraversare il ciclo nelle due direzioni, possiamo supporre che lo schema sia $0, 1, 2, 0, 1, 2, \ldots$ I lati del ciclo che uniscono un nodo con apice 2 a uno con apice 0 corrispondono ad archi di G_d e vengono percorsi nella direzione dell'arco. Quindi da un circuito hamiltoniano non orientato di G_u ricaviamo un circuito hamiltoniano orientato di G_d. □

PROBLEMA: problema del commesso viaggiatore.

INPUT: un grafo non orientato G con pesi interi sui lati, e un limite k.

OUTPUT: "sì" se e solo se in G c'è un circuito hamiltoniano tale che la somma dei pesi dei lati che lo compongono è minore o uguale a k.

RIDUZIONE DA: HC.

Teorema 10.24 Il problema del commesso viaggiatore è NP-completo.

DIMOSTRAZIONE Procediamo per riduzione da HC. Dato un grafo G, costruiamo un grafo pesato G' con gli stessi nodi e lati di G, ma con il peso 1 su tutti i lati. Il limite k è uguale al numero n di nodi di G. In G' c'è un circuito hamiltoniano di peso n se e solo se in G c'è un circuito hamiltoniano. □

10.4.6 Riepilogo dei problemi NP-completi

La Figura 10.12 indica tutte le riduzioni compiute in questo capitolo. Sono indicate anche le riduzioni da tutti i problemi, per esempio da TSP, a SAT. In realtà nel Teorema 10.9 abbiamo ridotto a SAT il linguaggio di ogni macchina di Turing non deterministica e polinomiale in tempo. Anche se non l'abbiamo detto esplicitamente, queste TM ne comprendono una che risolve TSP, una che risolve IS, e così via. Di conseguenza tutti i problemi NP-completi si possono ridurre l'uno all'altro in tempo polinomiale e rappresentano, di fatto, aspetti diversi dello stesso problema.

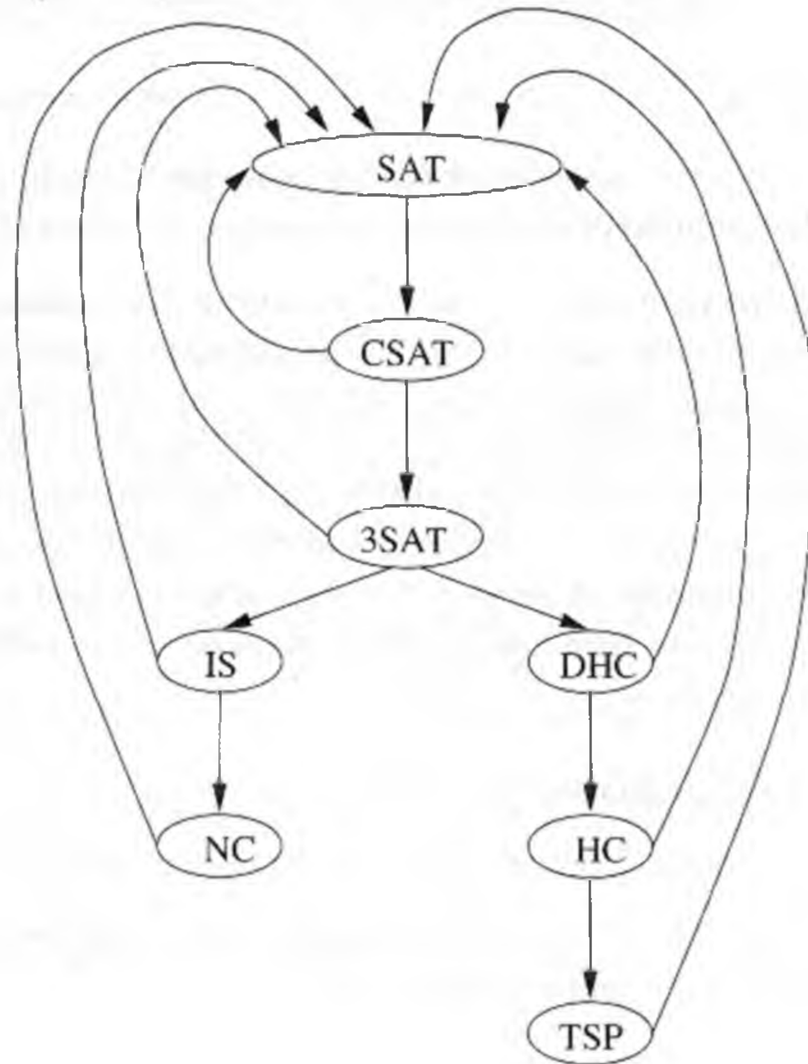


Figura 10.12 Riduzioni fra problemi NP-completi.

10.4.7 Esercizi

* **Esercizio 10.4.1** Una *k-clique* di un grafo G è un insieme di k nodi di G tale che ogni coppia di suoi nodi è unita da un lato. Una 2-*clique* è quindi una coppia di nodi collegati da un lato, e una 3-*clique* è un triangolo. Definiamo il problema CLIQUE: dato un grafo G e una costante k, G ha una *k-clique*?

a) Qual è il più grande valore di k per cui il grafo G nella Figura 10.1 soddisfa CLIQUE?

b) Quanti lati ha una *k-clique* in funzione di k?

c) Dimostrate che CLIQUE è NP-completo riducendo il problema della copertura per nodi a CLIQUE.

*! **Esercizio 10.4.2** Un *problema di colorazione* è così definito: dati un grafo G e un intero k, possiamo assegnare a ogni nodo un colore scelto fra k in modo che nessun lato abbia gli estremi dello stesso colore? Il grafo della Figura 10.1, per esempio, si può colorare con tre colori, assegnando il rosso ai nodi 1 e 4, il verde al 2 e il blu al 3. In generale, se un grafo contiene una *k-clique*, non possiamo usare meno di k colori (ma possono servirne più di k).

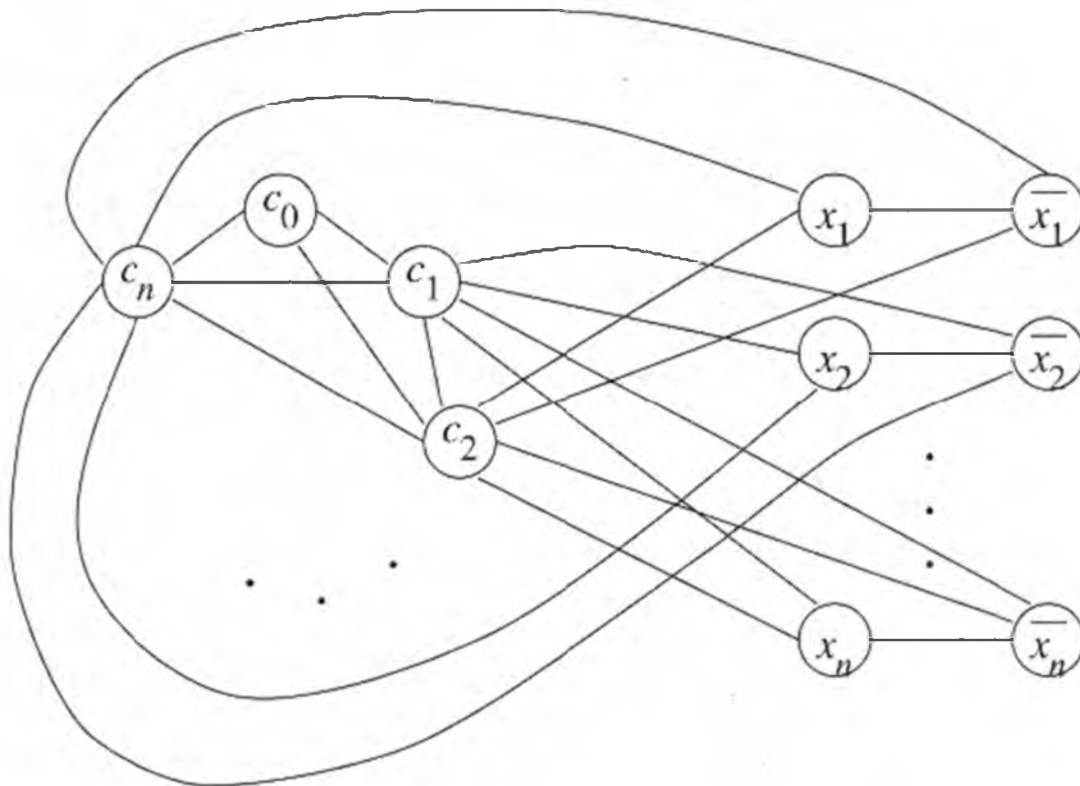


Figura 10.13 Una parte della costruzione che dimostra l'NP-completezza del problema di colorazione.

In questo esercizio spieghiamo una parte della costruzione per dimostrare che il problema della colorazione è NP-completo. Portatela a termine. La riduzione parte da 3SAT. Supponiamo di avere un'espressione in 3-CNF con n variabili. La riduzione trasforma l'espressione in un grafo, di cui una parte è illustrata nella Figura 10.13. Come si vede a sinistra, ci sono $n + 1$ nodi $c_0, c_1, \ldots, c_n$ che formano una $(n + 1)$-*clique*. Tutti questi nodi devono avere colori diversi. Chiameremo c_j il colore assegnato al nodo c_j.

Per ogni variabile x_i ci sono due nodi, corrispondenti a x_i e $\overline{x_i}$. Essi sono uniti da un lato, e non possono quindi avere lo stesso colore. Ogni nodo x_i è inoltre collegato a c_j per ogni j diverso da 0 e da i. Ne consegue che x_i o $\overline{x_i}$ deve avere il colore c_0, e l'altro il colore c_i. Si può pensare che quello colorato c_0 sia vero e l'altro falso. In questo modo la colorazione corrisponde a un assegnamento di valori di verità.

Per completare la costruzione dovete disegnare una parte di grafo per ogni clausola

dell'espressione. Dev'essere possibile colorare l'intero grafo usando solo i colori da c_0 a c_n se e solo se ogni clausola è vera rispetto all'assegnamento corrispondente alla scelta dei colori. Il grafo ricavato si può quindi colorare con $n + 1$ colori se e solo se l'espressione data è soddisfacibile.

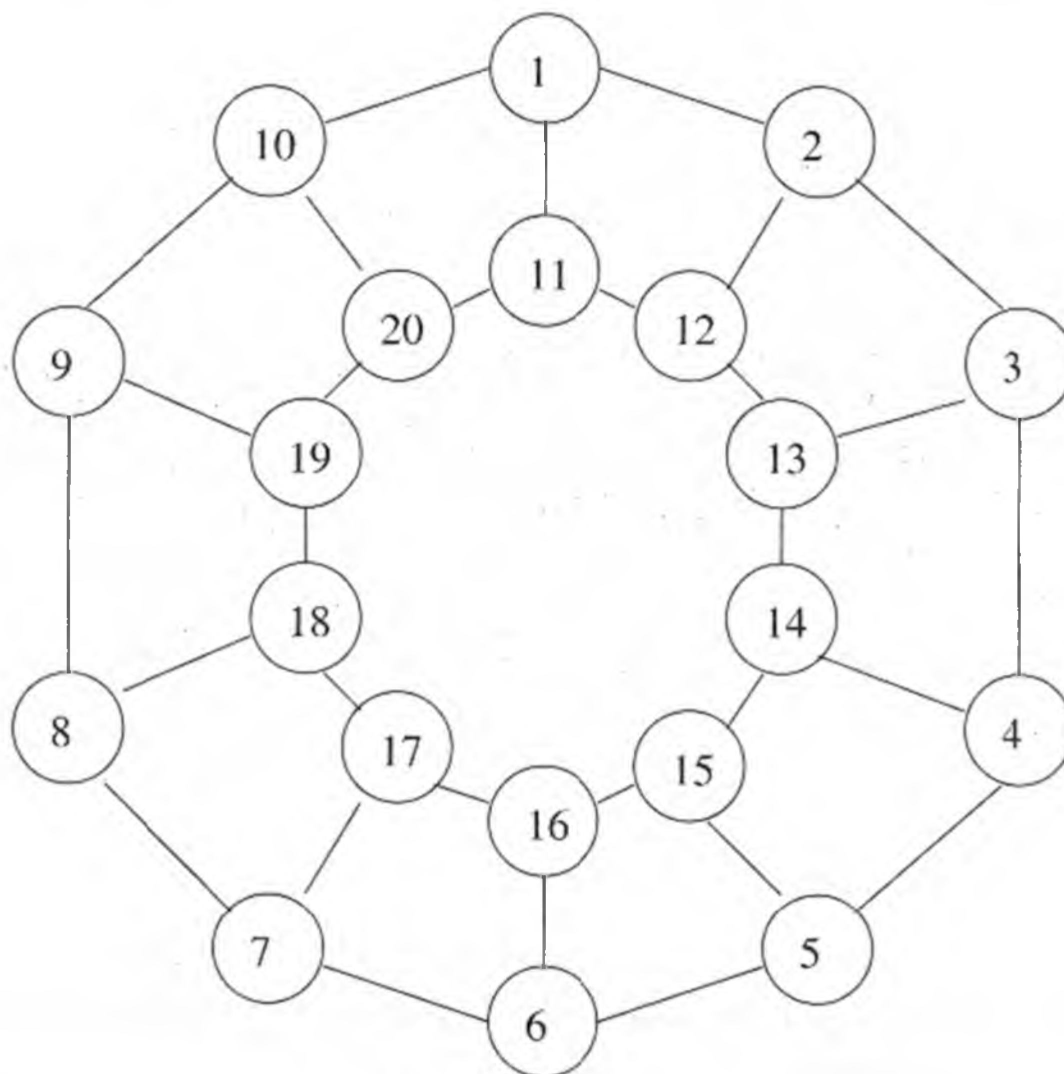


Figura 10.14 Un grafo.

! **Esercizio 10.4.3** Non occorre un grafo enorme per rendere molto difficile risolvere "a mano" questioni NP-complete. Considerate il grafo della Figura 10.14.

* a) Il grafo contiene un circuito hamiltoniano?

b) Qual è l'insieme indipendente più grande?

c) Qual è la più piccola copertura per nodi?

d) Qual è la più piccola copertura per lati (vedi Paragrafo 10.4.3)?

e) Si può colorare il grafo con due colori?

Esercizio 10.4.4 Dimostrate che i seguenti problemi sono NP-completi.

a) Il *problema dell'isomorfismo di grafi*: dati i grafi G_1 e G_2, G_1 contiene una copia di G_2 come sottografo? Possiamo cioè trovare un sottoinsieme dei nodi di G_1 che, insieme con i lati che li uniscono in G_1, forma una copia esatta di G_2, a patto di scegliere un'opportuna corrispondenza fra i nodi di G_2 e quelli del sottografo di G_1? *Suggerimento*: cercate una riduzione dal problema della *clique* descritto nell'Esercizio 10.4.1.

! b) Il *problema feedback edge*: dati un grafo G e un intero k, esiste un insieme di k lati di G tale che ogni ciclo di G contenga almeno uno di essi?

! c) Il *problema della programmazione lineare intera*: dato un insieme di vincoli lineari della forma $\sum_{i=1}^{n} a_i x_i \leq c$ o $\sum_{i=1}^{n} a_i x_i \geq c$, dove gli a_i e c sono costanti intere, e $x_1, x_2, \ldots, x_n$ sono variabili, esiste un assegnamento di valori interi alle variabili che verifica tutti i vincoli?

! d) Il *problema dell'insieme dominante*: dati un grafo G e un intero k, esiste un sottoinsieme S di k nodi di G tale che ogni nodo o è in S o è adiacente a un nodo in S?

e) Il *problema degli idranti*: dati un grafo G, una distanza d e un numero f di idranti, possiamo scegliere f nodi di G in modo che nessun nodo sia distante più di d da un idrante (la distanza è il numero di lati da percorrere)?

*! f) Il *problema della semi-clique*: dato un grafo G con un numero pari di vertici, esiste una *clique* (vedi Esercizio 10.4.1) formata da metà dei nodi di G? *Suggerimento*: riducete CLIQUE a questo problema. Dovete scoprire come aggiungere nodi in modo da regolare la dimensione della *clique* più grande.

!! g) Il *problema della pianificazione a tempi unitari*: dati k "compiti"

$$T_1, T_2, \ldots, T_k$$

un numero p di "processori", un "limite di tempo" t, e dei "vincoli di precedenza" della forma $T_i < T_j$ tra coppie di compiti, stabilite se esiste una pianificazione dei compiti tale che:

1. a ogni compito è assegnato un istante compreso fra 1 e t
2. lo stesso istante è assegnato al massimo a p compiti
3. i vincoli di precedenza sono rispettati, cioè a T_i si assegna un istante che precede quello assegnato a T_j se $T_i < T_j$.

!! h) Il *problema della copertura esatta*: dati un insieme S e una famiglia di sottoinsiemi di S, $S_1, S_2, \ldots, S_n$, esiste un'altra famiglia $T \subseteq \{S_1, S_2, \ldots, S_n\}$ tale che ogni elemento x di S si trovi esattamente in un membro di T?

!! i) Il *problema dello zaino*: data una lista di k interi $i_1, i_2, \ldots, i_k$, è possibile ripartirli in due insiemi le cui somme coincidono? *Nota*: a prima vista può sembrare che questo problema sia in $\mathcal{P}$, se si pensa che gli interi siano piccoli; in effetti, se i loro valori sono limitati da un polinomio in k, esiste un algoritmo in tempo polinomiale. In generale, però, in una lista di k interi rappresentati in codice binario, di lunghezza totale pari a n, possono esserci valori quasi esponenziali in n.

Esercizio 10.4.5 Un *cammino hamiltoniano* in un grafo G è una permutazione $n_1, n_2, \ldots, n_k$ di tutti i nodi tale che, per ogni $i = 1, 2, \ldots, k-1$, esiste un lato da n_i a n_{i+1}. Un *cammino hamiltoniano orientato* è lo stesso per grafi orientati: dev'esserci un arco da n_i a n_{i+1}. Osserviamo che questa definizione è solo leggermente più debole di quella di circuito hamiltoniano. Se si richiedesse anche l'esistenza di un lato o arco da n_k a n_1, le due definizioni coinciderebbero. Il problema del cammino hamiltoniano (orientato) chiede se un grafo (orientato) possiede almeno un cammino hamiltoniano (orientato).

* a) Dimostrate che il problema del cammino hamiltoniano orientato è NP-completo. *Suggerimento*: fate una riduzione da DHC. Scegliete un nodo e sdoppiatelo in modo che i due nodi risultanti siano gli estremi di un cammino hamiltoniano orientato e che quel cammino esista se e solo se il grafo originale ha un circuito hamiltoniano orientato.

b) Dimostrate che il problema del cammino hamiltoniano non orientato è NP-completo. *Suggerimento*: adattate la costruzione del Teorema 10.23.

*! c) Dimostrate che questo problema è NP-completo: dati un grafo G e un intero k, G possiede un albero di copertura con al massimo k foglie? *Suggerimento*: fate una riduzione dal problema del cammino hamiltoniano.

! d) Dimostrate che questo problema è NP-completo: dati un grafo G e un intero d, G possiede un albero di copertura senza nodi di grado maggiore di d (il *grado* di un nodo n nell'albero di copertura è il numero di lati dell'albero che hanno n a un estremo)?

10.5 Riepilogo

- *Le classi $\mathcal{P}$ ed $\mathcal{NP}$*: $\mathcal{P}$ è formata da tutti i linguaggi o problemi accettati da una macchina di Turing in tempo polinomiale rispetto alla lunghezza dell'input. $\mathcal{NP}$ è

la classe di linguaggi o problemi accettati da TM non deterministiche con un limite polinomiale sul tempo impiegato in una sequenza di scelte non deterministiche.

- *La questione $\mathcal{P} = \mathcal{NP}$*: non è noto se $\mathcal{P}$ ed $\mathcal{NP}$ siano la stessa classe di linguaggi oppure no, ma l'opinione più diffusa è che esistano linguaggi in $\mathcal{NP}$ che non sono in $\mathcal{P}$.

