GIORGIO AUSIELLO,
FABRIZIO D'AMORE, GIORGIO GAMBOSI

LINGUAGGI MODELLI COMPLESSITÀ

SCIENZE E TECNOLOGIE INFORMATICHE

FrancoAngeli

Linguaggi, Modelli, Complessità

Giorgio Ausiello
Università di Roma "La Sapienza"

Fabrizio d'Amore
Università di Roma "La Sapienza"

Giorgio Gambosi
Università di Roma "Tor Vergata"

Questo documento è stato
scritto con l'ausilio del sistema
LaTeX2ε.
Stampato il 10 aprile 2002.

Da queste premesse incontrovertibili dedusse che la Biblioteca è totale, e che i suoi scaffali registrano tutte le possibili combinazioni dei venticinque simboli ortografici (numero, anche se vastissimo, non infinito) cioè tutto ciò ch'è dato di esprimere, in tutte le lingue. Tutto: la storia minuziosa dell'avvenire, le autobiografie degli arcangeli, il catalogo fedele della Biblioteca, migliaia e migliaia di cataloghi falsi, la dimostrazione della falsità di questi cataloghi, la dimostrazione della falsità del catalogo autentico, l'evangelo gnostico di Basilide, il commento di questo evangelo, il commento del commento di questo evangelo, il resoconto veridico della tua morte, la traduzione di ogni libro in tutte le lingue, le interpolazioni di ogni libro in tutti i libri.

J.L. Borges *La Biblioteca di Babele*

Indice

Introduzione

Capitolo 1

Concetti matematici di base

In questo primo capitolo vengono richiamate, in modo sintetico, alcune nozioni matematiche preliminari riguardanti concetti che sono alla base degli argomenti trattati nel volume: relazioni, funzioni, strutture algebriche, ecc. Inoltre vengono introdotte definizioni, proprietà elementari ed operazioni su linguaggi, strutture matematiche che hanno un ruolo fondamentale in informatica, dotate di proprietà insiemistiche ed algebriche particolarmente interessanti. Infine vengono illustrate alcune delle tecniche che saranno più frequentemente utilizzate nella dimostrazione di teoremi nel corso dei successivi capitoli.

Come si è detto nella premessa, in questo capitolo assumeremo noti il concetto primitivo di insieme e la definizione delle principali operazioni su insiemi (unione, intersezione, complementazione). Tra gli insiemi cui ci riferiremo, alcuni sono particolarmente importanti; essi sono riportati in Tabella 1.1 insieme ai simboli utilizzati per rappresentarli. Assumeremo anche noti i significati dei connettivi logici (and, or, not, implica, equivale a) e dei quantificatori (esiste, per ogni), che utilizzeremo anche al di fuori del contesto di specifiche teorie logiche, semplicemente come notazione per esprimere in modo formale e sintetico i concetti ad essi corrispondenti.

1.1 Insiemi, relazioni e funzioni. Cardinalità e contabilità degli insiemi.

1.1.1 Insieme delle parti

Sia dato un insieme A. Ricordiamo che con $x \in A$ indichiamo che l'elemento x appartiene ad A, con $B \subseteq A$ indichiamo che l'insieme B è un sottoinsieme di A e con $B \subset A$ indichiamo che B è un sottoinsieme proprio di A. Se A è un insieme costituito da un numero finito n di elementi, con $|A|$ indichiamo la sua

simbolo	descrizione
$\mathbb{N}$	naturali
$\mathbb{N}^+$	naturali positivi
$\mathbb{Z}$	interi
$\mathbb{Z}^+$	interi positivi (coincide con $\mathbb{N}^+$)
$\mathbb{Z}^-$	interi negativi
$\mathbb{Q}$	razionali
$\mathbb{Q}^+$	razionali positivi
$\mathbb{Q}^-$	razionali negativi
$\mathbb{R}$	reali
$\mathbb{R}^+$	reali positivi
$\mathbb{R}^-$	reali negativi

Tabella 1.1 Simboli adottati per indicare alcuni insiemi di particolare interesse.

cardinalità, cioè il numero di elementi che lo compongono e quindi abbiamo $|A|= n$. Infine con $\emptyset$ indichiamo l'insieme vuoto, tale cioè che $|\emptyset|= 0$.

Definizione 1.1 *L'insieme composto da tutti i sottoinsiemi di un insieme A si dice* insieme delle parti di *A, e si indica con $\mathcal{P}(A)$*

Si noti che $\mathcal{P}(A)$ contiene A e l'insieme vuoto $\emptyset$.