- *Riduzioni polinomiali*: se è possibile trasformare in tempo polinomiale istanze di un problema in istanze di un secondo problema che ha la medesima risposta, sì o no, diciamo che il primo problema è riducibile in tempo polinomiale al secondo.

- *Problemi NP-completi*: un linguaggio è NP-completo se è in $\mathcal{NP}$ ed esiste una riduzione polinomiale da ogni linguaggio in $\mathcal{NP}$ al linguaggio in questione. Siamo convinti che nessuno dei problemi NP-completi sia in $\mathcal{P}$. A suffragare questa tesi si può addurre il fatto che non è ancora stato trovato un algoritmo polinomiale per uno qualunque delle migliaia di problemi NP-completi conosciuti.

- *I problemi NP-completi di soddisfacibilità*: il teorema di Cook ha indicato il primo problema NP-completo, cioè se un'espressione booleana sia soddisfacibile, riducendo tutti i problemi in $\mathcal{NP}$ al problema SAT in tempo polinomiale. Il problema rimane NP-completo anche se si vincola l'espressione a consistere in un prodotto di clausole, ciascuna formata da tre soli letterali. Questo è il problema 3SAT.

- *Altri problemi NP-completi*: si conosce una vasta gamma di problemi NP-completi. Di ciascuno si dimostra l'NP-completezza tramite una riduzione polinomiale da un problema NP-completo già noto. Abbiamo fatto riduzioni per dimostrare NP-completi i seguenti problemi: insieme indipendente, copertura per nodi, versioni orientata e no del problema del circuito hamiltoniano, problema del commesso viaggiatore.

10.6 Bibliografia

Il concetto di NP-completezza come prova che un problema non può essere risolto in tempo polinomiale, così come la dimostrazione che SAT, CSAT e 3SAT sono NP-completi, si deve a Cook [3]. In genere si attribuisce uguale importanza a un successivo studio di Karp [6], il quale dimostrò che l'NP-completezza non è solo un fenomeno circoscritto, ma è comune a molti tra i più difficili problemi di combinatoria che erano stati studiati per anni nel campo della ricerca operativa e in altre discipline. Ogni problema di cui si è dimostrata l'NP-completezza nel Paragrafo 10.4 deriva da quel lavoro: insieme indipendente, copertura per nodi, circuito hamiltoniano e TSP. Vi si trovano inoltre svariati problemi citati negli esercizi: *clique*, *feedback edge*, zaino, colorazione e copertura esatta.

Il libro di Garey e Johnson [4] compendia una grande mole di materiale su quanto sappiamo circa i problemi NP-completi e casi speciali polinomiali. In [5] sono raccolti articoli su come approssimare la soluzione di un problema NP-completo in tempo polinomiale.

Altri contributi alla teoria dell'NP-completezza meritano un riconoscimento. Lo studio delle classi di linguaggi definite dal tempo di esecuzione di macchine di Turing prese le mosse da Hartmanis e Stearns [8]. Cobham [2] fu il primo a isolare il concetto di classe $\mathcal{P}$, in contrapposizione agli algoritmi con uno specifico tempo di esecuzione polinomiale, come $O(n^2)$. Di Levin [7] segnaliamo la scoperta indipendente, per quanto successiva, dell'idea di NP-completezza.

L'NP-completezza della programmazione lineare intera, citata nell'Esercizio 10.4.4(c), compare in [1], così come in note inedite di J. Gathen e M. Sieveking. L'NP-completezza della pianificazione a tempi di esecuzione unitari (Esercizio 10.4.4(g)) è discussa in [9].

1. I. Borosh, L. B. Treybig, "Bounds on positive integral solutions of linear Diophantine equations," *Proceedings of the AMS* **55** (1976), pp. 299–304.

2. A. Cobham, "The intrinsic computational difficulty of functions," *Proc. 1964 Congress for Logic, Mathematics, and the Philosophy of Science*, North Holland, Amsterdam, pp. 24–30.

3. S. C. Cook, "The complexity of theorem-proving procedures," *Third ACM Symposium on Theory of Computing* (1971), ACM, New York, pp. 151–158.

4. M. R. Garey, D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, H. Freeman, New York, 1979.

5. D. S. Hochbaum (ed.), *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Co., 1996.

6. R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations* (R. E. Miller, ed.), Plenum Press, New York, pp. 85–104, 1972.

7. L. A. Levin, "Universal sorting problems," *Problemi Peredachi Informatsii* **9**:3 (1973), pp. 115–116.

8. J. Hartmanis, R. E. Stearns, "On the computational complexity of algorithms," *Transactions of the AMS* **117** (1965), pp. 285–306.

9. J. D. Ullman, "NP-complete scheduling problems," *J. Computer and System Sciences* **10**:3 (1975), pp. 384–393.

Capitolo 11

Altre classi di problemi

La storia dei problemi intrattabili non si esaurisce con $\mathcal{NP}$. Molte altre classi di problemi sembrano intrattabili o sono interessanti per ragioni diverse. Alcune questioni che le riguardano, come la questione $\mathcal{P} = \mathcal{NP}$, rimangono irrisolte.

Cominceremo studiando una classe strettamente collegata a $\mathcal{P}$ ed $\mathcal{NP}$: la classe dei complementi dei linguaggi $\mathcal{NP}$, detta "co-$\mathcal{NP}$". Se $\mathcal{P} = \mathcal{NP}$, allora co-$\mathcal{NP}$ è uguale a entrambe perché $\mathcal{P}$ è chiusa rispetto alla complementazione. Si ritiene invece che co-$\mathcal{NP}$ sia diversa da ambedue e che nessun problema NP-completo sia in co-$\mathcal{NP}$.

Considereremo quindi la classe $\mathcal{PS}$, formata da tutti i problemi che possono essere risolti da una macchina di Turing impiegando una quantità di nastro polinomiale nella lunghezza dell'input. Queste TM possono richiedere tempo esponenziale, purché si muovano entro un'area limitata del nastro. Diversamente dal caso del tempo polinomiale, possiamo dimostrare che il non determinismo non aumenta la potenza delle TM quando ci si limita a uno spazio polinomiale. Ovviamente $\mathcal{PS}$ include $\mathcal{NP}$, ma non sappiamo se coincide con $\mathcal{NP}$ o con $\mathcal{P}$. Ci aspettiamo che non valga nessuna delle due uguaglianze, e descriveremo un problema completo per $\mathcal{PS}$ che sembra non essere in $\mathcal{NP}$.

Ci occuperemo poi degli algoritmi randomizzati e di due classi di linguaggi che si collocano tra $\mathcal{P}$ ed $\mathcal{NP}$. La prima è la classe $\mathcal{RP}$ dei linguaggi "polinomiali random", che hanno un algoritmo in tempo polinomiale basato sul "lancio della monetina", ossia, in pratica, su un generatore di numeri casuali. L'algoritmo conferma l'appartenenza dell'input al linguaggio oppure risponde "non so". Se l'input è nel linguaggio, c'è una probabilità maggiore di 0 che l'algoritmo risponda "sì". Eseguendolo più volte, possiamo decidere correttamente l'appartenenza con una probabilità tendente a 1.

Anche la seconda classe, detta $\mathcal{ZPP}$ (*Zero-error Probabilistic Polynomial*, polinomiale probabilistico con errore nullo), ricorre alla randomizzazione. Gli algoritmi per i linguaggi appartenenti a questa classe, però, dicono "sì, l'input è nel linguaggio" oppu-

re "no, non c'è". Il tempo di esecuzione previsto è polinomiale, ma ci possono essere esecuzioni che impiegano tempo superiore a qualsiasi limite polinomiale.

Per collegare fra loro questi concetti esaminiamo un problema importante: la verifica di primalità. Molti sistemi crittografici si basano su due elementi:

1. la capacità di scoprire velocemente grandi numeri primi (per permettere una comunicazione tra macchine non soggetta a intercettazione da parte di terzi)

2. l'ipotesi che ci voglia un tempo esponenziale per scomporre un numero intero in fattori primi, se il tempo viene misurato come funzione della lunghezza n dell'intero scritto in binario.

Vedremo che la verifica di primalità appartiene sia a $\mathcal{NP}$ sia a co-$\mathcal{NP}$[1], ed è perciò improbabile riuscire a dimostrare che è un problema NP-completo. È un peccato, perché le dimostrazioni di NP-completezza sono la prova più comune che un problema richiede con ogni probabilità un tempo esponenziale. Vedremo inoltre che la verifica di primalità è nella classe $\mathcal{RP}$. Si tratta di una circostanza fortunata, perché nella pratica i sistemi crittografici basati sui numeri primi usano effettivamente un algoritmo nella classe $\mathcal{RP}$ per trovarli. D'altronde è una circostanza negativa, perché dà ulteriore peso all'ipotesi che non saremo in grado di dimostrare la NP-completezza della verifica di primalità.

11.1 Complementi dei linguaggi in $\mathcal{NP}$

La classe dei linguaggi in $\mathcal{P}$ è chiusa rispetto alla complementazione (si veda l'Esercizio 10.1.6). Per dimostrarlo, supponiamo che L sia in $\mathcal{P}$ e che M sia una TM per L. Modifichiamo M in modo che accetti $\overline{L}$: introduciamo un nuovo stato accettante q e facciamo andare in q la nuova TM quando M si arresta in uno stato non accettante; rendiamo non accettanti gli stati accettanti di M. A questo punto la TM modificata accetta $\overline{L}$ impiegando lo stesso tempo di TM, con la possibile aggiunta di una mossa. Di conseguenza $\overline{L}$ è in $\mathcal{P}$ se lo è L.

Non è noto se $\mathcal{NP}$ sia chiusa rispetto alla complementazione. Si ritiene che non lo sia, e in particolare ci aspettiamo che, se un linguaggio L è NP-completo, il suo complemento non sia in $\mathcal{NP}$.

---

[1]Di recente è stata annunciata la scoperta che la verifica di primalità appartiene a $\mathcal{P}$ (si veda in proposito `http://www.cse.iitk.ac.in/users/manindra/index.html`). Questo nuovo risultato non invalida le considerazioni svolte qui e nel Paragrafo 11.5; conferma anzi l'ipotesi che il problema non sia NP-completo. In particolare non se ne può dedurre un algoritmo efficiente per fattorizzare numeri primi. [N.d.T.]

11.1.1 La classe di linguaggi co-$\mathcal{NP}$

Co-$\mathcal{NP}$ è l'insieme dei linguaggi i cui complementi sono in $\mathcal{NP}$. Abbiamo osservato all'inizio del Paragrafo 11.1 che il complemento di un linguaggio in $\mathcal{P}$ è in $\mathcal{P}$, e dunque in $\mathcal{NP}$. D'altra parte si pensa che nessun problema NP-completo abbia il complemento in $\mathcal{NP}$, e quindi che nessun problema NP-completo sia in co-$\mathcal{NP}$. Analogamente si ritiene che i complementi dei problemi NP-completi, che per definizione sono in co-$\mathcal{NP}$, non siano in $\mathcal{NP}$. La Figura 11.1 illustra le relazioni presunte fra le classi $\mathcal{P}$, $\mathcal{NP}$ e co-$\mathcal{NP}$. Dobbiamo però considerare che se $\mathcal{P}$ dovesse risultare uguale a $\mathcal{NP}$, tutte e tre le classi coinciderebbero.

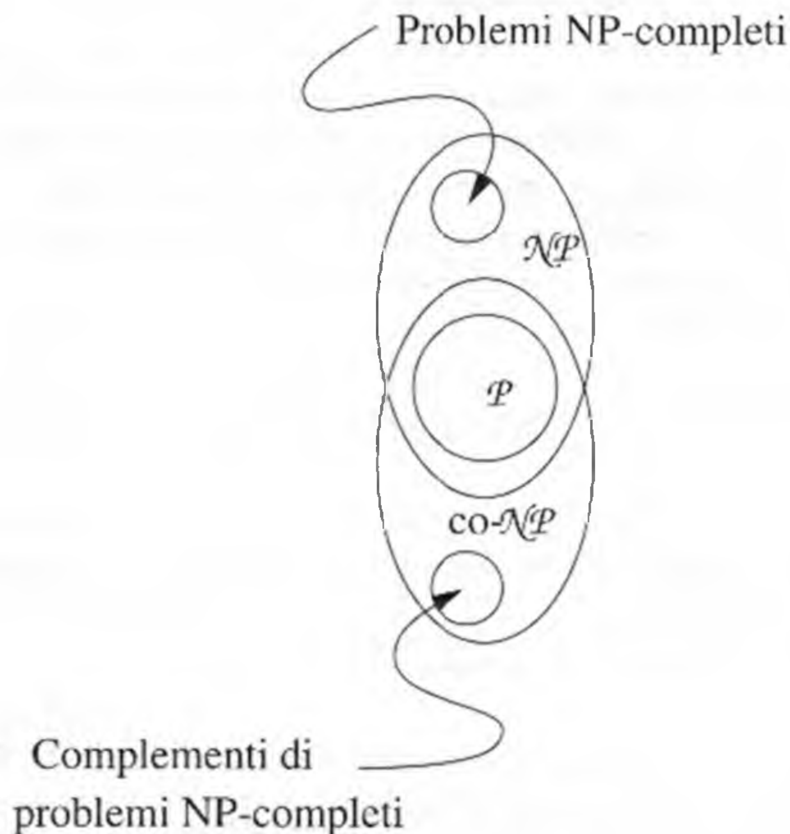


Figura 11.1 Le presunte relazioni tra co-$\mathcal{NP}$ e altre classi di linguaggi.

Esempio 11.1 Consideriamo il complemento del linguaggio SAT, che è senz'altro un membro di co-$\mathcal{NP}$ e che denoteremo con *USAT* (da *unsatisfiable*, non soddisfacibile). USAT comprende tutte le stringhe che codificano espressioni booleane non soddisfacibili. In USAT, però, ci sono anche le stringhe che non codificano espressioni booleane valide perché nessuna di esse è in SAT. Si pensa che USAT non sia in $\mathcal{NP}$, ma non esiste alcuna dimostrazione in merito.

Un altro esempio di problema che supponiamo essere in co-$\mathcal{NP}$ ma non in $\mathcal{NP}$ è TAUT, l'insieme di tutte le espressioni booleane (codificate) che sono *tautologie*, cioè

sono vere per ogni assegnamento di valori di verità. Osserviamo che un'espressione E è una tautologia se e solo se $\neg E$ è insoddisfacibile. Di conseguenza TAUT e USAT sono collegate: se un'espressione booleana E è in TAUT, $\neg E$ è in USAT, e viceversa. Osserviamo che USAT contiene anche stringhe che non rappresentano espressioni valide, mentre tutte le stringhe in TAUT sono espressioni valide. □

11.1.2 Problemi NP-completi e co-$\mathcal{NP}$

Supponiamo che $\mathcal{P} \neq \mathcal{NP}$. È ancora possibile che la situazione di co-$\mathcal{NP}$ non sia quella illustrata dalla Figura 11.1 perché $\mathcal{NP}$ e co-$\mathcal{NP}$ potrebbero essere uguali, ma più grandi di $\mathcal{P}$. In altre parole potremmo scoprire che i problemi come USAT e TAUT possono essere risolti in tempo polinomiale non deterministico (cioè sono in $\mathcal{NP}$), ma non in tempo polinomiale deterministico. Il fatto che non siamo riusciti a trovare anche solo un problema NP-completo il cui complemento sia in $\mathcal{NP}$ è tuttavia un indizio forte che $\mathcal{NP} \neq$ co-$\mathcal{NP}$, come proveremo nel prossimo teorema.

Teorema 11.2 $\mathcal{NP} =$ co-$\mathcal{NP}$ se e solo se esiste un problema NP-completo il cui complemento è in $\mathcal{NP}$.

DIMOSTRAZIONE (Solo se) Se $\mathcal{NP}$ e co-$\mathcal{NP}$ sono uguali, ogni problema NP-completo L, trovandosi in $\mathcal{NP}$, è anche in co-$\mathcal{NP}$. Dato che il complemento di un problema in co-$\mathcal{NP}$ è in $\mathcal{NP}$, il complemento di L è in $\mathcal{NP}$.

(Se) Sia P un problema NP-completo il cui complemento $\overline{P}$ è in $\mathcal{NP}$. Per ogni linguaggio L in $\mathcal{NP}$ esiste dunque una riduzione polinomiale di L a P. La stessa riduzione è anche una riduzione polinomiale di $\overline{L}$ a $\overline{P}$. Dimostriamo che $\mathcal{NP} =$ co-$\mathcal{NP}$ provando che ciascuna è contenuta nell'altra.

$\mathcal{NP} \subseteq$ co-$\mathcal{NP}$: supponiamo che L sia in $\mathcal{NP}$. Allora $\overline{L}$ è in co-$\mathcal{NP}$. Combiniamo la riduzione polinomiale di $\overline{L}$ a $\overline{P}$ con il presunto algoritmo polinomiale non deterministico per $\overline{P}$ per produrre un algoritmo polinomiale non deterministico per $\overline{L}$. Ne consegue che per ogni L in $\mathcal{NP}$ anche $\overline{L}$ è in $\mathcal{NP}$. D'altra parte $\overline{L}$ è in co-$\mathcal{NP}$ perché è il complemento di un linguaggio in $\mathcal{NP}$. Abbiamo dunque $\mathcal{NP} \subseteq$ co-$\mathcal{NP}$.

co-$\mathcal{NP} \subseteq \mathcal{NP}$: supponiamo che L sia in co-$\mathcal{NP}$. Esiste allora una riduzione polinomiale di $\overline{L}$ a P perché P è NP-completo ed $\overline{L}$ è in $\mathcal{NP}$. Essa è contemporaneamente anche una riduzione di L a $\overline{P}$. Poiché $\overline{P}$ è in $\mathcal{NP}$, combiniamo la riduzione con l'algoritmo polinomiale non deterministico per $\overline{P}$ per dimostrare che L è in $\mathcal{NP}$. □

11.1.3 Esercizi

! **Esercizio 11.1.1** Dite se i seguenti problemi sono in $\mathcal{NP}$ e in co-$\mathcal{NP}$, e descrivete il complemento di ognuno di essi. Se il problema o il suo complemento sono NP-completi, dimostratelo.

* a) Il problema TRUE-SAT: data un'espressione booleana E, vera quando tutte le variabili hanno valore vero, esiste un altro assegnamento di valori di verità che la rende vera?

b) Il problema FALSE-SAT: data un'espressione booleana E, falsa quando tutte le sue variabili hanno valore falso, esiste un altro assegnamento di valori di verità che la rende falsa?

c) Il problema DOUBLE-SAT: data un'espressione booleana E, esistono almeno due assegnamenti di valori di verità che la rendono vera?

d) Il problema NEAR-TAUT: data un'espressione booleana E, esiste al massimo un assegnamento di valori di verità che la rende falsa?

!! **Esercizio 11.1.2** Supponiamo che esista una funzione f, biunivoca sull'insieme degli interi a n-bit, tale che:

1. $f(x)$ può essere computata in tempo polinomiale
2. $f^{-1}(x)$ non può essere computata in tempo polinomiale.

Dimostrate che il linguaggio formato da coppie di interi (x, y) tale che

$$f^{-1}(x) < y$$

sarebbe in $(\mathcal{NP} \cap \text{co-}\mathcal{NP}) - \mathcal{P}$.

11.2 Problemi risolvibili in spazio polinomiale

Esaminiamo ora una classe di problemi che include NP e che si ritiene sia più ampia, anche se non è certo. Questa classe è definita ammettendo che una macchina di Turing usi uno spazio polinomiale nella dimensione dell'input, a prescindere dal tempo. Inizialmente distingueremo tra i linguaggi accettati da TM deterministiche e quelli accettati da TM non deterministiche con un limite polinomiale sullo spazio, ma vedremo presto che le due classi di linguaggi in realtà coincidono.

Esistono problemi P completi rispetto allo spazio polinomiale, nel senso che tutti i problemi in questa classe sono riducibili a P in *tempo* polinomiale. Di conseguenza, se P è in $\mathcal{P}$ o in $\mathcal{NP}$, tutti i linguaggi accettati da TM con limite polinomiale sullo spazio sono rispettivamente in $\mathcal{P}$ o $\mathcal{NP}$. Daremo un esempio di questi problemi: le "formule booleane con quantificatori".

11.2.1 Macchine di Turing a spazio polinomiale

La Figura 11.2 illustra una macchina di Turing con limite polinomiale sullo spazio. Esiste un polinomio $p(n)$ tale che la TM, a fronte di un input w di lunghezza n, non visita mai più di $p(n)$ celle del nastro. Per il Teorema 8.12 possiamo presumere che il nastro sia semi-infinito e che la TM non si sposti mai a sinistra della prima cella di input.

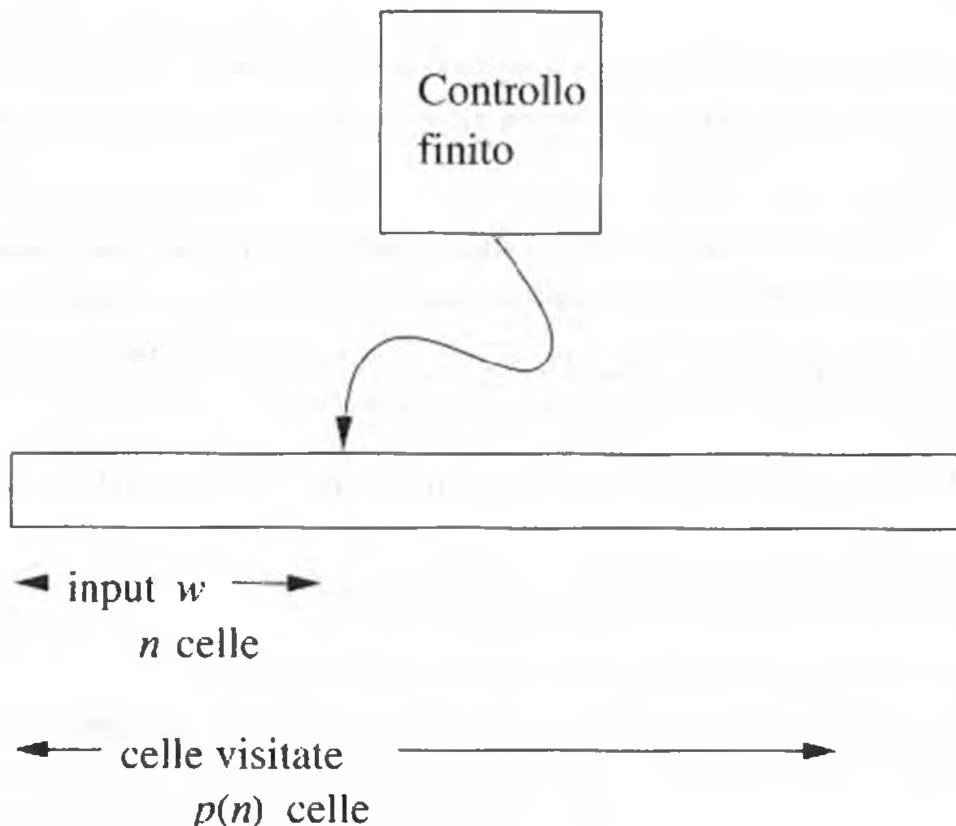


Figura 11.2 Una TM a spazio polinomiale.

Definiamo $\mathcal{PS}$ (*Polynomial Space*, spazio polinomiale) come la classe di tutti e soli i linguaggi accettati da macchine di Turing deterministiche con limite polinomiale sullo spazio. Definiamo inoltre $\mathcal{NPS}$ (*Nondeterministic Polynomial Space*, spazio polinomiale non deterministico) come la classe formata dai linguaggi accettati da TM non deterministiche con limite polinomiale sullo spazio. Evidentemente $\mathcal{PS} \subseteq \mathcal{NPS}$ perché ogni TM deterministica è un caso speciale di TM non deterministica. Dimostreremo il risultato sorprendente che $\mathcal{PS} = \mathcal{NPS}$.[2]

[2]Questa classe è talvolta denominata PSPACE. Dato che non useremo più $\mathcal{NPS}$ dopo aver dimostrato l'equivalenza $\mathcal{PS} = \mathcal{NPS}$, noi preferiamo invece la sigla $\mathcal{PS}$ per denotare la classe di problemi risolti in spazio polinomiale deterministico (o non deterministico).

11.2.2 Le relazioni di $\mathcal{PS}$ ed $\mathcal{NPS}$ con altre classi

Innanzitutto le relazioni $\mathcal{P} \subseteq \mathcal{PS}$ e $\mathcal{NP} \subseteq \mathcal{NPS}$ dovrebbero essere evidenti. La ragione è che se una TM fa solo un numero polinomiale di mosse, allora usa non più di un numero polinomiale di celle; in particolare il numero massimo di celle che può visitare corrisponde al numero di mosse più uno. Una volta dimostrato $\mathcal{PS} = \mathcal{NPS}$, vedremo che effettivamente le tre classi formano una catena rispetto al contenimento: $\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PS}$.

Una proprietà essenziale delle TM con limite polinomiale sullo spazio è che possono fare solo un numero esponenziale di mosse prima di dover ripetere una ID. Questa proprietà ci serve per dimostrare altre proprietà interessanti di $\mathcal{PS}$, e per dimostrare che $\mathcal{PS}$ contiene solo linguaggi ricorsivi, cioè linguaggi con algoritmi. Osserviamo che nella definizione di $\mathcal{PS}$ o $\mathcal{NPS}$ non si richiede che la TM si arresti. È possibile che la TM compia cicli perpetui senza lasciare un'area del nastro di dimensione polinomiale.

Teorema 11.3 Se M è una TM (deterministica o non deterministica) con limite polinomiale sullo spazio, e $p(n)$ è il suo limite polinomiale sullo spazio, allora esiste una costante c tale che, se M accetta un input w di lunghezza n, lo fa entro $c^{1+p(n)}$ mosse.

DIMOSTRAZIONE Il concetto di fondo è che M deve ripetere una ID prima di superare le $c^{1+p(n)}$ mosse. Se M ripete una ID e poi accetta, deve esistere una sequenza più breve di ID che conduce all'accettazione. In altre parole, se $\alpha \overset{*}{\vdash} \beta \overset{*}{\vdash} \beta \overset{*}{\vdash} \gamma$, dove α è la ID iniziale, β è la ID ripetuta e γ è la ID accettante, allora $\alpha \overset{*}{\vdash} \beta \overset{*}{\vdash} \gamma$ è una sequenza più breve di ID che conduce all'accettazione.

Il ragionamento per cui c deve esistere poggia sul fatto che, se lo spazio usato dalla TM è limitato, c'è solo un numero limitato di ID. In particolare, sia t il numero di simboli di nastro di M e sia s il numero di stati di M. Se si usano solo $p(n)$ celle di nastro, il numero di ID diverse di M è al massimo $sp(n)t^{p(n)}$. Possiamo infatti scegliere uno degli s stati, collocare la testina su una qualunque delle $p(n)$ posizioni di nastro, e riempire le $p(n)$ celle con una qualsiasi delle $t^{p(n)}$ sequenze di simboli di nastro.

Poniamo $c = s + t$ e consideriamo l'espansione binomiale di $(t + s)^{1+p(n)}$:

$$t^{1+p(n)} + \big(1 + p(n)\big)st^{p(n)} + \cdots$$

Osserviamo che il secondo termine è almeno tanto grande quanto $sp(n)t^{p(n)}$, il che dimostra che $c^{1+p(n)}$ è almeno uguale al numero di ID possibili di M. Concludiamo la dimostrazione osservando che se M accetta w di lunghezza n, lo fa con una sequenza di mosse che non ripete una ID. Perciò M accetta con una sequenza di mosse che non è più lunga del numero di ID distinte, che è $c^{1+p(n)}$. □

Possiamo ricorrere al Teorema 11.3 per convertire una TM con limite polinomiale sullo spazio in una equivalente che si arresta sempre dopo aver fatto al massimo un numero esponenziale di mosse. Infatti, sapendo che la TM accetta entro un numero esponenziale

di mosse, possiamo contare quante mosse sono state fatte e arrestare la TM se ha fatto un numero sufficiente di mosse senza accettare.

Teorema 11.4 Se L è un linguaggio in $\mathcal{PS}$ (rispettivamente $\mathcal{NPS}$), allora L è accettato da una TM deterministica (rispettivamente non deterministica), con limite polinomiale sullo spazio, che si arresta dopo aver fatto al più $c^{q(n)}$ mosse, per un polinomio $q(n)$ e una costante $c > 1$.

DIMOSTRAZIONE Dimostreremo l'enunciato per le TM deterministiche; lo stesso ragionamento vale per le NTM. Sappiamo che L è accettato da una TM M_1 che ha un limite polinomiale $p(n)$ sullo spazio. Allora, per il Teorema 11.3, se M_1 accetta w lo fa in non più di $c^{1+p(|w|)}$ passi.

Definiamo una nuova TM M_2 dotata di due nastri. Sul primo nastro M_2 simula M_1, sul secondo conta in base c fino a $c^{1+p(|w|)}$. Se M_2 raggiunge questo numero, si arresta senza accettare. Perciò M_2 usa $1 + p(|w|)$ celle sul secondo nastro. Abbiamo ipotizzato che M_1 non impieghi più di $p(|w|)$ celle, per cui anche M_2 non usa più di $p(|w|)$ celle sul primo nastro.

Se convertiamo M_2 in una TM M_3 a nastro singolo, possiamo essere certi che M_3 non usa più di $1 + p(n)$ celle di nastro, su qualsiasi input di lunghezza n. M_3 può impiegare il quadrato del tempo di esecuzione di M_2, ma non più di $O\big(c^{2p(n)}\big)$.[3] Dato che M_3 non compie più di $dc^{2p(n)}$ mosse per una costante d, possiamo scegliere $q(n) = 2p(n) + \log_c d$. M_3 fa dunque al massimo $c^{q(n)}$ passi. Poiché M_2 si arresta sempre, anche M_3 fa lo stesso. Dal momento che M_1 accetta L, lo fanno anche M_2 ed M_3. Di conseguenza M_3 soddisfa l'enunciato del teorema. □

11.2.3 Spazio polinomiale deterministico e non deterministico

Se il confronto tra $\mathcal{P}$ ed $\mathcal{NP}$ sembra difficile, sorprende che lo stesso confronto tra $\mathcal{PS}$ ed $\mathcal{NPS}$ sia invece facile: queste due classi di linguaggi infatti coincidono. La dimostrazione consiste nel simulare una TM non deterministica con un limite polinomiale sullo spazio $p(n)$ mediante una TM deterministica con limite polinomiale sullo spazio $O\big(p^2(n)\big)$.

Il nocciolo della prova consiste nel verificare, in modo ricorsivo e deterministico, se una NTM N può passare dalla ID I alla ID J in non più di m mosse. Una DTM D tenta sistematicamente tutte le ID intermedie K per controllare se da I può passare a K in $m/2$ mosse e da K a J in altre $m/2$ mosse. Supponiamo cioè che esista una funzione ricorsiva $reach(I, J, m)$ che decide se $I \overset{*}{\vdash} J$ in m mosse al massimo.

---

[3] In effetti la regola generale desunta dal Teorema 8.10 non è l'affermazione più forte che possiamo fare. Dato che qualsiasi nastro usa solo $1 + p(n)$ celle, le testine simulate nella conversione da multinastro a mononastro possono allontanarsi fra loro al massimo di $1 + p(n)$ celle. Di conseguenza $c^{1+p(n)}$ mosse della TM M_2 multinastro possono essere simulate in $O\Big(p(n)c^{p(n)}\Big)$ passi, che è meno di $O\Big(c^{2p(n)}\Big)$, come abbiamo affermato prima.

Trattiamo il nastro di D come uno stack in cui sono collocati gli argomenti delle chiamate ricorsive di *reach*. Uno *stack frame*[4] D contiene $[I, J, m]$. La Figura 11.3 presenta uno schema dell'algoritmo eseguito da *reach*.

```
BOOLEAN FUNCTION reach(I,J,m)
    ID: I,J; INT: m;
    BEGIN
        IF (m == 1) THEN /* base */ BEGIN
            verifica se I == J o
                     se I può diventare J in una mossa;
            RETURN TRUE in caso positivo,
                   FALSE altrimenti;
        END;
        ELSE /* passo induttivo */ BEGIN
            FOR ogni ID K DO
                IF (reach(I,K,m/2) AND
                    reach(K,J,m/2)) THEN
                    RETURN TRUE;
            RETURN FALSE;
        END;
    END;
```

Figura 11.3 La funzione ricorsiva *reach* verifica se da una ID si può passare a un'altra entro un numero fissato di mosse.

È importante osservare che *reach*, sebbene chiami se stessa due volte, compie le chiamate in sequenza, e dunque in ogni istante solo una di esse è attiva. Così, se partiamo da uno stack frame $[I_1, J_1, m]$, a ogni istante c'è solo una chiamata $[I_2, J_2, m/2]$, una $[I_3, J_3, m/4]$, una $[I_4, J_4, m/8]$, e così via, finché a un certo punto il terzo argomento diventa 1 e *reach* può applicare il passo di base senza ulteriori chiamate ricorsive. *reach* verifica se $I = J$ oppure $I \vdash J$, restituendo `TRUE` se almeno una delle due è vera e `FALSE` in caso contrario. Dato un valore iniziale m del numero di mosse, la Figura 11.4 illustra lo stack della DTM D con tutte le possibili chiamate attive a *reach*.

Se può sembrare che siano possibili molte chiamate a *reach* e che il nastro della Figura 11.4 possa diventare molto lungo, dimostreremo che non potrà essere "troppo lungo". Se si parte da un numero di mosse pari a m, sul nastro ci possono essere a ogni istante solo $\log_2 m$ stack frame. Poiché il Teorema 11.4 garantisce che la NTM N non può fare più di $c^{p(n)}$ mosse, m può avere un valore iniziale non maggiore. Di conseguenza il numero di

[4]Per *stack frame* intendiamo un'area dello stack destinata ai dati di una chiamata di funzione.

I_1 J_1 m	I_2 J_2 $m/2$	I_3 J_3 $m/4$	I_4 J_4 $m/8$	$\cdots$

Figura 11.4 Il nastro di una DTM che simula una NTM mediante chiamate ricorsive a *reach*.

stack frame è al più $\log_2 c^{p(n)}$, che è $O(p(n))$. Disponiamo ora degli elementi essenziali su cui basare la dimostrazione del seguente teorema.

Teorema 11.5 (*Teorema di Savitch*) $\mathcal{PS} = \mathcal{NPS}$.

DIMOSTRAZIONE È evidente che $\mathcal{PS} \subseteq \mathcal{NPS}$, dato che ogni DTM è anche una NTM. Dobbiamo perciò dimostrare soltanto che $\mathcal{NPS} \subseteq \mathcal{PS}$, cioè che, se L è accettato da una NTM N con limite polinomiale sullo spazio $p(n)$, per un polinomio $p(n)$, allora L è accettato anche da una DTM D con limite polinomiale sullo spazio $q(n)$, per un altro polinomio $q(n)$. In effetti dimostreremo che $q(n)$ può essere scelto di un ordine di grandezza pari al quadrato di $p(n)$.

Per il Teorema 11.3 possiamo assumere innanzitutto che se N accetta, lo fa in non più di $c^{1+p(n)}$ passi per una costante c. Dato un input w di lunghezza n, D scopre come si comporta N a fronte di un input w ponendo ripetutamente sul nastro la tripla $[I_0, J, m]$ e chiamando *reach* con questi argomenti, dove:

1. I_0 è la ID iniziale di N con input w
2. J è una qualsiasi ID accettante che usa al più $p(n)$ celle di nastro; le diverse J sono enumerate sistematicamente da D mediante un nastro ausiliario
3. $m = c^{1+p(n)}$.

Abbiamo già dimostrato che non ci sono mai più di $\log_2 m$ chiamate ricorsive attive nello stesso momento, cioè una con terzo argomento m, una con $m/2$, una con $m/4$, e così via, fino a 1. Di conseguenza non ci sono più di $\log_2 m$ frame sullo stack, e $\log_2 m$ è $O(p(n))$.

Gli stack frame occupano ciascuno uno spazio $O(p(n))$. Infatti ognuna delle due ID richiede di scrivere solo $1 + p(n)$ celle, e se si scrive m in binario abbiamo bisogno di $\log_2 c^{1+p(n)} = O(p(n))$ celle. Di conseguenza l'intero stack frame, formato dalle due ID e da un intero, occupa uno spazio $O(p(n))$.

Poiché D ha al massimo $O(p(n))$ stack frame, lo spazio complessivo è $O(p^2(n))$. Questo spazio è un polinomio se lo è $p(n)$, e possiamo concludere che L ha una DTM con limite polinomiale sullo spazio. □

In conclusione possiamo aggiungere le classi a spazio polinomiale al quadro delle relazioni fra classi di complessità. Il diagramma completo è illustrato nella Figura 11.5.

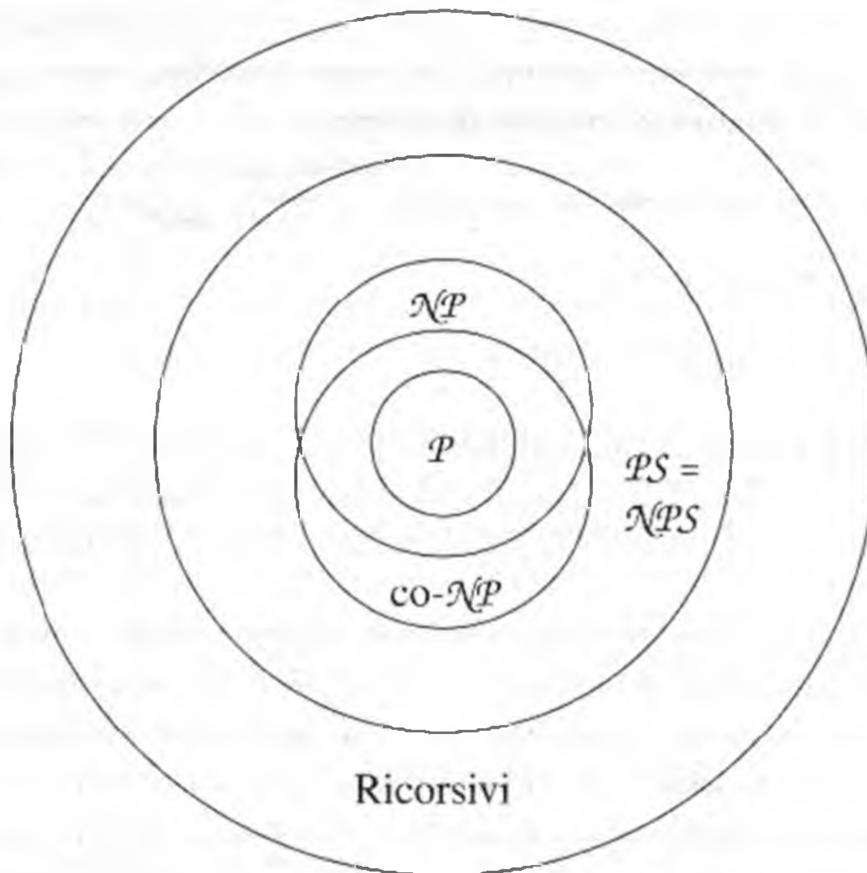


Figura 11.5 Relazioni note tra classi di linguaggi.

11.3 Un problema completo per $\mathcal{PS}$

In questo paragrafo presentiamo il problema delle "formule booleane con quantificatori" e dimostriamo che è completo per $\mathcal{PS}$.