È facile dimostrare che se $|A|$ è finita e pari ad n allora $|\mathcal{P}(A)|$ è pari a 2^n: per questa ragione l'insieme $\mathcal{P}(A)$ viene anche indicato con 2^A.

Esercizio 1.1 Dimostrare che se A è un insieme finito $|2^A|= 2^{|A|}$.

Definizione 1.2 *Due insiemi A e B si dicono* uguali*, e si scrive $A = B$, se ogni elemento di A è anche elemento di B e, viceversa, se ogni elemento di B è anche elemento di A, cioè*

$$A = B \Longleftrightarrow (A \subseteq B \wedge B \subseteq A).$$

1.1.2 Principio di Induzione Matematica

Nel caso in cui si debbano dimostrare proprietà relative ad insiemi finiti, in teoria si può sempre pensare di verificare esaustivamente le proprietà su tutti gli elementi, per quanto tale procedimento potrebbe risultare non praticabile per insiemi aventi numerosi elementi. Nel caso in cui si debbano dimostrare

proprietà relative ad insiemi infiniti, tale procedimento è chiaramente non accettabile, nemmeno in teoria: per questo motivo è necessario utilizzare metodi adeguati a questo genere di insiemi.

Un metodo particolarmente utile per dimostrare proprietà dei numeri naturali è il *principio di induzione matematica*. Questo principio afferma quanto segue.

> Data una proposizione $P(n)$ definita per un generico numero naturale n, si ha che essa è vera per tutti i naturali se
>
> - $P(0)$ è vera (passo base dell'induzione);
> - per ogni naturale k, $P(k)$ vera (ipotesi induttiva) implica $P(k+1)$ vera (passo induttivo).

Esempio 1.1 Un classico esempio di dimostrazione per induzione matematica è quello della formula che esprime il valore della somma dei naturali non superiori ad n

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

Questa relazione può essere dimostrata per induzione osservando che per quanto riguarda il passo base si ha che

$$\sum_{i=0}^{0} i = \frac{0(0+1)}{2} = 0,$$

mentre per quanto riguarda il passo induttivo si ha

$$\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + (k+1) = \frac{k(k+1)}{2} + (k+1) = \frac{k^2+3k+2}{2} = \frac{(k+1)(k+2)}{2}.$$

Il che prova l'asserto.

Una versione più generale del principio di induzione matematica afferma quanto segue.

> Data una proposizione $P(n)$ definita per $n \geq n_0$ (con n_0 intero non negativo) si ha che essa è vera per tutti gli $n \geq n_0$ se:
>
> - $P(n_0)$ è vera (passo base dell'induzione);
> - per ogni naturale $k \geq n_0$, $P(k)$ vera (ipotesi induttiva) implica $P(k+1)$ vera (passo induttivo).

Ciò è giustificato dall'osservazione che applicare la versione generalizzata allo scopo di dimostrare che $P(n)$ è vera per ogni $n \geq n_0$ è equivalente a dimostrare che la proposizione $P_{n_0}(n) \equiv P(n - n_0)$ è vera per ogni naturale. Applicare dunque la versione generalizzata, per un qualsiasi $n_0 \geq 0$, equivale ad applicare quella originale modificando leggermente la proposizione P qualora sia $n_0 > 0$.

Affinché il metodo sia applicato correttamente è necessario che il passo induttivo sia valido per *ogni* valore di $k \geq n_0$.

Esercizio 1.2 Dimostrare per induzione che $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ per $n \geq 1$.

Si noti che, applicando in modo non corretto il principio di induzione matematica, si può erroneamente giungere alla conclusione che in un branco di cavalli, tutti gli esemplari hanno il manto dello stesso colore. Si consideri il seguente ragionamento:

Passo base

Nel caso di un solo cavallo il predicato è senz'altro vero.

Passo induttivo

Si consideri un qualunque insieme di $n+1$ cavalli; allontanando momentaneamente un cavallo da tale branco, diciamo il cavallo c_i, otteniamo un branco di n cavalli che, per ipotesi induttiva, è costituito da cavalli con il manto di uno stesso colore α. Se riportiamo c_i nel branco, e successivamente allontaniamo $c_j \neq c_i$ otteniamo di nuovo un insieme di n cavalli che, per ipotesi induttiva, è costituito da cavalli con il manto di uno stesso colore β. Dunque c_i ha colore β e c_j ha colore α. Poiché nel branco ci sono $n-1$ cavalli che non sono mai stati allontanati, e poiché il manto di questi non può aver cambiato colore, ne segue necessariamente che $\alpha = \beta$ e che c_i, c_j e gli altri $n - 1$ cavalli hanno tutti il manto del medesimo colore.

Esercizio 1.3 Che errore è stato commesso nell'applicazione del principio di induzione?

Nel seguito utilizzeremo frequentemente dimostrazioni per induzione. Si noti che, in alcuni casi, si farà uso di una formulazione del principio di induzione più forte di quella fornita precedentemente. Tale principio viene chiamato *principio di induzione completa* ed afferma quanto segue.

> Data una proposizione $P(n)$ definita per un generico numero naturale $n \geq n_0$ si ha che essa è vera per tutti gli $n \geq n_0$ se:
>
> - $P(n_0)$ è vera (passo base dell'induzione);
> - per ogni naturale $k \geq n_0$, $P(i)$ vera per ogni i, $n_0 \leq i \leq k$ (ipotesi induttiva), implica $P(k + 1)$ vera (passo induttivo).

Esempio 1.2 Supponiamo di voler dimostrare la proposizione:

"Ogni intero $n \geq 2$ è divisibile per un numero primo".

Il passo base dell'induzione è costituito dal fatto che l'asserto è vero per $n = 2$.

Consideriamo ora un generico numero $k > 2$. Se k è composto, allora esso è divisibile per due interi k_1 e k_2 entrambi minori di k. Poichè per ipotesi induttiva sia k_1 che k_2 sono divisibili per un numero primo (ad esempio p_1 e p_2, rispettivamente), ne segue che anche k è divisibile per p_1 (e per p_2). Se, al contrario, k è un numero primo, allora esso è divisibile per se stesso. Ne segue dunque che l'asserto è verificato per ogni $n \geq 2$.

In molte applicazioni il principio di induzione completa risulta più comodo da utilizzare rispetto al principio di induzione matematica. Nell'esempio abbiamo infatti visto che la validità della proprietà di interesse per un intero k non sembra potersi semplicemente inferire dalla sola validità della proprietà per il valore $k - 1$.

Peraltro si può osservare che il principio di induzione matematica implica il principio di induzione completa, e viceversa.

Teorema 1.1 *Per ogni prefissata proprietà P il principio di induzione matematica ed il principio di induzione completa sono equivalenti.*

Dimostrazione. Supponiamo per semplicità che la base dell'induzione sia $n_0 = 0$. Si osservi che i due principi possono essere rispettivamente formulati nel seguente modo:

1. $\neg P(0) \vee \exists k'(P(k') \wedge \neg P(k'+1)) \vee \forall n P(n)$
2. $\neg P(0) \vee \exists k''(P(0) \wedge \ldots \wedge P(k'') \wedge \neg P(k''+1)) \vee \forall n P(n)$.

Chiaramente, se $P(0)$ è falso o è vero che $\forall n P(n)$, l'equivalenza è banalmente dimostrata.

Se, al contrario, $P(0)$ è vera ed è falso che $\forall n P(n)$ allora si deve dimostrare l'equivalenza dei termini intermedi delle disgiunzioni. Chiaramente dal fatto che esiste k'' tale che $P(0) \wedge \ldots \wedge P(k'') \wedge \neg P(k''+1)$ sia vero si può dedurre che esiste $k' = k''$ che verifica la congiunzione $P(k') \wedge \neg P(k'+1)$. Viceversa se disponiamo di k' che verifica $P(k') \wedge \neg P(k'+1)$ possiamo osservare che o $P(i)$ è vera per ogni i, $0 \leq i \leq k'$ e quindi in tal caso $k'' = k'$, o esiste un valore $0 \leq k'' < k'$ tale che $P(0) \wedge \ldots \wedge P(k'') \wedge \neg P(k''+1)$. □

Esercizio 1.4 I numeri di Fibonacci sono definiti mediante le seguenti regole: $F_0 = 1$, $F_1 = 1$ e $F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$.

Utilizzando il principio di induzione completa si dimostri che, per ogni $n \geq 2$, dato un qualsivoglia intero positivo k, $1 \leq k \leq n$, $F_n = F_k F_{n-k} + F_{k-1} F_{n-(k+1)}$.

1.1.3 Relazioni

Siano dati due insiemi A e B.