11.3.1 PS-completezza

Diremo che un problema P è *completo per $\mathcal{PS}$* (PS-completo) se:

1. P è in $\mathcal{PS}$
2. ogni linguaggio L in $\mathcal{PS}$ è riducibile a P in tempo polinomiale.

Si noti che, sebbene ci occupiamo di spazio polinomiale (e non di tempo), la condizione per la PS-completezza è analoga a quella per la NP-completezza: la riduzione dev'essere compiuta in tempo polinomiale. La ragione è che vogliamo sapere che $\mathcal{P} = \mathcal{PS}$ se un problema PS-completo dovesse risultare in $\mathcal{P}$, e che $\mathcal{NP} = \mathcal{PS}$ se un problema PS-completo è in $\mathcal{NP}$. Se la riduzione richiedesse solo uno spazio polinomiale, la

dimensione dell'output potrebbe essere esponenziale nella dimensione dell'input, e non potremmo dimostrare il prossimo teorema. Dal momento che consideriamo riduzioni in tempo polinomiale, otteniamo le relazioni desiderate.

Teorema 11.6 Sia P un problema PS-completo.

a) Se P è in $\mathcal{P}$, allora $\mathcal{P} = \mathcal{PS}$.

b) Se P è in $\mathcal{NP}$, allora $\mathcal{NP} = \mathcal{PS}$.

DIMOSTRAZIONE Dimostriamo (a). Sappiamo che per ogni L in $\mathcal{PS}$ esiste una riduzione in tempo polinomiale di L a P. Sia $q(n)$ il tempo impiegato dalla riduzione. Supponiamo inoltre che P sia in $\mathcal{P}$, e quindi abbia un algoritmo che richiede un tempo polinomiale $p(n)$.

Data una stringa w, di cui vogliamo verificare l'appartenenza a L, possiamo servirci della riduzione per convertirla in una stringa x che è in P se e solo se w è in L. Poiché la riduzione impiega un tempo $q(|w|)$, la stringa x non può essere più lunga di $q(|w|)$. Possiamo verificare l'appartenenza di x a P in tempo $p(|x|)$, che è $p(q(|w|))$, un polinomio in $|w|$. Concludiamo che esiste un algoritmo in tempo polinomiale per L.

Ciascun linguaggio L in $\mathcal{PS}$ è quindi anche in $\mathcal{P}$. Poiché $\mathcal{P}$ è contenuto in $\mathcal{PS}$, deduciamo che se P è in $\mathcal{P}$, allora $\mathcal{P} = \mathcal{PS}$. La dimostrazione di (b), dove p è in $\mathcal{NP}$, è del tutto simile ed è lasciata al lettore. □

11.3.2 Formule booleane con quantificatori

Presentiamo ora un problema P completo per $\mathcal{PS}$. Prima però dobbiamo chiarire i termini in cui si definisce questo problema, detto "formule booleane con quantificatori" (QBF, *Quantified Boolean Formulas*).

Una formula booleana con quantificatori è un'espressione booleana con l'aggiunta degli operatori $\forall$ ("per ogni") ed $\exists$ ("esiste"). L'espressione $(\forall x)(E)$ significa che E è vera quando tutte le occorrenze di x in E sono sostituite da 1 (vero), e che è altrettanto vera quando tutte le occorrenze di x sono sostituite da 0 (falso). L'espressione $(\exists x)(E)$ significa che E è vera quando tutte le occorrenze di x sono sostituite da 1 o quando tutte le occorrenze di x sono sostituite da 0, oppure in entrambi i casi.

Per semplificare la descrizione assumiamo che nessuna QBF contenga due o più *quantificatori* ($\forall$ o $\exists$) per la stessa variabile x. Questa restrizione non è essenziale e corrisponde approssimativamente a impedire che due diverse funzioni di un programma possano usare la stessa variabile locale.[5] Le *formule booleane con quantificatori* sono definite formalmente come segue.

[5]Nei programmi e nelle formule booleane con quantificatori è sempre possibile rinominare uno dei due usi distinti dello stesso nome della variabile. Per i programmi non c'è alcuna ragione di evitare il reimpiego dello stesso nome locale, ma nelle QBF è opportuno imporre che ciò non accada.

1. 0 (false), 1 (true), e una qualsiasi variabile sono QBF.

2. Se E ed F sono QBF, lo sono anche (E), $\neg(E)$, $(E) \wedge (F)$ e $(E) \vee (F)$, che rappresentano una E tra parentesi, la negazione di E, l'AND di E ed F, e l'OR di E ed F. Le parentesi superflue possono essere eliminate secondo le regole di precedenza consuete: NOT, poi AND, infine OR. Di norma seguiremo lo stile "aritmetico" per AND e OR, dove AND è rappresentato dalla giustapposizione (nessun operatore) e OR è rappresentato dal $+$. Quindi scriveremo spesso $(E)(F)$ in luogo di $(E) \wedge (F)$, e $(E) + (F)$ in luogo di $(E) \vee (F)$.

3. Se F è una QBF senza quantificatori per la variabile x, allora $(\forall x)(E)$ e $(\exists x)(E)$ sono QBF. Diremo che la *portata* (*scope*) di x è l'espressione E. Intuitivamente x è definita solo entro E, così come la portata di una variabile in un programma si estende alla funzione in cui è dichiarata. Le parentesi prima e dopo E (ma non prima e dopo il quantificatore) possono essere eliminate se non c'è ambiguità. Per evitare un eccesso di parentesi le une dentro le altre, scriveremo una catena di quantificatori come

$$(\forall x)\Big((\exists y)\big((\forall z)(E)\big)\Big)$$

con la sola coppia attorno a E, omettendo quelle relative a ogni quantificatore lungo la catena, in questo modo: $(\forall x)(\exists y)(\forall z)(E)$.

Esempio 11.7 Ecco un esempio di QBF:

$$(\forall x)\big((\exists y)(xy) + (\forall z)(\neg x + z)\big) \tag{11.1}$$

Partiamo dalle variabili x e y, le connettiamo con AND e applichiamo il quantificatore $(\exists y)$ per produrre la sottoespressione $(\exists y)(xy)$. Analogamente costruiamo l'espressione booleana $\neg x + z$ e applichiamo il quantificatore $(\forall z)$ ottenendo la sottoespressione $(\forall z)(\neg x + z)$. Combiniamo poi le due espressioni con OR; non servono parentesi perché + (OR) ha la precedenza più bassa. Infine applichiamo il quantificatore $(\forall x)$ per produrre la QBF finale. $\square$

11.3.3 Valutazione di formule booleane con quantificatori

Dobbiamo ancora definire in termini formali il significato di una QBF. Se però leggiamo $\forall$ come "per ogni" ed $\exists$ come "esiste", possiamo farcene un'idea. La QBF dell'Esempio 11.7 afferma che per ogni x (cioè $x = 0$ o $x = 1$) esiste un y tale che sia x sia y sono vere, oppure per ogni z, $\neg x + z$ è vera. Per dimostrare che l'enunciato è valido, osserviamo che se $x = 1$ possiamo scegliere $y = 1$ e rendere vera xy. Se $x = 0$, allora $\neg x + z$ è vera per ambedue i valori di z.

Se una variabile x si trova nella portata di un quantificatore per x, diremo che l'occorrenza di x è *vincolata*. Negli altri casi diremo che è *libera*.

Esempio 11.8 Nella QBF dell'equazione (11.1) ogni occorrenza di ciascuna variabile è vincolata, in quanto si trova nella portata di un quantificatore per la variabile. Per esempio la portata della variabile y quantificata in $(\exists y)(xy)$ è l'espressione xy. Dunque in questo caso l'occorrenza di y è vincolata. L'uso di x in xy è legato al quantificatore $(\forall x)$, la cui portata è l'intera espressione. □

Il valore di una QBF che non ha variabili libere è 0 oppure 1 (cioè vero o falso). Possiamo calcolare il valore della QBF per induzione sulla lunghezza n dell'espressione.

BASE Se l'espressione ha lunghezza 1, può essere solo una costante 0 o 1, perché se fosse una variabile sarebbe libera. Il valore dell'espressione coincide con l'espressione stessa.

INDUZIONE Supponiamo di avere un'espressione senza variabili libere di lunghezza $n > 1$ e di saper valutare una qualsiasi espressione di lunghezza inferiore, purché non abbia variabili libere. Una QBF di questo tipo può avere sei forme.

1. L'espressione è della forma (E). Quindi E è di lunghezza $n - 2$ e siamo in grado di valutarla. Il suo valore è 0 o 1. Il valore di (E) è lo stesso.

2. L'espressione è della forma $\neg E$. Quindi E è di lunghezza $n - 1$ e può essere valutata. Se $E = 1$, allora $\neg E = 0$, e viceversa.

3. L'espressione è della forma EF. Allora sia E sia F sono più brevi di n, per cui possono essere valutate. Il valore di EF è 1 se sia E sia F hanno valore 1, e 0 se una delle due vale 0.

4. L'espressione è della forma $E + F$. Allora sia E sia F sono più brevi di n, per cui possono essere valutate. Il valore di $E + F$ è 1 se E oppure F ha valore 1, e 0 se entrambe sono 0.

5. Se l'espressione è della forma $(\forall x)(E)$, in primo luogo sostituiamo tutte le occorrenze di x in E con 0, e otteniamo l'espressione E_0; sostituiamo poi ogni occorrenza di x in E con 1, ottenendo l'espressione E_1. Osserviamo che E_0 ed E_1 soddisfano due condizioni.

   (a) Non hanno variabili libere perché un'occorrenza libera di una variabile in E_0 o E_1 non può essere x, e sarebbe quindi libera in E.

   (b) Hanno lunghezza $n - 6$, e sono quindi più brevi di n.

   Valutiamo E_0 ed E_1. Se entrambe hanno valore 1, allora $(\forall x)(E)$ ha valore 1; altrimenti ha valore 0. Osserviamo che la regola rispecchia l'interpretazione "per ogni x" di $(\forall x)$.

6. Se l'espressione è $(\exists x)(E)$, procediamo come nel punto (5), costruendo E_0 ed E_1 e valutandole. Se E_0 o E_1 ha valore 1, allora $(\exists x)(E)$ ha valore 1; altrimenti ha valore 0. Osserviamo che la regola rispecchia l'interpretazione "esiste x" di $(\exists x)$.

Esempio 11.9 Valutiamo la QBF dell'Equazione (11.1). Essa è della forma $(\forall x)(E)$. Dobbiamo quindi valutare prima E_0, che è:

$$(\exists y)(0y) + (\forall z)(\neg 0 + z) \tag{11.2}$$

Il valore dell'espressione dipende dai valori delle due espressioni legate dall'OR: $(\exists y)(0y)$ e $(\forall z)(\neg 0 + z)$; E_0 ha valore 1 se una delle due espressione vale 1. Per valutare $(\exists y)(0y)$ dobbiamo sostituire $y = 0$ e $y = 1$ nella sottoespressione $0y$, e controllare che almeno una delle sottoespressioni così ottenute abbia il valore 1. Sia $0 \wedge 0$ sia $0 \wedge 1$ hanno valore 0, per cui $(\exists y)(0y)$ ha valore 0.[6]

Fortunatamente $(\forall z)(\neg 0 + z)$ ha valore 1, come si vede sostituendo prima $z = 0$ e poi $z = 1$. Poiché $\neg 0 = 1$, le due espressioni che dobbiamo valutare sono $1 \vee 0$ e $1 \vee 1$. Avendo entrambe valore 1, sappiamo che $(\forall z)(\neg 0 + z)$ ha valore 1. Possiamo ora concludere che E_0, cioè l'Equazione (11.2), ha valore 1.

Dobbiamo inoltre verificare che anche E_1, ottenuta sostituendo $x = 1$ nell'Equazione (11.1), valga 1:

$$(\exists y)(1y) + (\forall z)(\neg 1 + z) \tag{11.3}$$

L'Espressione $(\exists y)(1y)$ ha valore 1, come si vede sostituendo $y = 1$. Di conseguenza E_1, ovvero l'Equazione (11.3), ha valore 1. Concludiamo che l'intera espressione, cioè l'Equazione (11.1), ha valore 1. □

11.3.4 PS-completezza del problema QBF

Possiamo ora definire il *problema delle formule booleane con quantificatori*: data una QBF E priva di variabili libere, E vale 1? Denoteremo questo problema con QBF, e continueremo a impiegare lo stesso acronimo anche per indicare "formula booleana con quantificatori". Il contesto dovrebbe eliminare ogni ambiguità.

Dimostreremo che il problema QBF è completo per $\mathcal{PS}$. La dimostrazione combina idee tratte dai teoremi 10.9 e 11.5. Dal Teorema 10.9 prendiamo l'idea di rappresentare la computazione di una TM tramite variabili logiche, ognuna delle quali indica se una certa cella ha un certo valore in un dato momento. Nel caso di tempo polinomiale, però, come nel Teorema 10.9, il numero di variabili rilevanti è polinomiale. Siamo stati dunque

[6]Abbiamo fatto uso della notazione alternativa per AND e OR perché non possiamo utilizzare la giustapposizione e + per le espressioni che coinvolgono 0 e 1 senza rendere le espressioni simili a numeri di più cifre o a una somma aritmetica. Confidiamo che il lettore accetti entrambe le notazioni come rappresentanti degli stessi operatori logici.

in grado di generare, in tempo polinomiale, un'espressione indicante che la TM accetta il suo input. Quando trattiamo un limite polinomiale sullo spazio, il numero delle ID in una computazione può essere esponenziale nella dimensione dell'input, e non possiamo scrivere in tempo polinomiale un'espressione booleana per indicare la correttezza della computazione. Fortunatamente abbiamo un linguaggio più espressivo, e la disponibilità dei quantificatori ci permette di scrivere una QBF di lunghezza polinomiale che indica che la TM con limite polinomiale sullo spazio accetta il suo input.

Dal Teorema 11.5 prendiamo la tecnica ricorsiva per esprimere il fatto che una ID può diventarne un'altra in un numero elevato di mosse. In altre parole, per dire che la ID I può diventare la ID J in m mosse, diciamo che esiste una ID K tale che I diventa K in $m/2$ mosse e K diventa J in altre $m/2$ mosse. Il linguaggio delle formule booleane con quantificatori ci permette di dirlo con un'espressione di lunghezza polinomiale, anche se m è esponenziale nella lunghezza dell'input.

Prima di procedere alla dimostrazione che ogni linguaggio in $\mathcal{PS}$ è riducibile in tempo polinomiale a QBF, dobbiamo dimostrare che QBF è in $\mathcal{PS}$. Anche questa parte della dimostrazione della PS-completezza non è immediata: la isoliamo come teorema a sé stante.

Teorema 11.10 QBF è in $\mathcal{PS}$.

DIMOSTRAZIONE Nel Paragrafo 11.3.3 abbiamo descritto una procedura ricorsiva per valutare una QBF F. Possiamo implementarla per mezzo di uno stack memorizzato sul nastro di una macchina di Turing, come abbiamo fatto nella dimostrazione del Teorema 11.5. Supponiamo che F sia di lunghezza n. Creiamo un record di lunghezza $O(n)$ per F, che contiene F stessa e lo spazio per la sottoespressione di F che stiamo trattando. Due esempi presi dalle sei forme possibili di F chiariranno il processo di valutazione.

1. Supponiamo che $F = F_1 + F_2$. Operiamo allora come segue.

   (a) Poniamo F_1 in un record a sé, a destra di quello per F.

   (b) Valutiamo ricorsivamente F_1.

   (c) Se il valore di F_1 è 1, restituiamo il valore 1 per F.

   (d) Se il valore di F_1 è 0, sostituiamo il suo record con un record per F_2 e valutiamo ricorsivamente F_2.

   (e) Restituiamo come valore di F il valore restituito da F_2.

2. Supponiamo che $F = (\exists x)(E)$. Operiamo allora come segue.

   (a) Formiamo l'espressione E_0 sostituendo 0 a ogni occorrenza di x e collochiamo E_0 in un suo record, a destra di quello per F.

   (b) Valutiamo ricorsivamente E_0.

(c) Se il valore di E_0 è 1, restituiamo 1 come valore di F.

(d) Se il valore di E_0 è 0, formiamo E_1 sostituendo 1 a x in E.

(e) Sostituiamo il record per E_0 con un record per E_1, e valutiamo ricorsivamente E_1.

(f) Restituiamo come valore di F il valore restituito da E_1.

Lasciamo al lettore il compito di descrivere i passi da compiere nei casi in cui F ha una delle altre quattro possibili forme: $F_1 F_2$, $\neg E$, (E) oppure $(\forall x)(E)$. Il caso di base, in cui F è una costante, ci obbliga a restituire quella costante senza formare altri record.

Osserviamo che in ogni caso a destra del record per un'espressione di lunghezza m ce ne sarà uno per un'espressione di lunghezza inferiore a m. Anche se dobbiamo spesso valutare due sottoespressioni diverse, le valutiamo una per volta. Pertanto nel caso (1) non abbiamo mai simultaneamente sul nastro il record per F_1, o per una sua sottoespressione, e quello per F_2, o per una sua sottoespressione. Lo stesso vale per E_0 ed E_1 nel caso (2).

Perciò, se partiamo con un'espressione di lunghezza n, sullo stack non ci possono mai essere più di n record. Inoltre la lunghezza di ogni record è $O(n)$. Quindi l'intero nastro non diventa mai più lungo di $O(n^2)$. Abbiamo ora una costruzione per una TM con limite polinomiale sullo spazio che accetta QBF; il suo limite in spazio è quadratico. Osserviamo che quest'algoritmo impiega di norma un tempo esponenziale in n, per cui *non* è polinomiale in tempo. □

Passiamo ora alla riduzione da un linguaggio arbitrario L in $\mathcal{PS}$ al problema QBF. Vorremmo usare variabili proposizionali y_{ijA} come nel Teorema 10.9 per asserire che la j-esima posizione nell'i-esima ID è A. Però, dato che esiste un numero esponenziale di ID, per un input w di lunghezza n non potremmo neppure scrivere quelle variabili in tempo polinomiale in n. Sfruttiamo invece i quantificatori per far sì che lo stesso insieme di variabili rappresenti molte ID diverse. L'idea è sviluppata nella dimostrazione che segue.

Teorema 11.11 Il problema QBF è PS-completo.

DIMOSTRAZIONE Sia L in $\mathcal{PS}$, accettato da una TM deterministica M che usa al massimo uno spazio $p(n)$ su input di lunghezza n. Per il Teorema 11.3 sappiamo che esiste una costante c tale che M accetta in non più di $c^{1+p(n)}$ mosse un input di lunghezza n. Descriveremo ora come, a partire da un input w di lunghezza n, si possa costruire in tempo polinomiale una QBF E senza variabili libere e con valore 1 se e solo se w è in $L(M)$.

Per formare E dovremo introdurre un numero polinomiale di *ID-variabili*, cioè insiemi di variabili del tipo y_{jA}; y_{jA} indica che la j-esima posizione della ID rappresentata ha il simbolo A, con j compreso fra 0 e $p(n)$. Il simbolo A è un simbolo di nastro oppure uno stato di M. Di conseguenza il numero di variabili proposizionali in una ID-variabile è

polinomiale in n. Assumiamo che tutte le variabili proposizionali in diverse ID-variabili siano distinte, ossia che nessuna variabile proposizionale appartenga a due diverse ID-variabili. Se il numero di ID-variabili è polinomiale, anche il numero totale di variabili proposizionali è polinomiale.

Conviene servirsi della notazione $(\exists I)$, dove I è una ID-variabile, per denotare $(\exists x_1)(\exists x_2)\cdots(\exists x_m)$, dove $x_1, x_2, \ldots, x_m$ sono tutte le variabili proposizionali nella ID-variabile I. Analogamente $(\forall I)$ sta per il quantificatore $\forall$ applicato a tutte le variabili proposizionali in I.

La QBF formata da w ha la forma

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F)$$

dove:

1. I_0 e I_f sono ID-variabili che rappresentano rispettivamente la ID iniziale e quella finale

2. S è un'espressione che indica "avvio corretto", cioè I_0 è proprio la ID iniziale di M con input w

3. N è un'espressione che indica "mossa lecita", cioè M passa da I_0 a I_f

4. F è un'espressione che indica "terminazione corretta", cioè I_f è una ID accettante.

Osserviamo che, mentre l'intera espressione non ha variabili libere, le variabili di I_0 sono libere in S, quelle di I_f sono libere in F, ed entrambi i gruppi di variabili sono liberi in N.

Avvio corretto

S è l'AND logico di letterali; ogni letterale è una delle variabili di I_0. S ha il letterale y_{jA} se la j-esima posizione della ID iniziale con input w è A, altrimenti ha il letterale $\overline{y_{jA}}$. Quindi, se $w = a_1 a_2 \cdots a_n$, allora $y_{0q_0}, y_{1a_1}, y_{2a_2}, \ldots, y_{na_n}$ e tutte le y_{jB}, per $j = n+1, n+2, \ldots, p(n)$, compaiono senza negazione, mentre tutte le altre variabili di I_0 sono negate. Assumiamo che q_0 sia lo stato iniziale di M e B sia il suo blank.

Terminazione corretta

Per essere una ID accettante, I_f deve avere uno stato accettante. Scriviamo perciò F come l'OR logico delle variabili y_{jA}, scelte tra le variabili proposizionali di I_f per le quali A è uno stato accettante. La posizione j è arbitraria.

Una costruzione inefficiente

A prima vista si sarebbe tentati di costruire N_{2i} da N_i secondo una strategia *divide et impera*: se $I \overset{*}{\vdash} J$ in $2i$ o meno mosse, allora deve esistere una ID K per cui in i mosse o meno si ha $I \overset{*}{\vdash} K$ e $K \overset{*}{\vdash} J$. Se però scriviamo la formula che esprime questo concetto, per esempio $N_{2i}(I, J) = (\exists K)\big(N_i(I, K) \wedge N_i(K, J)\big)$, la lunghezza dell'espressione raddoppia, come i. Per esprimere tutte le possibili computazioni di M, i dev'essere esponenziale in n. Impiegheremmo quindi troppo tempo per scrivere N, che risulterebbe di lunghezza esponenziale.

Mossa lecita

L'espressione N è costruita ricorsivamente in modo da raddoppiare il numero di mosse considerate aggiungendo solo $O\big(p(n)\big)$ simboli all'espressione in costruzione e impiegando solo un tempo $O\big(p(n)\big)$ per scrivere l'espressione. Definiamo l'abbreviazione $I = J$, dove I e J sono ID-variabili, come AND logico delle espressioni che uguagliano le variabili corrispondenti di I e J. In altre parole, se I consiste nelle variabili y_{jA} e J nelle variabili z_{jA}, allora $I = J$ è l'AND delle espressioni $\big(y_{jA}z_{jA} + (\overline{y_{jA}})(\overline{z_{jA}})\big)$, con j che varia da 0 a $p(n)$, e con A che è un simbolo di nastro o uno stato di M.

Costruiamo ora le espressioni $N_i(I, J)$, per $i = 1, 2, 4, 8, \cdots$ a indicare che $I \overset{*}{\vdash} J$ in i o meno mosse. In queste espressioni solo le variabili proposizionali delle ID-variabili I e J sono libere; tutte le altre sono vincolate.

BASE Per $i = 1$, $N_i(I, J)$ afferma che $I = J$ oppure $I \vdash J$. Abbiamo appena visto come esprimere la condizione $I = J$. Per la condizione $I \vdash J$ rinviamo il lettore alla discussione nella sezione "Mossa successiva lecita" della dimostrazione del Teorema 10.9, dove viene trattato lo stesso problema (una ID segue da quella che la precede). L'espressione N_1 è l'OR logico di queste due espressioni. Osserviamo che è possibile scrivere N_1 in tempo $O\big(p(n)\big)$.

INDUZIONE Costruiamo $N_{2i}(I, J)$ a partire da N_i. Nel riquadro "Una costruzione inefficiente" sottolineiamo che la soluzione immediata, con l'uso di due copie di N_i per costruire N_{2i}, non rispetta i limiti di spazio e di tempo necessari. Il modo corretto di scrivere N_{2i} impiega una sola copia di N_i nell'espressione, passando entrambi gli argomenti (I, K) e (K, J) alla stessa espressione. In altre parole $N_{2i}(I, J)$ userà un'unica sottoespressione $N_i(P, Q)$. Scriviamo $N_{2i}(I, J)$ per affermare che esiste una ID K tale che per tutte le ID P e Q vale una delle seguenti condizioni:

1. $(P, Q) \neq (I, K)$ e $(P, Q) \neq (K, J)$

2. $N_i(P, Q)$ è vera.

In altri termini $N_i(I, K)$ e $N_i(K, J)$ sono vere e non ci interessa sapere se $N_i(P, Q)$ è vera per altri motivi. Ecco una QBF per $N_{2i}(I, J)$:

$$\begin{aligned} N_{2i}(I, J) \quad = \quad & (\exists K)(\forall P)(\forall Q)\Big(N_i(P, Q) \ \vee \\ & (\neg(I = P \wedge K = Q) \wedge \neg(K = P \wedge J = Q))\Big) \end{aligned}$$

Osserviamo che è possibile scrivere N_{2i} nello stesso tempo che occorre per scrivere N_i, più un tempo aggiuntivo $O\big(p(n)\big)$.

Per completare la costruzione di N, dobbiamo definire N_m per il più piccolo m che sia una potenza di 2 e che sia maggiore o uguale a $c^{1+p(n)}$, il numero più elevato possibile di mosse che la TM M può fare prima di accettare un input w di lunghezza n. Dobbiamo applicare il passo induttivo descritto sopra per un numero di volte pari a $\log_2(c^{1+p(n)})$, o $O\big(p(n)\big)$. Poiché ciascun uso del passo induttivo impiega tempo $O\big(p(n)\big)$, concludiamo che N può essere costruita in tempo $O\big(p^2(n)\big)$.

Conclusione della dimostrazione del Teorema 11.11

Abbiamo illustrato come trasformare l'input w in una QBF

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F)$$

in tempo polinomiale in $|w|$. Inoltre abbiamo spiegato perché ognuna delle espressioni S, N ed F è vera se e solo se le sue variabili libere rappresentano le ID I_0 e I_f, che sono rispettivamente la ID iniziale e quella accettante di una computazione di M su input w con $I_0 \overset{*}{\vdash} I_f$. Quindi questa QBF ha valore 1 se e solo se M accetta w. □

11.3.5 Esercizi

Esercizio 11.3.1 Completate la dimostrazione del Teorema 11.10 trattando i seguenti casi:

a) $F = F_1 F_2$

b) $F = (\forall x)(E)$

c) $F = \neg(E)$

d) $F = (E)$.

*!! **Esercizio 11.3.2** Dimostrate che il seguente problema è PS-completo: data un'espressione regolare E, E è equivalente a Σ^*, dove Σ è l'insieme di simboli che compaiono in E? *Suggerimento*: invece di ridurre QBF a questo problema, può risultare più facile dimostrare la riduzione al problema di un linguaggio arbitrario in $\mathcal{PS}$. Per ogni TM con limite polinomiale sullo spazio, dimostrate che, dato un input w, si può costruire in tempo polinomiale un'espressione regolare che genera tutte le stringhe che *non* sono sequenze di ID della TM che conducono all'accettazione di w.

!! **Esercizio 11.3.3** Il gioco detto *Shannon Switching Game* si svolge su un grafo G con due nodi terminali s e t, tra due giocatori, che chiameremo SHORT e CUT. Ogni giocatore, a partire da SHORT, si alterna nella selezione di un vertice di G, diverso da s e t, che rimane in suo possesso fino alla fine della partita. SHORT vince se seleziona un insieme di nodi che, con s e t, forma un cammino in G da s a t. CUT vince se tutti i nodi sono stati selezionati e SHORT non ha completato un cammino da s a t. Dimostrate che il seguente problema è PS-completo: dato G, SHORT può vincere indipendentemente dalle scelte di CUT?

11.4 Classi di linguaggi basate sulla randomizzazione

Passiamo ora a due classi di linguaggi definiti per mezzo di macchine di Turing che possono servirsi di numeri casuali nella computazione. I linguaggi di programmazione più diffusi consentono di scrivere algoritmi che sfruttano un generatore di numeri casuali. La funzione `rand()`, o una funzione dal nome simile, che restituisce un numero apparentemente "casuale" o impredicibile, esegue in realtà un algoritmo specifico che può essere ripetuto, anche se è molto difficile rilevare uno schema nella sequenza di numeri prodotti. Un semplice esempio (che in pratica non si usa) può consistere nell'elevare al quadrato l'intero precedente nella sequenza e prendere i bit mediani del risultato. I numeri prodotti da una tale procedura meccanica sono detti *pseudo casuali*.

In questo paragrafo definiremo un tipo di macchina di Turing che modella la generazione di numeri casuali e il loro impiego negli algoritmi. Definiremo poi due classi di linguaggi, $\mathcal{RP}$ e $\mathcal{ZPP}$, che si servono in modi diversi di questo genere di casualità e di un vincolo polinomiale sul tempo. È interessante osservare che le due classi sembrano includere poco più di $\mathcal{P}$, ma le differenze sono importanti. In particolare vedremo nel Paragrafo 11.5 che alcune delle questioni principali sulla sicurezza informatica riguardano in realtà la relazione di queste classi con $\mathcal{P}$ ed $\mathcal{NP}$.

11.4.1 Quicksort: un esempio di algoritmo randomizzato

"Quicksort" è un noto algoritmo di ordinamento, di cui descriviamo in sintesi il funzionamento. Data una sequenza di elementi $a_1, a_2, \ldots, a_n$ da ordinare, ne scegliamo uno, per

esempio a_1, e dividiamo la sequenza in due parti: gli elementi pari ad a_1 o più piccoli e gli elementi più grandi. L'elemento selezionato è detto *pivot*. Scegliendo un'opportuna rappresentazione dei dati, possiamo ripartire in tempo $O(n)$ una sequenza di lunghezza n in due sequenze le cui lunghezze sommate danno ancora n. Possiamo poi ordinare separatamente, in modo ricorsivo, la sequenza degli elementi piccoli (minori o uguali al pivot) e quella degli elementi grandi (maggiori del pivot); il risultato sarà una sequenza ordinata di tutti gli n elementi.

Con un po' di fortuna il pivot risulterà essere un numero a metà della sequenza ordinata. La lunghezza delle sottoliste sarà allora vicina a $n/2$. Se a ogni passo siamo fortunati, dopo circa $\log_2 n$ livelli di ricorsione avremo sequenze di lunghezza 1, ovviamente già ordinate. Di conseguenza il lavoro totale si compone di $O(\log n)$ livelli, ciascuno di costo $O(n)$, per un tempo complessivo $O(n \log n)$.

Non è detto però che tutto vada così liscio. Per esempio, se la sequenza è ordinata in partenza, la scelta del primo elemento produce una sottosequenza di un solo elemento con gli elementi restanti nella sottosequenza degli elementi grandi. In questo caso Quicksort si comporta in modo molto simile all'ordinamento per selezione e impiega un tempo proporzionale a n^2 per ordinare gli n elementi.

Perciò le migliori implementazioni del Quicksort non scelgono il pivot in una posizione specifica della sequenza, ma in una posizione casuale. Ognuno degli n elementi ha quindi probabilità $1/n$ di essere scelto come pivot.[7] Il tempo medio di esecuzione del Quicksort randomizzato è $O(n \log n)$. Ma poiché rimane una probabilità, per quanto piccola, che ogni scelta del pivot prenda l'elemento più grande o il più piccolo, il tempo di esecuzione del Quicksort nel caso peggiore è ancora $O(n^2)$. Nonostante ciò, Quicksort è il metodo preferito in molte applicazioni (per esempio viene usato nel comando `sort` di UNIX) perché il suo tempo medio di esecuzione è migliore rispetto ad altre soluzioni, anche in confronto a quelle che nel caso peggiore sono $O(n \log n)$.

11.4.2 Un modello di macchina di Turing con randomizzazione

Per rappresentare in termini astratti in una macchina di Turing la capacità di compiere scelte casuali in modo analogo a un programma che chiama una o più volte un generatore di numeri casuali, ricorriamo alla variante di una TM multinastro illustrata nella Figura 11.6. Il primo nastro contiene l'input, secondo la prassi per le TM multinastro. Anche il secondo nastro contiene inizialmente simboli diversi dal blank. L'intero nastro è in effetti occupato da 0 e 1, ciascuno scelto a caso e in modo indipendente; la probabilità di scegliere 0 è 1/2, e lo stesso vale per 1. Il secondo nastro sarà denominato *nastro casuale*. Il terzo nastro e i successivi, se vengono usati, sono inizialmente bianchi

[7] Non forniremo la prova di questa affermazione. Per una dimostrazione e un'analisi del tempo medio di esecuzione per Quicksort, rimandiamo a D. E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, 1973.

e hanno la funzione di "nastri ausiliari". Chiameremo questo modello *macchina di Turing randomizzata*.

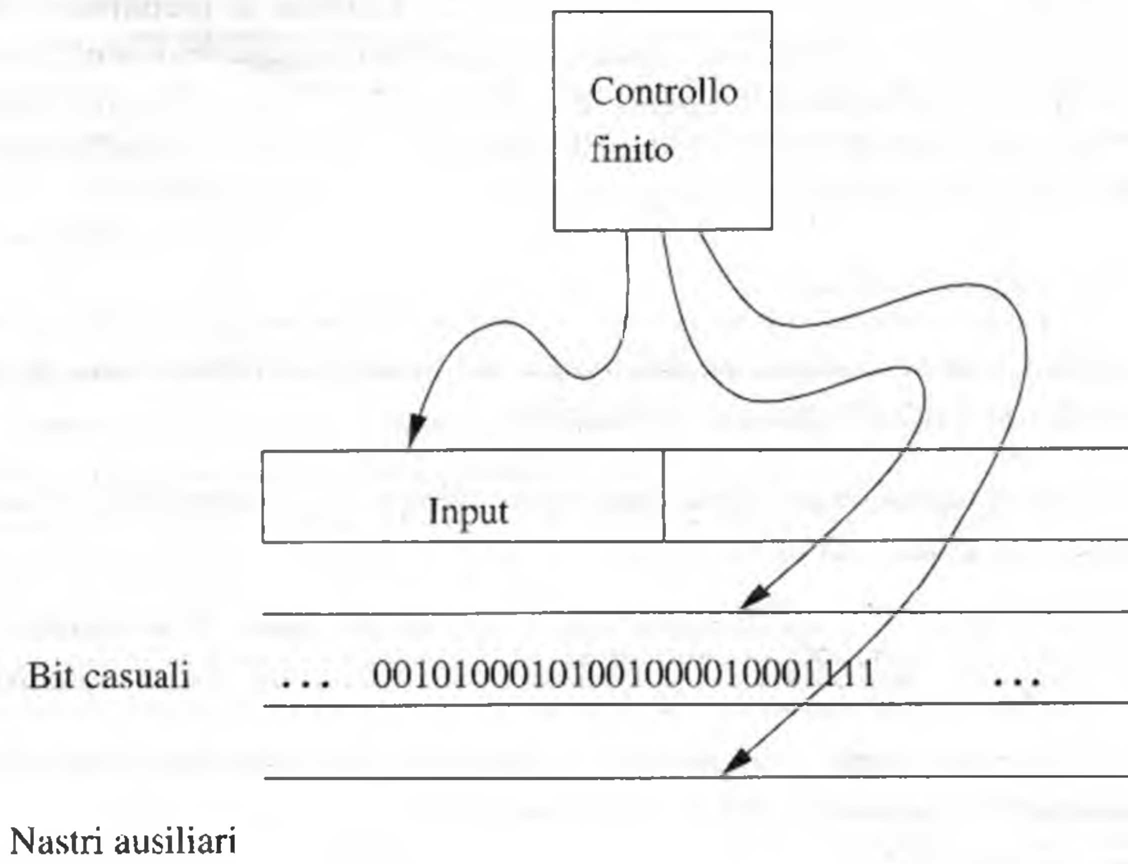


Figura 11.6 Una macchina di Turing in grado di usare numeri "generati" casualmente.

Se l'idea di preparare la TM con un numero infinito di 0 e 1 scelti a caso non pare realistica, si può pensare a una TM equivalente il cui secondo nastro sia inizialmente bianco. Quando la seconda testina guarda un blank, ha luogo un "lancio di monetina" interno e la TM scrive immediatamente uno 0 o un 1 sulla cella, che vi rimane per sempre. In questo modo non si compie alcun lavoro prima di avviare la TM randomizzata, tanto meno un lavoro infinito, ma il secondo nastro sembra coperto di 0 e 1 scelti a caso perché i bit casuali compaiono dovunque si trovi la testina del secondo nastro.

Esempio 11.12 Possiamo implementare la versione randomizzata del Quicksort su una TM randomizzata. Il punto cruciale è il processo ricorsivo in cui si prende una sottosequenza (che ipotizziamo memorizzata in celle consecutive sul nastro di input e delimitata da marcatori alle due estremità), si sceglie a caso un pivot e si divide la sottosequenza in due sottosequenze. Descriviamo il succedersi delle operazioni.

1. Supponiamo che la sottosequenza da dividere abbia lunghezza m. Per scegliere un numero casuale tra 1 ed m occorrono circa $O(\log m)$ nuovi bit casuali sul secondo

nastro; l'm-esimo elemento della sottosequenza diventa il pivot. Osserviamo che non sempre siamo in grado di scegliere ogni intero tra 1 ed m con la stessa probabilità, perché m può non essere una potenza di 2. Tuttavia, se prendiamo $\lceil 2\log_2 m \rceil$ bit dal nastro 2, li consideriamo come un numero compreso fra 0 e circa m^2, prendiamo il resto della divisione per m e aggiungiamo 1, avremo tutti i numeri tra 1 ed m con una probabilità di scelta abbastanza vicina a $1/m$, come richiesto per il funzionamento corretto del Quicksort.

2. Copiamo il pivot sul nastro 3.

3. Percorriamo la sottosequenza delimitata sul nastro 1, copiando sul nastro 4 gli elementi che non sono più grandi del pivot.

4. Percorriamo nuovamente la sottosequenza sul nastro 1, copiando sul nastro 5 gli elementi più grandi del pivot.

5. Copiamo il nastro 4 e poi il nastro 5 nello spazio sul nastro 1 che prima conteneva la sottosequenza delimitata. Collochiamo un marcatore tra le due sequenze.

6. Se una delle due sotto-sottosequenze, o entrambe, ha più di un elemento, la (o le) ordiniamo ricorsivamente con lo stesso algoritmo.

Osserviamo che questa implementazione del Quicksort impiega un tempo $O(n \log n)$, anche se il dispositivo di calcolo è una TM multinastro e non un computer convenzionale. L'esempio è interessante non tanto per il tempo di esecuzione quanto per l'impiego dei bit casuali sul secondo nastro per generare il comportamento casuale della macchina di Turing. □

11.4.3 Il linguaggio di una macchina di Turing randomizzata

Di solito consideriamo il linguaggio accettato da una macchina di Turing (o da un FA o da un PDA) anche se è l'insieme vuoto oppure l'insieme di tutte le stringhe sull'alfabeto di input. Nel trattare macchine di Turing randomizzate bisogna usare una certa cautela in merito al concetto di accettazione; è anche possibile che una TM randomizzata non accetti alcun linguaggio. Quando esaminiamo le operazioni di una TM randomizzata M a fronte di un input w, dobbiamo considerare tutti i contenuti possibili del nastro casuale. È possibile che M accetti un input con determinate stringhe casuali e lo rifiuti con altre; anzi, se la TM randomizzata deve operare con maggior efficienza rispetto a una deterministica, è essenziale che contenuti diversi del nastro casuale inducano comportamenti diversi.[8]

[8]Osserviamo che la TM randomizzata descritta nell'Esempio 11.12 non riconosce un linguaggio, ma opera una trasformazione dell'input. Il tempo di esecuzione, a differenza del risultato, dipende dal contenuto del nastro casuale.

Se applichiamo a una TM randomizzata il criterio di accettazione per stato finale, come in quelle tradizionali, ogni input w ha una probabilità di essere accettato pari alla frazione dei possibili contenuti del nastro casuale che porta all'accettazione. Poiché il numero dei possibili contenuti del nastro è infinito, il calcolo della probabilità richiede una certa cura. Dal momento che ogni sequenza di mosse che porta all'accettazione visita solo una porzione finita del nastro casuale, la probabilità di una particolare configurazione è finita e uguale a 2^{-m}, se m è il numero di celle del nastro casuale che sono state visitate e che hanno influenzato almeno una mossa della TM. Un esempio illustrerà il calcolo in un caso semplice.

Esempio 11.13 La tabella della Figura 11.7 riproduce la funzione di transizione di una TM randomizzata M. M usa solo un nastro di input e il nastro casuale. Il suo comportamento è semplicissimo: non cambia mai un simbolo sui nastri e muove le testine soltanto verso destra (direzione R) oppure le mantiene ferme (direzione S). Anche se non abbiamo definito una notazione formale per le transizioni di una TM randomizzata, le voci della tabella dovrebbero essere chiare: ogni riga corrisponde a uno stato e ogni colonna a una coppia di simboli XY, dove X è il simbolo visto sul nastro di input e Y è il simbolo visto sul nastro casuale. L'elemento $qUVDE$ significa che la TM entra nello stato q, scrive U sul nastro di input e V sul nastro casuale, muove la testina di input in direzione D e quella del nastro casuale in direzione E.

	00	01	10	11	$B0$	$B1$
$\to q_0$	$q_1 00RS$	$q_3 01SR$	$q_2 10RS$	$q_3 11SR$		
q_1	$q_1 00RS$				$q_4 B0SS$	
q_2			$q_2 10RS$		$q_4 B0SS$	
q_3	$q_3 00RR$			$q_3 11RR$	$q_4 B0SS$	$q_4 B1SS$
$*q_4$						

Figura 11.7 La funzione di transizione di una macchina di Turing randomizzata.

Riassumiamo il comportamento di M su una stringa di input w di 0 e 1. Nello stato iniziale q_0, M guarda il primo bit casuale e svolge una verifica su w, a seconda che il bit casuale sia 0 o 1.

Se è 0, M verifica se w consiste in un solo simbolo, 0 o 1, eventualmente ripetuto. In questo caso M non legge nessun altro bit casuale, ma mantiene ferma la testina del secondo nastro. Se il primo bit di w è 0, M passa allo stato q_1, in cui si muove verso destra sopra ogni 0, ma muore se vede un 1. Se M raggiunge il primo blank sul nastro di input mentre si trova nello stato q_1, passa allo stato accettante q_4. Analogamente, se il primo bit di w è 1, e il primo bit casuale è 0, M passa allo stato q_2, dove controlla se tutti gli altri bit di w sono 1. Se è così, accetta.

Consideriamo ora che cosa fa M se il primo bit casuale è 1. M confronta w con il secondo bit causale e con quelli successivi, accettando solo se le due stringhe coincidono. Di conseguenza dallo stato q_0, se legge 1 sul secondo nastro, M passa allo stato q_3. Nel farlo M muove la testina del nastro casuale verso destra sopra un nuovo bit casuale, mentre la testina del nastro di input viene mantenuta ferma, per cui tutti i simboli di w saranno confrontati con i bit casuali. Nello stato q_3, M confronta i due nastri, spostando verso destra le due testine. Se a un certo punto i simboli non coincidono, M muore senza accettare; se invece raggiunge un blank sul nastro di input, accetta.

Calcoliamo ora la probabilità di accettazione di certi input. In primo luogo consideriamo un input omogeneo, cioè formato con un solo simbolo, come 0^i per $i \geq 1$. Il primo bit casuale sarà 0 con probabilità 1/2. In questo caso la verifica di omogeneità dà esito positivo e 0^i viene accettato. D'altra parte c'è una probabilità 1/2 che il primo bit casuale sia 1. In questo caso 0^i sarà accettato se e solo se i bit casuali dal secondo all'$(i+1)$-esimo sono tutti 0, il che accade con probabilità 2^{-i}. Pertanto la probabilità totale di accettazione di 0^i è

$$\frac{1}{2} + \frac{1}{2}2^{-i} = \frac{1}{2} + 2^{-(i+1)}$$

Consideriamo ora il caso di un input eterogeneo w, cioè un input formato sia da 0 sia da 1, come 00101. L'input non viene accettato se il primo bit casuale è 0. Se il primo bit casuale è 1, la sua probabilità di accettazione è 2^{-i}, dove i è la lunghezza dell'input. Perciò la probabilità totale di accettazione di un input eterogeneo di lunghezza i è $2^{-(i+1)}$. Per esempio la probabilità di accettazione di 00101 è 1/64. □

Concludiamo che è possibile calcolare la probabilità con cui una TM randomizzata accetta una stringa assegnata. Che la stringa sia nel linguaggio o no dipende da come viene definita l'"appartenenza" al linguaggio di una TM randomizzata. Nei prossimi paragrafi daremo due diverse definizioni di accettazione, ognuna delle quali conduce a una classe di linguaggi differente.

11.4.4 La classe $\mathcal{RP}$

L'essenza della prima classe di linguaggi, chiamata $\mathcal{RP}$ (da "random polynomial", ossia polinomiale casuale), è che per essere in $\mathcal{RP}$ un linguaggio L dev'essere accettato da una TM randomizzata M nel senso che ora definiamo.

1. Se w non è in L, allora la probabilità che M accetti w è 0.

2. Se w è in L, allora la probabilità che M accetti w è almeno 1/2.

3. Esiste un polinomio $T(n)$ tale che se l'input w è di lunghezza n, tutte le esecuzioni di M, indipendentemente dai contenuti del nastro casuale, si arrestano dopo al massimo $T(n)$ passi.

Non determinismo e caso

Tra una TM randomizzata e una TM non deterministica ci sono analogie superficiali. Possiamo immaginare le scelte non deterministiche di una NTM come governate da un nastro con bit casuali: ogni volta che la NTM ha una scelta, consulta il nastro casuale e sceglie dando la stessa probabilità a tutte le opzioni. Se però interpretiamo una NTM in questo modo, la regola di accettazione è piuttosto diversa da quella per $\mathcal{RP}$: un input viene rifiutato se la sua probabilità di accettazione è 0, e viene accettato se la probabilità è qualsiasi valore maggiore di 0, per quanto piccolo.

Nella definizione di $\mathcal{RP}$ ci sono due questioni indipendenti. I punti (1) e (2) definiscono una macchina di Turing randomizzata di tipo speciale, a volte denominata algoritmo *Montecarlo*. A prescindere dal tempo di esecuzione, si può dire che una TM randomizzata è "Montecarlo" se accetta con probabilità 0 oppure con probabilità superiore a 1/2, senza casi intermedi. Il punto (3) riguarda solo il tempo di esecuzione, che non dipende dal fatto che la TM sia "Montecarlo" o no.

Esempio 11.14 Consideriamo la TM randomizzata dell'Esempio 11.13. Essa soddisfa certamente la condizione (3) perché il suo tempo di esecuzione è $O(n)$, indipendentemente dai contenuti del nastro casuale. Però non accetta nessun linguaggio nel senso richiesto dalla definizione di $\mathcal{RP}$. Infatti, mentre gli input omogenei come 000 sono accettati con probabilità minima 1/2 e perciò soddisfano il punto (2), esistono altri input, come 001, accettati con una probabilità che non è né 0 né almeno 1/2; per esempio 001 è accettato con probabilità 1/16. □

Esempio 11.15 Descriviamo in termini informali una TM randomizzata che è sia polinomiale sia Montecarlo, e dunque accetta un linguaggio in $\mathcal{RP}$. L'input sarà interpretato come un grafo, e la domanda è se il grafo contenga un triangolo, ossia tre nodi le cui coppie sono connesse da lati. Gli input con un triangolo sono nel linguaggio, gli altri no.

L'algoritmo Montecarlo sceglie ripetutamente e a caso un lato (x, y) e un nodo z, diverso da x e y. Ogni scelta viene fatta leggendo nuovi bit sul nastro casuale. Per ogni x, y e z selezionati, la TM verifica se l'input contiene i lati (x, z) e (y, z), e se è così dichiara che il grafo di input ha un triangolo.

In totale vengono compiute k scelte di un lato e di un nodo; la TM accetta se una di esse risulta un triangolo; altrimenti si ferma senza accettare. Se il grafo non ha triangoli, nessuna delle k scelte risulterà un triangolo. La condizione (1) della definizione di $\mathcal{RP}$ è quindi soddisfatta: se l'input non è nel linguaggio, la probabilità di accettazione è 0.

Supponiamo che il grafo abbia n nodi ed e lati. Se c'è almeno un triangolo, la probabilità che i suoi tre nodi siano selezionati in un'esecuzione è $(\frac{3}{e})(\frac{1}{n-2})$. Infatti tre degli e lati sono nel triangolo, e se uno di essi viene scelto, la probabilità che anche il terzo nodo venga selezionato è $1/(n-2)$. Si tratta di una probabilità bassa, ma l'esperimento è ripetuto k volte. La probabilità che almeno uno dei k esperimenti produca il triangolo è:

$$1 - \left(1 - \frac{3}{e(n-2)}\right)^k \tag{11.4}$$

Secondo una nota approssimazione, per piccoli valori di x, $(1-x)^k$ è circa e^{-kx}, dove $e = 2.718\cdots$ è la base dei logaritmi naturali. Di conseguenza, se scegliamo k in modo che, per esempio, $kx = 1$, e^{-kx} sarà significativamente minore di $1/2$ e $1 - e^{-kx}$ sarà significativamente maggiore di 1/2, più precisamente circa 0.63. Perciò possiamo scegliere $k = e(n-2)/3$ per garantire che la probabilità di accettazione di un grafo con un triangolo secondo l'Equazione 11.4 sia almeno 1/2. L'algoritmo descritto è dunque Montecarlo.

Dobbiamo ora considerare il tempo di esecuzione della TM. Sia e sia n non superano la lunghezza dell'input, e si è scelto che k non sia maggiore del quadrato della lunghezza, perché è proporzionale al prodotto di e ed n. Ogni tentativo scandisce l'input al massimo quattro volte (per scegliere il lato e il nodo, e poi per controllare la presenza di due ulteriori lati), ed è quindi lineare rispetto alla lunghezza dell'input. Pertanto la TM si arresta dopo un tempo al massimo cubico nella lunghezza dell'input, cioè la TM ha un tempo di esecuzione polinomiale e soddisfa quindi la terza e ultima condizione per cui un linguaggio è in $\mathcal{RP}$.

Concludiamo che il linguaggio dei grafi con un triangolo è nella classe $\mathcal{RP}$. Dato che sarebbe possibile fare una ricerca sistematica di tutti i possibili triangoli, il linguaggio è anche in $\mathcal{P}$. Come abbiamo detto all'inizio del Paragrafo 11.4, è difficile trovare esempi in $\mathcal{RP} - \mathcal{P}$. □

11.4.5 Riconoscimento di linguaggi in $\mathcal{RP}$

Supponiamo ora di avere una macchina di Turing Montecarlo M che riconosce un linguaggio L in tempo polinomiale. Abbiamo una stringa w e vogliamo sapere se è in L. Se eseguiamo M su L usando una monetina o un altro dispositivo che generi numeri casuali per simulare la creazione di bit casuali, sappiamo che:

1. se w non è in L, l'esecuzione non condurrà alla sua accettazione

2. se w è in L, esiste almeno il 50% di probabilità che sia accettato.

Se però consideriamo il risultato di un'esecuzione come definitivo, a volte rifiuteremo w quando avremmo dovuto accettarlo (un *falso negativo*), anche se non accetteremo mai

Il ruolo di 1/2 nella definizione di $\mathcal{RP}$

Abbiamo definito $\mathcal{RP}$ imponendo che la probabilità di accettare una stringa w in L sia almeno 1/2, ma avremmo potuto usare una qualunque costante compresa tra 0 e 1, estremi esclusi. Il Teorema 11.16 afferma che, ripetendo l'esperimento fatto da M per un congruo numero di volte, possiamo elevare a piacimento la probabilità di accettazione, fino a 1 escluso. La tecnica per diminuire la probabilità di non accettazione di una stringa in L che abbiamo usato nel Paragrafo 11.4.5 ci permetterà, data una TM randomizzata con una probabilità maggiore di 0 di accettare w in L, di aumentare questa probabilità fino a 1/2, ripetendo l'esperimento per un numero fisso di volte.

Continueremo a richiedere 1/2 come probabilità di accettazione nella definizione di $\mathcal{RP}$, ma dobbiamo essere consapevoli che qualunque probabilità non nulla è adeguata. Peraltro un cambio della costante modifica il linguaggio definito da una particolare TM randomizzata. Nell'Esempio 11.14 abbiamo visto che, abbassando la probabilità richiesta a 1/16, la stringa 001 sarebbe nel linguaggio della TM randomizzata.

quando non avremmo dovuto (un *falso positivo*). Di conseguenza dobbiamo distinguere la TM randomizzata dall'algoritmo che usiamo per decidere se w è in L. Non è possibile evitare del tutto i falsi negativi, sebbene si possa ridurre la probabilità di incorrervi entro limiti prefissati, ripetendo più volte la verifica.

Per esempio, se vogliamo una probabilità su un miliardo di falso negativo, possiamo eseguire una verifica trenta volte. Se w è in L, la probabilità che tutte e trenta le verifiche non portino all'accettazione non supera 2^{-30}, che è meno di 10^{-9}, cioè una su un miliardo. In generale, se vogliamo che la probabilità di falsi negativi sia minore di $c > 0$, dobbiamo eseguire la verifica $\log_2(1/c)$ volte. Questo numero è una costante se lo è c. Poiché un'esecuzione della TM randomizzata M impiega tempo polinomiale (assumiamo che L sia in $\mathcal{RP}$), sappiamo che anche la ripetizione della verifica richiede tempo polinomiale. Le conclusioni di questo ragionamento si possono enunciare in forma di teorema.

Teorema 11.16 Se L è in $\mathcal{RP}$, allora per ogni costante $c > 0$ piccola a piacere, esiste un algoritmo polinomiale randomizzato che risponde alla domanda se un input w è in L, non produce falsi positivi, e produce falsi negativi con una probabilità non maggiore di c. □

11.4.6 La classe $\mathcal{ZPP}$

La seconda classe di linguaggi legata alla randomizzazione è detta *polinomiale probabilistica con errore nullo*, o $\mathcal{ZPP}$. Essa è basata su TM randomizzate che si arrestano sempre con un tempo medio di arresto che è un polinomio nella lunghezza dell'input. Queste TM accettano entrando in uno stato accettante (e arrestandosi in quel punto) e rifiutano un input se si arrestano senza aver accettato. La definizione della classe $\mathcal{ZPP}$ è quindi molto simile a quella di $\mathcal{P}$, salvo per il fatto che $\mathcal{ZPP}$ permette al caso di influire sul comportamento della TM, e che il tempo di esecuzione misurato è quello medio anziché quello relativo al caso peggiore.

Una TM che dà sempre la risposta corretta, ma il cui tempo di esecuzione varia a seconda dei valori dei bit casuali, è detta macchina di Turing *Las Vegas* o algoritmo Las Vegas. Possiamo dunque considerare $\mathcal{ZPP}$ come la classe dei linguaggi accettati da macchine di Turing Las Vegas con tempo medio di esecuzione polinomiale.

11.4.7 Relazioni tra $\mathcal{RP}$ e $\mathcal{ZPP}$

Una semplice relazione lega le due classi randomizzate appena definite. Prima di enunciare il relativo teorema dobbiamo però esaminare i complementi delle classi. Dovrebbe essere chiaro che se L è in $\mathcal{ZPP}$ lo è anche $\overline{L}$. Infatti, se L è accettato da una TM M Las Vegas con tempo medio polinomiale, allora $\overline{L}$ è accettato da una variante di M in cui trasformiamo l'accettazione in arresto senza accettazione, mentre se M si arresta senza accettare, passiamo a uno stato accettante e ci arrestiamo.

Non è invece ovvio che $\mathcal{RP}$ sia chiusa rispetto alla complementazione, in quanto la definizione delle macchine di Turing Montecarlo tratta in modo diverso accettazione e rifiuto. Di conseguenza definiamo la classe co-$\mathcal{RP}$ come l'insieme dei linguaggi L tali che $\overline{L}$ sia in $\mathcal{RP}$; cioè co-$\mathcal{RP}$ rappresenta i complementi dei linguaggi in $\mathcal{RP}$.

Teorema 11.17 $\mathcal{ZPP} = \mathcal{RP} \cap$ co-$\mathcal{RP}$.

DIMOSTRAZIONE In primo luogo dimostriamo $\mathcal{RP} \cap$ co-$\mathcal{RP} \subseteq \mathcal{ZPP}$. Sia L un linguaggio in $\mathcal{RP} \cap$ co-$\mathcal{RP}$. Allora c'è una TM Montecarlo per L e una per $\overline{L}$, ciascuna con tempo di esecuzione polinomiale. Sia $p(n)$ un polinomio sufficientemente grande da limitare il tempo di esecuzione di ambedue le macchine. Definiamo una TM Las Vegas M per L.

1. Eseguiamo la TM Montecarlo per L; se accetta, allora M accetta e si arresta.

2. In caso contrario eseguiamo la TM Montecarlo per $\overline{L}$. Se la TM accetta, allora M si arresta senza accettare. Altrimenti M ritorna al passo (1).

Ovviamente M accetta un input w solo se questo si trova in L e lo rifiuta solo se non si trova in L. Il tempo di esecuzione di un turno (esecuzione dei passi 1 e 2) è $2p(n)$. La

probabilità che un turno risolva la questione è almeno 1/2. Infatti, se w è in L, allora il passo (1) ha il 50% di probabilità di condurre all'accettazione da parte di M, e se w non è in L, il passo (2) ha il 50% di probabilità di condurre al rifiuto da parte di M. Quindi il tempo medio di esecuzione di M non è maggiore di

$$2p(n) + \frac{1}{2}2p(n) + \frac{1}{4}2p(n) + \frac{1}{8}2p(n) + \cdots = 4p(n)$$

Consideriamo ora l'inverso: assumiamo che L sia in $\mathcal{ZPP}$ e dimostriamo che L è sia in $\mathcal{RP}$ sia in co-$\mathcal{RP}$. Sappiamo che L è accettato da una TM Las Vegas M_1 con un tempo medio di esecuzione che è un polinomio $p(n)$. Costruiamo una TM Montecarlo TM M_2 per L. M_2 simula M_1 per $2p(n)$ passi. Se M_1 accetta durante questo intervallo di tempo, lo fa anche M_2; altrimenti M_2 rifiuta.

Supponiamo che l'input w di lunghezza n non sia in L. Allora M_1 non accetterà w, e perciò non lo farà neppure M_2. Supponiamo che w sia in L. Prima o poi M_1 accetterà sicuramente w, ma non è detto che lo faccia entro $2p(n)$ passi.

Noi affermiamo però che la probabilità che M_1 accetti w entro $2p(n)$ passi è almeno 1/2. Supponiamo che la probabilità di accettazione di w da parte di M_1 entro il tempo $2p(n)$ sia una costante $c < 1/2$. Allora il tempo medio di esecuzione di M_1 su input w è almeno $(1 - c)2p(n)$, dato che $1 - c$ è la probabilità che M_1 impieghi *più* di $2p(n)$. Se $c < 1/2$, allora $2(1 - c) > 1$, e il tempo medio di esecuzione di M_1 su w è maggiore di $p(n)$. Abbiamo così contraddetto l'ipotesi che M_1 abbia un tempo medio di esecuzione non superiore a $p(n)$, e perciò concludiamo che la probabilità che M_2 accetti è almeno 1/2. Di conseguenza M_2 è una TM Montecarlo con limite polinomiale sul tempo, il che dimostra che L è in $\mathcal{RP}$.

Per dimostrare che anche L è in co-$\mathcal{RP}$, usiamo essenzialmente la stessa costruzione, complementando però il risultato di M_2. In altre parole, per accettare $\overline{L}$ facciamo sì che M_2 accetti quando M_1 rifiuta entro un tempo $2p(n)$, mentre M_2 rifiuta nel caso opposto. In questo modo M_2 è una TM Montecarlo con limite polinomiale sul tempo per $\overline{L}$. □

11.4.8 Relazioni con le classi $\mathcal{P}$ ed $\mathcal{NP}$

Il Teorema 11.17 indica che $\mathcal{ZPP} \subseteq \mathcal{RP}$. Possiamo collocare queste classi tra $\mathcal{P}$ ed $\mathcal{NP}$ grazie a due semplici teoremi.

Teorema 11.18 $\mathcal{P} \subseteq \mathcal{ZPP}$.

DIMOSTRAZIONE Una TM deterministica con limite polinomiale sul tempo è anche una TM Las Vegas con limite polinomiale sul tempo, che non ricorre alla possibilità di compiere scelte casuali. □

Teorema 11.19 $\mathcal{RP} \subseteq \mathcal{NP}$.

DIMOSTRAZIONE Supponiamo di avere una TM Montecarlo M_1 con limite polinomiale sul tempo per un linguaggio L. Possiamo costruire una TM non deterministica M_2 per L con lo stesso limite di tempo. Ogni volta che M_1 esamina un bit casuale per la prima volta, M_2 sceglie, in modo non deterministico, entrambi i valori possibili di quello specifico bit, e li scrive su un nastro che simula il nastro casuale di M_1. M_2 accetta quando e solo quando M_1 accetta.

Sia w in L. Allora, dato che M_1 ha almeno il 50% di probabilità di accettare w, deve esistere una sequenza di bit sul nastro casuale che porta all'accettazione di w. M_2 sceglierà, fra le altre, quella sequenza di bit, e quindi accetterà per quella scelta. Quindi w è in $L(M_2)$. Se però w non è in L, nessuna sequenza di bit casuali porta M_1 all'accettazione, e perciò nessuna sequenza di scelte induce M_2 ad accettare. Di conseguenza w non è in $L(M_2)$. □

La Figura 11.8 illustra la relazione tra le classi che abbiamo introdotto e le altre classi "vicine".

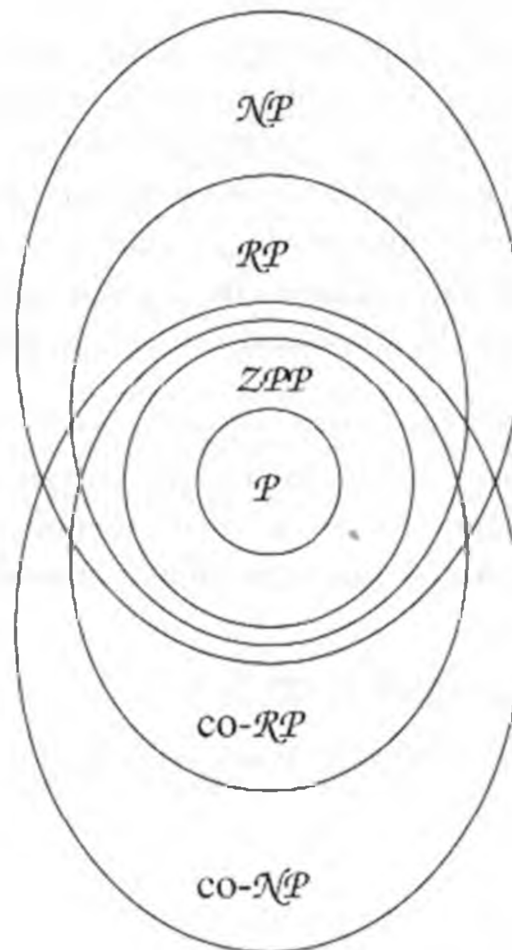


Figura 11.8 Relazioni di $\mathcal{ZPP}$ ed $\mathcal{RP}$ con altre classi.

11.5 Complessità e numeri primi

In questo paragrafo esamineremo un problema particolare: verificare se un intero è primo. Il problema è interessante anzitutto perché, come spiegheremo, i numeri primi e la verifica di primalità sono componenti essenziali nei sistemi di sicurezza per i computer. Mostreremo poi che i numeri primi sono sia in $\mathcal{NP}$ sia in co-$\mathcal{NP}$. Infine discuteremo un algoritmo randomizzato con cui si dimostra che sono anche in $\mathcal{RP}$.

11.5.1 L'importanza della verifica di primalità

Un numero intero p è *primo* se gli unici interi che lo dividono senza resto sono 1 e il numero stesso. Se un numero intero non è primo, diciamo che è *composto*. Ogni numero composto può essere scritto come prodotto di numeri primi in modo unico, salvo che per l'ordine dei fattori.

Esempio 11.20 I numeri primi più bassi sono 2, 3, 5, 7, 11, 13 e 17. L'intero 504 è composto e la sua scomposizione in fattori è $2^3 \times 3^2 \times 7$. □

I metodi più comuni per aumentare la sicurezza nei computer si fondano sull'ipotesi che sia difficile fattorizzare un numero, cioè trovare i fattori primi di un numero composto. In particolare i sistemi basati sui cosiddetti codici RSA (dalle iniziali degli inventori della tecnica, ossia R. Rivest, A. Shamir e L. Adleman) usano numeri interi, per esempio di 128 bit, che sono il prodotto di due primi, ciascuno di circa 64 bit. Presentiamo due situazioni in cui i numeri primi hanno una funzione importante.

Crittografia a chiave pubblica

Vogliamo comprare un libro in una libreria on-line. Il venditore chiede il numero della nostra carta di credito, ma è rischioso inserirlo in un modulo e trasmetterlo per telefono oppure via Internet: qualcuno potrebbe essere in ascolto sulla linea o intercettare pacchetti durante il loro percorso.

Per evitare che un malintenzionato legga il numero della carta di credito, il sito di vendita invia al browser una chiave k, per esempio il prodotto di due numeri primi con lunghezza totale di 128 bit, generato a questo scopo dal computer della libreria virtuale. Il nostro browser applica una funzione $y = f_k(x)$, che usa sia la chiave k sia i dati x da crittografare. La funzione f, che fa parte dello schema RSA, è pubblica, e quindi nota anche a potenziali spie. Si ritiene però che non sia possibile, senza conoscere la scomposizione in fattori di k, calcolare la funzione inversa f_k^{-1}, per la quale $x = f_k^{-1}(y)$, in un tempo inferiore a un esponenziale nella lunghezza di k.

Di conseguenza, anche se una spia vede y e conosce f, non può recuperare x, che in questo caso è il numero di carta di credito, senza prima scoprire k e scomporlo in fattori

primi. D'altra parte la libreria virtuale, che conosce la scomposizione in fattori della chiave k, avendola generata, può applicare facilmente f_k^{-1} e ottenere x da y.

Firme con chiave pubblica

Descriviamo il contesto per cui sono stati sviluppati i codici RSA. Si vuole "firmare" un messaggio elettronico in modo che i destinatari possano individuare con facilità il mittente e che non sia possibile "falsificarne" la firma. Immaginiamo di voler firmare un messaggio x = "Mi impegno a pagare \$10 a Sally Lee", ma di voler impedire che Sally, o una terza persona, possa comporre un messaggio simile firmato a nostra insaputa.

A tal fine scegliamo una chiave k, di cui solo noi conosciamo i fattori primi, e la pubblichiamo, per esempio sul nostro sito Web, in modo che chiunque possa applicare la funzione f_k a un messaggio. Se vogliamo firmare il messaggio x e spedirlo a Sally, calcoliamo $y = f_k^{-1}(x)$ e le inviamo y. Sally recupera dal sito f_k, la *chiave pubblica*, con cui calcola $x = f_k(y)$. Viene così a sapere con certezza che ci siamo impegnati a pagarle \$10.

Se neghiamo di aver inviato il messaggio y, Sally può affermare davanti a un giudice che solo noi conosciamo la funzione f_k^{-1}, e che sarebbe impossibile per lei, o per un'altra persona, scoprirla. Dunque solo noi possiamo aver creato y. Il sistema si fonda sull'ipotesi, probabile ma non dimostrata, che è troppo difficile scomporre in fattori primi il prodotto di due numeri primi grandi.

Requisiti concernenti la complessità della verifica di primalità

Si ritiene che le due situazioni descritte sopra siano adeguate e sicure, nel senso che ci vuole davvero un tempo esponenziale per scomporre in fattori il prodotto di due numeri primi grandi. La teoria della complessità studiata qui e nel Capitolo 10 ha un duplice impiego nello studio della sicurezza e della crittografia.

1. La costruzione di chiavi pubbliche richiede la capacità di trovare velocemente numeri primi grandi. Un risultato fondamentale della teoria dei numeri afferma che la probabilità che un numero di n bit sia un numero primo è nell'ordine di $1/n$. Perciò, se avessimo un metodo polinomiale in tempo (rispetto a n, non al valore del numero primo) per verificare se un numero di n bit è un numero primo, potremmo scegliere numeri a caso, esaminarli e fermarci quando ne abbiamo trovato uno primo. Questo metodo produce un algoritmo Las Vegas polinomiale in tempo per scoprire numeri primi, dato che il numero medio di numeri che dobbiamo esaminare prima di incontrare un primo di n bit è circa n. Per esempio, se vogliamo primi di 64 bit, dobbiamo verificare in media circa 64 interi, anche se in un caso sfortunato potremmo doverne verificare un numero indefinitamente maggiore. Purtroppo sembra che non esista un algoritmo che garantisce un tempo polinomiale per

i primi; come vedremo nel Paragrafo 11.5.4 esiste invece un algoritmo Montecarlo in tempo polinomiale.

2. La sicurezza della crittografia basata su RSA dipende dalla mancanza di un metodo generale e polinomiale in tempo (rispetto al numero di bit della chiave) per scomporre in fattori un numero, in particolare un numero di cui si sa che è il prodotto di due numeri primi grandi. Sarebbe bello poter dimostrare che l'insieme dei primi, o almeno l'insieme dei numeri composti, è NP-completo. In questo caso, infatti, un algoritmo polinomiale che scompone in fattori proverebbe che $\mathcal{P} = \mathcal{NP}$ perché produrrebbe un algoritmo in tempo polinomiale per quei due linguaggi. Purtroppo vedremo nel Paragrafo 11.5.5 che sia i numeri primi sia i numeri composti sono in $\mathcal{NP}$. In quanto complementi l'uno dell'altro, se uno dei due fosse NP-completo ne conseguirebbe che $\mathcal{NP}$ = co-$\mathcal{NP}$, cosa di cui dubitiamo. Inoltre il fatto che l'insieme dei numeri primi è in $\mathcal{RP}$ significa che, se potessimo dimostrare che i numeri primi sono NP-completi, potremmo concludere che $\mathcal{RP} = \mathcal{NP}$: un'altra situazione improbabile.

11.5.2 Introduzione all'aritmetica modulare

Prima di esaminare gli algoritmi per riconoscere l'insieme dei numeri primi, presentiamo le nozioni fondamentali dell'*aritmetica modulare*, ossia le comuni operazioni aritmetiche eseguite modulo un intero, spesso un numero primo. Sia p un intero. Gli *interi modulo* p sono $0, 1, \ldots, p-1$.

Possiamo definire somma e prodotto modulo p su questo insieme di p interi eseguendo il normale calcolo e considerando il resto della divisione per p del risultato. La somma è molto semplice perché il risultato o è minore di p, e non dobbiamo fare altro, o è compreso fra p e $2p-2$, nel qual caso basta sottrarre p per ottenere un intero nell'intervallo $0, 1, \ldots, p-1$. La somma modulare rispetta le normali proprietà algebriche: è commutativa, associativa, e ha 0 come identità. La sottrazione è ancora l'inverso della somma; possiamo calcolare la differenza modulare $x - y$ sottraendo come sempre e aggiungendo p se il risultato è minore di 0. L'opposto di x, cioè $-x$, coincide con $0 - x$ come nell'aritmetica ordinaria. Quindi $-0 = 0$, e se $x \neq 0$, $-x$ coincide con $p - x$.

Esempio 11.21 Poniamo $p = 13$. Allora $3 + 5 = 8$ e $7 + 10 = 4$. Infatti nell'aritmetica ordinaria $7 + 10 = 17$, che non è inferiore a 13. Perciò sottraiamo 13 per ottenere il risultato corretto, 4. Il valore di -5 modulo 13 è $13 - 5$, cioè 8. La differenza $11 - 4$ modulo 13 è 7, mentre la differenza $4 - 11$ è 6. Infatti nell'aritmetica ordinaria $4 - 11 = -7$, e aggiungendo 13 otteniamo 6. □

La moltiplicazione modulo p viene svolta moltiplicando come si fa con i numeri ordinari e prendendo poi il resto del risultato diviso per p. Anche la moltiplicazione soddisfa

le comuni proprietà algebriche: è commutativa e associativa, 1 è l'identità, 0 è l'elemento neutro, e la moltiplicazione è distributiva rispetto alla somma. La divisione per valori non nulli è più complessa, e l'esistenza di inversi di interi modulo p dipende dal fatto che p sia o no un numero primo. In generale, se x è un intero modulo p, ossia $0 \leq x < p$, allora x^{-1}, o $1/x$, è quel numero y tale che $xy = 1$ modulo p.

1	2	3	4	5	6
2	4	6	1	3	5
3	6	2	5	1	4
4	1	5	2	6	3
5	3	1	6	4	2
6	5	4	3	2	1

Figura 11.9 Moltiplicazione modulo 7.

Esempio 11.22 Nella Figura 11.9 vediamo la tabella di moltiplicazione per interi non nulli modulo il numero primo 7. L'elemento sulla riga i e sulla colonna j è il prodotto ij modulo 7. Osserviamo che ogni intero non nullo ha un inverso; 2 e 4 sono ciascuno l'inverso dell'altro, così come 3 e 5, mentre 1 e 6 sono i propri inversi. In altre parole 2×4, 3×5, 1×1 e 6×6 sono tutti 1. Di conseguenza possiamo dividere per qualsiasi numero non nullo x/y calcolando y^{-1} e moltiplicando poi $x \times y^{-1}$. Per esempio $3/4 = 3 \times 4^{-1} = 3 \times 2 = 6$.

1	2	3	4	5
2	4	0	2	4
3	0	3	0	3
4	2	0	4	2
5	4	3	2	1

Figura 11.10 Moltiplicazione modulo 6.

Confrontiamo questo caso con la tabella di moltiplicazione modulo 6. In primo luogo osserviamo che solo 1 e 5 *hanno* un inverso; ciascuno è il proprio inverso. Esistono inoltre numeri diversi da 0, ma il cui prodotto è 0, come 2 e 3. Si tratta di una situazione che non si verifica mai nella comune aritmetica degli interi e nell'aritmetica modulo un numero primo. □

Un'altra differenza fra la moltiplicazione modulo un numero primo e quella modulo un numero composto si rivela molto importante per le verifiche di primalità. Il *grado* di

un numero a modulo p è la più piccola potenza positiva di a uguale a 1. Enunciamo, senza dimostrarle, alcune proprietà importanti.

- Se p è un numero primo, allora $a^{p-1} = 1$ modulo p. Questo enunciato è detto *teorema di Fermat*.[9]
- Il grado di a modulo un numero primo p è sempre un divisore di $p - 1$.
- Se p è un numero primo, esiste sempre un a di grado $p - 1$ modulo p.

Esempio 11.23 Riconsideriamo la tabella della moltiplicazione modulo 7 nella Figura 11.9. Il grado di 2 è 3, dato che $2^2 = 4$ e $2^3 = 1$. Il grado di 3 è 6, dato che $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$ e $3^6 = 1$. Con calcoli analoghi si scopre che 4 ha grado 3, 5 ha grado 6, 6 ha grado 2, e 1 ha grado 1. □

11.5.3 La complessità del calcolo in aritmetica modulare

Prima di trattare le applicazioni dell'aritmetica modulare alla verifica di primalità, dobbiamo stabilire alcune proprietà relative al tempo di calcolo delle operazioni essenziali. Supponiamo di voler effettuare calcoli nell'algebra modulo un numero primo p, dove la rappresentazione binaria di p è lunga n bit, ovvero p è dell'ordine di grandezza di 2^n. Come sempre il tempo di esecuzione di un calcolo è dato in termini di n, la lunghezza dell'input, anziché in termini di p, il "valore" dell'input. Per esempio, poiché contare fino a p richiede un tempo $O(2^n)$, un calcolo che comporti p passi *non* sarà in tempo polinomiale come funzione di n.

Possiamo certamente sommare due numeri modulo p in un tempo $O(n)$ con un computer o con una TM multinastro. Dobbiamo semplicemente sommare i numeri binari, e se il risultato è maggiore o uguale a p, sottrarre p. Analogamente possiamo moltiplicare due numeri in un tempo $O(n^2)$ con un computer o con una macchina di Turing. Dopo aver moltiplicato i numeri nel modo consueto e aver ottenuto un risultato di non più di $2n$ bit, dividiamo per p e teniamo il resto.

Calcolare una potenza del numero x è più complicato perché l'esponente potrebbe a sua volta essere esponenziale in n. Come vedremo, un passo importante è l'elevamento di x alla potenza $p - 1$. Dato che $p - 1$ è dell'ordine di 2^n, se moltiplicassimo x per se stesso $p - 2$ volte compiremmo $O(2^n)$ moltiplicazioni. Anche se ogni moltiplicazione coinvolge solo numeri di n bit e può essere effettuata in un tempo $O(n^2)$, il tempo totale sarà $O(n^2 2^n)$, che non è polinomiale in n.

Possiamo però sfruttare la ricorsione per calcolare x^{p-1} (oppure ogni altra potenza di x fino a p) in un tempo polinomiale rispetto a n.

[9] Da non confondere con l'"ultimo teorema di Fermat", che asserisce la non esistenza di soluzioni intere a $x^n + y^n = z^n$ per $n \geq 3$.

1. Calcoliamo le potenze $x, x^2, x^4, x^8, \ldots$ con esponente minore o uguale a $p-1$, che sono al massimo n. Ogni valore è un numero di n bit, calcolato in un tempo $O(n^2)$ elevando al quadrato il valore precedente della sequenza. Il costo totale è quindi $O(n^3)$.

2. Troviamo la rappresentazione binaria di $p-1$, cioè $p-1 = a_{n-1} \cdots a_1 a_0$. Possiamo scrivere

$$p - 1 = a_0 + 2a_1 + 4a_2 + \cdots + 2^{n-1}a_{n-1}$$

dove ogni a_j è 0 oppure 1. Quindi

$$x^{p-1} = x^{a_0+2a_1+4a_2+\cdots+2^{n-1}a_{n-1}}$$

che è il prodotto dei valori x^{2^j} per i quali $a_j = 1$. Poichè nel passo (1) abbiamo calcolato i (non più di n) valori x^{2^j}, ognuno dei quali è un numero di n bit, possiamo calcolare il loro prodotto in un tempo $O(n^3)$.

Quindi l'intero calcolo di x^{p-1} richiede un tempo $O(n^3)$.

11.5.4 Verifica di primalità in tempo polinomiale randomizzato

Consideriamo ora come impiegare un calcolo randomizzato per trovare numeri primi grandi. Più precisamente dimostreremo che il linguaggio dei numeri composti è in $\mathcal{RP}$. Il metodo usato in pratica per generare numeri primi di n bit consiste nel prendere a caso un numero di n bit e applicare un numero elevato di volte, diciamo 50, il metodo Montecarlo per riconoscere i numeri composti. Se almeno un esperimento indica che il numero è composto, sappiamo che non è un numero primo. Se nessun esperimento indica che è composto, la probabilità che lo sia non supera 2^{-50}. Quindi possiamo affermare con una certa fiducia che il numero è primo, e fondare le operazioni su questo.

Non daremo l'algoritmo completo, e discuteremo invece un'idea che funziona, tranne che in un numero molto ristretto di casi. Ricordiamo il teorema di Fermat: se p è primo allora x^{p-1} modulo p è sempre 1. È inoltre un fatto che se p è un numero composto, ed esiste un qualsiasi x per il quale x^{p-1} modulo p non è 1, allora per almeno la metà dei valori di x nell'intervallo da 1 a $p-1$ vale $x^{p-1} \neq 1$.

Quindi possiamo utilizzare la seguente procedura come metodo Montecarlo per i numeri composti.

1. Prendiamo un numero x a caso nell'intervallo da 1 a $p-1$.

2. Calcoliamo x^{p-1} modulo p. Osserviamo che, se p è un numero di n bit, il calcolo richiede un tempo $O(n^3)$, come abbiamo visto alla fine del Paragrafo 11.5.3.

Fattorizzare in tempo polinomiale randomizzato

L'algoritmo del Paragrafo 11.5.4 rivela che un numero è composto, ma non dice come fattorizzarlo. Si ritiene che non ci sia modo di fattorizzare un numero in tempo polinomiale anche ricorrendo al caso, e nemmeno in media. Se questa assunzione fosse errata, le applicazioni presentate nel Paragrafo 11.5.1 non sarebbero sicure e non dovremmo usarle.

3. Se $x^{p-1} \neq 1$ modulo p, accettiamo: p è composto. Altrimenti ci arrestiamo senza accettare.

Se p è primo, allora $x^{p-1} = 1$. Quindi ci arresteremo sempre senza accettare; questo è uno dei requisiti di un metodo Montecarlo: se l'input non è nel linguaggio, non va mai accettato. Per quasi tutti i numeri composti, almeno la metà dei valori di x soddisfa $x^{p-1} \neq 1$. Pertanto avremo almeno il 50% di probabilità di accettare a ogni esecuzione dell'algoritmo, e questo è un altro requisito di un algoritmo per poter essere di tipo Montecarlo.
Quella appena descritta potrebbe essere la dimostrazione che i numeri composti sono in $\mathcal{RP}$, se non fosse per l'esistenza di un piccolo insieme di numeri composti c per i quali $x^{c-1} = 1$ modulo c per la maggior parte degli x nell'intervallo da 1 a $c - 1$, in particolare per gli x che non condividono un fattore primo con c. Questi numeri, detti *numeri di Carmichael*, richiedono un'altra verifica più complessa (che non descriviamo) per scoprire che sono numeri composti. Il più piccolo numero di Carmichael è 561. Ciò significa che $x^{560} = 1$ modulo 561 per tutti gli x che non sono divisibili per 3, 11 o 17, anche se $561 = 3 \times 11 \times 17$ è evidentemente un numero composto. Possiamo quindi, senza darne la dimostrazione completa, enunciare un teorema.

Teorema 11.24 L'insieme dei numeri composti è in $\mathcal{RP}$. □

11.5.5 Verifiche di primalità non deterministiche

Esaminiamo un altro risultato interessante e significativo legato alle verifiche di primalità: il linguaggio dei numeri primi è in $\mathcal{NP} \cap$ co-$\mathcal{NP}$. Pertanto il linguaggio dei numeri composti, complemento dei primi, è in $\mathcal{NP} \cap$ co-$\mathcal{NP}$. Se ne deduce che è improbabile che i numeri primi o i numeri composti siano NP-completi, perché in questo caso avremmo l'inattesa uguaglianza $\mathcal{NP}$ = co-$\mathcal{NP}$.

Una parte è semplice: i numeri composti sono ovviamente in $\mathcal{NP}$, quindi i primi sono in co-$\mathcal{NP}$. Proveremo innanzitutto questa proprietà.

Teorema 11.25 L'insieme dei numeri composti è in $\mathcal{NP}$.

DIMOSTRAZIONE Descriviamo un algoritmo non deterministico in tempo polinomiale per i numeri composti.

1. Dato un numero di n bit scegliamo un presunto fattore f di non più di n bit, evitando di scegliere $f = 1$ o $f = p$. Questa parte è non deterministica. Ogni possibile valore di f compare in una sequenza di scelte. Il tempo richiesto per una qualunque sequenza di scelte è $O(n)$.

2. Dividiamo p per f e verifichiamo se il resto è 0. Se sì, accettiamo il risultato. Questa parte è deterministica e può essere completata in tempo $O(n^2)$ su una TM multinastro.

Se p è composto deve avere almeno un fattore f diverso da 1 e p. Dato che la NTM sceglie tutti i numeri di lunghezza fino a n bit, in qualche diramazione sceglierà f. Questa diramazione porta all'accettazione. Nell'altro senso, l'accettazione da parte della NTM implica che è stato trovato un fattore di p diverso da 1 o da p stesso. Quindi la NTM descritta accetta il linguaggio formato da tutti e soli i numeri composti. □

Riconoscere i numeri primi con una NTM è più difficile. Se possiamo indovinare un motivo (un fattore) per cui un numero non è primo, e verificare che la scelta è corretta, come possiamo "indovinare" una ragione per cui un numero *è* primo? L'algoritmo non deterministico in tempo polinomiale si fonda sul fatto (affermato ma non provato) che se p è un numero primo, allora esiste un numero x compreso tra 1 e $p-1$ che ha grado $p-1$. Abbiamo visto nell'Esempio 11.23 che per il numero primo $p = 7$ i numeri 3 e 5 hanno ambedue grado 6.

Possiamo scegliere facilmente un numero x sfruttando il non determinismo di una NTM, ma non è chiaro come verificare che x ha grado $p-1$. Infatti, se applichiamo direttamente la definizione di "grado", dobbiamo verificare che $x^2, x^3, \ldots, x^{p-2}$ non valgono 1. Questo richiede l'esecuzione di $p-3$ moltiplicazioni, e quindi un tempo almeno 2^n, se p è un numero di n bit.

Una strategia migliore impiega un'altra proprietà menzionata ma non dimostrata: il grado di x modulo un primo p è un divisore di $p-1$. Quindi se conosciamo i fattori primi di $p-1$[10], è sufficiente verificare che $x^{(p-1)/q} \neq 1$ per ogni q fattore primo di $p-1$. Il numero di queste verifiche è $O(n)$, così che possiamo eseguirle tutte con un algoritmo in tempo polinomiale. Naturalmente non è facile fattorizzare $p-1$ in numeri primi, ma possiamo *scegliere* in modo non deterministico i fattori primi di $p-1$ e:

a) verificare che il loro prodotto sia $p-1$

[10] Osserviamo che, se p è primo, allora $p-1$ non è *mai* primo, tranne che nel caso poco interessante $p = 3$. Il motivo è che tutti i primi, eccetto 2, sono dispari.

b) verificare che ogni fattore sia primo usando ricorsivamente l'algoritmo non deterministico in tempo polinomiale che abbiamo definito.

I dettagli dell'algoritmo e la dimostrazione che è non deterministico e in tempo polinomiale si trovano nella dimostrazione del prossimo teorema.

Teorema 11.26 L'insieme dei numeri primi è in $\mathcal{NP}$.

DIMOSTRAZIONE Sia p un numero di n bit. Se n non è più grande di 2 (cioè p è 1, 2 o 3) rispondiamo direttamente alla domanda: 2 e 3 sono primi, mentre 1 non lo è. Altrimenti procediamo.

1. Scegliamo una lista di presunti fattori $(q_1, q_2, \ldots, q_k)$, le cui rappresentazioni binarie non superino in totale $2n$ bit, e nessuno dei quali abbia più di $n - 1$ bit. Lo stesso numero primo può apparire più di una volta, dato che $p - 1$ può avere un fattore primo elevato a una potenza maggiore di 1; per esempio se $p = 13$, i fattori primi di $p - 1 = 12$ formano la lista $(2, 2, 3)$. Questa parte è non deterministica e ogni diramazione richiede un tempo $O(n)$.

2. Moltiplichiamo i q tra loro e verifichiamo se il prodotto è $p - 1$. Questa parte non richiede un tempo maggiore di $O(n^2)$ ed è deterministica.

3. Se il prodotto è $p - 1$, verifichiamo ricorsivamente se ogni numero è primo usando l'algoritmo descritto.

4. Se i q sono tutti primi, scegliamo un valore di x e controlliamo se $x^{(p-1)/q_j} \neq 1$ per ogni q_j. Questa verifica garantisce che x ha grado $p - 1$ modulo p perchè, se così non fosse, il suo grado dovrebbe dividere almeno un $(p - 1)/q_j$, e abbiamo appena visto che non è vero. Osserviamo che ogni x elevato a una qualsiasi potenza del suo grado dev'essere 1. Per gli elevamenti a potenza si può applicare un metodo efficiente come quello descritto nel Paragrafo 11.5.3. Ci sono quindi al massimo k elevamenti a potenza, cioè senz'altro meno di n, ognuno dei quali può essere svolto in un tempo $O(n^3)$, con un tempo totale $O(n^4)$ per questo passo.

Infine dobbiamo verificare che questo algoritmo non deterministico impieghi un tempo polinomiale. Ogni passo, tranne quello ricorsivo (3), richiede un tempo non superiore a $O(n^4)$ in ogni diramazione non deterministica. Benchè la ricorsione sia complicata, possiamo rappresentare le chiamate ricorsive con l'albero della Figura 11.11. Alla radice sta il numero primo p di n bit che vogliamo esaminare. I suoi figli sono i q_j, gli ipotetici fattori di $p - 1$ da noi scelti, di cui dobbiamo stabilire se sono primi. Sotto ogni q_j si trovano gli ipotetici fattori di $q_j - 1$ che dobbiamo verificare, e così via, fino a numeri di non più di 2 bit, che sono le foglie dell'albero.

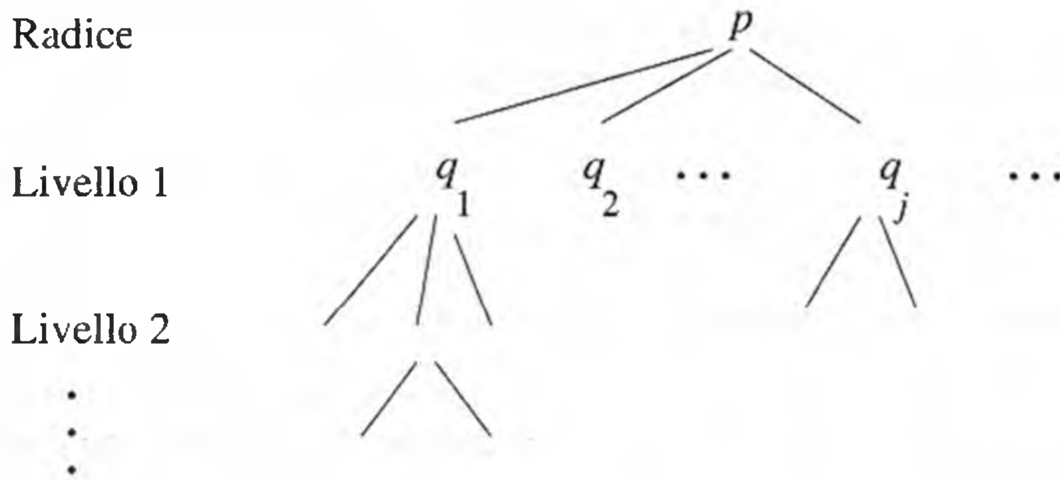


Figura 11.11 Le chiamate ricorsive dell'algorimo del Teorema 11.26 formano un albero di altezza e larghezza massima n.

Poichè il prodotto dei figli di ogni nodo è minore del valore del nodo stesso, il prodotto dei valori dei nodi a ogni livello di profondità dalla radice è al massimo p. Il lavoro richiesto per un nodo di valore i, escluso il lavoro svolto nelle chiamate ricorsive, è al massimo $a(\log_2 i)^4$, dove a è una costante; abbiamo infatti stabilito che questo lavoro è dell'ordine di grandezza della quarta potenza del numero di bit necessari per la rappresentazione binaria di i.

Per dare un limite superiore al lavoro necessario a ogni livello, dobbiamo massimizzare la somma $\sum_j a\big(\log_2(i_j)\big)^4$, con il vincolo che il prodotto $i_1 i_2 \cdots$ non superi p. Poichè la quarta potenza è una funzione convessa, il massimo si ottiene quando il valore è concentrato in uno degli i_j. Se $i_1 = p$ e non ci sono altri i_j, la somma è $a(\log_2 p)^4$. Questo è al massimo an^4, dato che n è il numero di bit della rappresentazione binaria di p. Quindi $\log_2 p$ non supera n.

Concludiamo che il lavoro richiesto a ogni livello è al massimo $O(n^4)$. Dato che ci sono al massimo n livelli, un lavoro $O(n^5)$ è sufficiente in ogni diramazione della procedura non deterministica che verifica se p sia primo o no. $\square$

Ora sappiamo che sia i primi sia il loro complemento sono in $\mathcal{NP}$. Se l'uno o l'altro fosse NP-completo, allora per il Teorema 11.2 avremmo una dimostrazione che $\mathcal{NP} =$ co-$\mathcal{NP}$.

11.5.6 Esercizi

Esercizio 11.5.1 Calcolate modulo 13:

a) $11 + 9$

* b) $9 - 11$

c) 5×8

* d) $5/8$

e) 5^8.

Esercizio 11.5.2 Nel Paragrafo 11.5.4 abbiamo affermato che, per la maggior parte dei valori di x compresi tra 1 e 560, $x^{560} = 1$ modulo 561. Scegliete alcuni valori di x e verificate l'equazione. Ricordate di esprimere prima 560 in binario e poi calcolate x^{2^j} modulo 561 per vari valori di j, per evitare di fare 559 moltiplicazioni, come spiegato nel Paragrafo 11.5.3.

Esercizio 11.5.3 Un intero x compreso tra 1 e $p-1$ si dice un *residuo quadratico* modulo p se esiste un intero y compreso fra 1 e $p-1$ tale che $y^2 = x$.

* a) Quali sono i residui quadratici modulo 7? Potete usare la tabella della Figura 11.9.

b) Quali sono i residui quadratici modulo 13?

! c) Dimostrate che se p è primo, il numero dei residui quadratici modulo p è $(p-1)/2$, cioè esattamente metà degli interi non nulli modulo p sono residui quadratici. *Suggerimento*: esaminate i risultati dei punti (a) e (b). Ne emerge uno schema che spieghi per quale ragione ogni residuo quadratico è il quadrato di due numeri diversi? Un intero può essere il quadrato di tre numeri diversi se p è primo?

11.6 Riepilogo

✦ *La classe co-$\mathcal{NP}$*: un linguaggio è in co-$\mathcal{NP}$ se il suo complemento è in $\mathcal{NP}$. Tutti i linguaggi in $\mathcal{P}$ sono senz'altro in co-$\mathcal{NP}$, ma è probabile che esistano linguaggi in $\mathcal{NP}$ che non sono in co-$\mathcal{NP}$, e viceversa. In particolare non sembra che i problemi NP-completi siano in co-$\mathcal{NP}$.

✦ *La classe $\mathcal{PS}$*: un linguaggio è in $\mathcal{PS}$ (spazio polinomiale) se è accettato da una TM deterministica per la quale esiste un polinomio $p(n)$ tale che, dato un input di lunghezza n, la TM non usa più di $p(n)$ celle del nastro.

✦ *La classe $\mathcal{NPS}$*: possiamo definire l'accettazione da parte di una TM non deterministica il cui uso del nastro è limitato da una funzione polinomiale della lunghezza dell'input. La classe di linguaggi corrispondente è detta $\mathcal{NPS}$. Per il teorema di Savitch sappiamo che $\mathcal{PS} = \mathcal{NPS}$. In particolare una NTM con limite di spazio $p(n)$ può essere simulata da una DTM in spazio $p^2(n)$.

- *Algoritmi randomizzati e macchine di Turing*: molti algoritmi sfruttano la casualità. In un computer reale si usa un generatore di numeri casuali che simula il "lancio della moneta". Una macchina di Turing può produrre un comportamento soggetto al caso se le viene dato un nastro aggiuntivo su cui viene scritta una sequenza casuale di bit.

- *La classe $\mathcal{RP}$*: un linguaggio è accettato in tempo polinomiale randomizzato se esiste una macchina di Turing randomizzata a tempo polinomiale che ha almeno il 50% di probabilità di accettare un input se è nel linguaggio. Se l'input non è nel linguaggio, allora la TM non accetta mai. La TM, o un algoritmo, che si comporta in questo modo è detta "Montecarlo".

- *La classe $\mathcal{ZPP}$*: un linguaggio è nella classe polinomiale probabilistica a errore nullo se è accettato da una macchina di Turing randomizzata che prende sempre la decisione corretta sull'appartenenza al linguaggio. La TM deve avere un tempo medio di esecuzione polinomiale; il caso peggiore può essere maggiore di qualsiasi polinomio. Una TM o un algoritmo che si comportano in questo modo sono detti "Las Vegas".

- *Relazioni tra classi di linguaggi*: la classe co-$\mathcal{RP}$ è l'insieme dei complementi dei linguaggi in $\mathcal{RP}$. Sono note le seguenti relazioni di inclusione: $\mathcal{P} \subseteq \mathcal{ZPP} \subseteq (\mathcal{RP} \cap \text{co-}\mathcal{RP})$. Inoltre $\mathcal{RP} \subseteq \mathcal{NP}$, e perciò co-$\mathcal{RP} \subseteq$ co-$\mathcal{NP}$.

- *I numeri primi ed $\mathcal{NP}$*: sia i numeri primi sia il complemento del linguaggio dei primi (i numeri composti) sono in $\mathcal{NP}$. Dunque è improbabile che i numeri primi o quelli composti siano NP-completi. Poiché esistono importanti schemi crittografici che si basano sui numeri primi, una tale dimostrazione sarebbe una prova evidente della loro sicurezza.

- *I numeri primi ed $\mathcal{RP}$*: i numeri composti sono in $\mathcal{RP}$. L'algoritmo polinomiale randomizzato per verificare se un numero è composto è usato per la generazione di numeri primi grandi, o almeno di numeri grandi con una probabilità arbitrariamente piccola di essere composti.

11.7 Bibliografia

La relazione [2] ha dato inizio allo studio delle classi di linguaggi definiti da limiti sullo spazio in una macchina di Turing. I primi problemi PS-completi sono descritti da Karp [4] in un lavoro che esamina l'importanza della NP-completezza. Ne abbiamo tratto la PS-completezza del problema enunciato nell'Esercizio 11.3.2, ossia se un'espressione regolare sia equivalente a Σ^*.

La PS-completezza delle formule booleane con quantificatori è contenuta in un lavoro mai pubblicato di L.J. Stockmeyer. La PS-completezza del gioco di Shannon (Esercizio 11.3.3) è tratta da [1].

La dimostrazione che i numeri primi sono in $\mathcal{NP}$ si deve a Pratt [9]. La presenza dei numeri composti in $\mathcal{RP}$ fu provata per la prima volta da Rabin [10]. È interessante notare che nello stesso periodo venne pubblicata una dimostrazione che i numeri primi sono in $\mathcal{P}$, a condizione che un'ipotesi, non dimostrata ma generalmente ritenuta valida, detta ipotesi estesa di Riemann, sia vera [6].

Numerosi libri permettono di approfondire le conoscenze sui temi presentati in questo capitolo. [7] tratta gli algoritmi randomizzati, compresi quelli per la verifica di primalità; [5] è un riferimento per gli algoritmi di aritmetica modulare; [3] e [8] trattano diverse altre classi di complessità non citate qui.

1. S. Even, R. E. Tarjan, "A combinatorial problem which is complete for polynomial space", *J. ACM* **23**:4 (1976), pp. 710–719.

2. J. Hartmanis, P. M. Lewis II, R. E. Stearns, "Hierarchies of memory limited computations," *Proc. Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design* (1965), pp. 179–190.

3. J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading MA, 1979.

4. R. M. Karp, "Reducibility among combinatorial problems", in *Complexity of Computer Computations* (R. E. Miller, ed.), Plenum Press, New York, 1972, pp. 85–104.

5. D. E. Knuth, *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*, Addison-Wesley, Reading MA, 1997 (terza edizione).

6. G. L. Miller, "Riemann's hypothesis and tests for primality", *J. Computer and System Sciences* **13** (1976), pp. 300–317.

7. R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge Univ. Press, 1995.

8. C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading MA, 1994.

9. V. R. Pratt, "Every prime has a succinct certificate", *SIAM J. Computing* **4**:3 (1975), pp. 214–220.

10. M. O. Rabin, "Probabilistic algorithms," in *Algorithms and Complexity: Recent Results and New Directions* (J. F. Traub, ed.), pp. 21–39, Academic Press, New York, 1976.