Definizione 1.3 *Il* prodotto cartesiano *di A e B, denotato con $A \times B$, è l'insieme*

$$C = \{\langle x, y\rangle \mid x \in A \wedge y \in B\},$$

cioè C è costituito da tutte le possibili coppie ordinate ove il primo elemento appartiene ad A ed il secondo a B[1]*.*

Il prodotto cartesiano gode della proprietà associativa, mentre non gode di quella commutativa.

Generalmente si usa la notazione A^n per indicare il prodotto cartesiano di A con se stesso, ripetuto n volte, cioè per indicare

$$\underbrace{A \times \cdots \times A.}_{n \text{ volte}}$$

Si noti che se $A = \emptyset$ o $B = \emptyset$ allora $A \times B = \emptyset$.

Definizione 1.4 *Una* relazione n-aria R *su $A_1, A_2, \ldots, A_n$ è un sottoinsieme del prodotto cartesiano $A_1 \times \cdots \times A_n$*

$$R \subseteq A_1 \times \cdots \times A_n.$$

Il generico elemento di una relazione viene indicato con il simbolo

$$\langle a_1, a_2, \ldots, a_n\rangle \in R,$$

oppure con il simbolo

$$R(a_1, \ldots, a_n);$$

n viene detto *arità* della relazione R. Nel caso delle relazioni binarie ($n = 2$) si usa anche la notazione $a_1 R a_2$.

Esempio 1.3 La relazione binaria "minore di" definita sui naturali, è l'insieme $R \subseteq \mathbb{N}^2$ definito da $R = \{\langle x, y\rangle \in \mathbb{N}^2 \mid \exists z \in \mathbb{N}(z \neq 0 \wedge x + z = y)\}$.

Esempio 1.4 La relazione "quadrato di" definita sui naturali è l'insieme $R \subseteq \mathbb{N}^2$

$$R = \big\{\langle x, y\rangle \mid x^2 = y\big\}.$$

Relazioni che godono di proprietà particolarmente interessanti sono le cosiddette relazioni d'ordine e le relazioni d'equivalenza.

[1] Quando considereremo una coppia di simboli, useremo le parentesi angolari se l'*ordine* con cui si considerano gli elementi è significativo, le normali parentesi tonde se l'ordine non è significativo. Così le coppie $\langle a, b\rangle$ e $\langle b, a\rangle$ sono distinte, mentre non lo sono (a, b) e (b, a)

Definizione 1.5 *Una relazione $R \subseteq A^2$ si dice* relazione d'ordine *se per ogni $x, y, z \in A$ valgono le seguenti proprietà*

1. $\langle x, x\rangle \in R$ (riflessività),
2. $\langle x, y\rangle \in R \wedge \langle y, x\rangle \in R \Longleftrightarrow x = y$ (antisimmetria),
3. $\langle x, y\rangle \in R \wedge \langle y, z\rangle \in R \iff \langle x, z\rangle \in R$ (transitività).

Un insieme A su cui è definita una relazione d'ordine viene detto *insieme parzialmente ordinato*.

Definizione 1.6 *Una relazione d'ordine $R \subseteq A^2$ tale che*

$$\langle a, b\rangle \in A^2 \Longleftrightarrow aRb \vee bRa,$$

dove cioè ogni elemento è in relazione con ogni altro elemento, si dice relazione di ordine totale.

Esempio 1.5 La relazione "$\leq$" è una relazione d'ordine su $\mathbb{N}$. In questo caso l'ordinamento è *totale* in quanto per ogni x ed y si ha che $\langle x, y\rangle \in R$ oppure che $\langle y, x\rangle \in R$, oppure valgono entrambe, nel qual caso si ha che $x = y$, per antisimmetria.

Esercizio 1.5 Dimostrare che la relazione "¡" su $\mathbb{N}$ non è una relazione d'ordine.

Definizione 1.7 *Una relazione $R \subseteq A^2$ si dice* relazione d'equivalenza *se, per ogni $x, y, z \in A$, valgono le seguenti proprietà*

1. $\langle x, x\rangle \in R$ (riflessività),
2. $\langle x, y\rangle \in R \Longleftrightarrow \langle y, x\rangle \in R$ (simmetria),
3. $\langle x, y\rangle \in R \wedge \langle y, z\rangle \in R \Longleftrightarrow \langle x, z\rangle \in R$ (transitività).

Esempio 1.6 Consideriamo l'insieme delle coppie $\langle n, m\rangle$ con $n \in \mathbb{N}$ ed $m \in \mathbb{N}^+$. La relazione

$$E = \big\{\langle\langle u, v\rangle, \langle p, q\rangle\rangle \mid uq = vp\big\}$$

è una relazione d'equivalenza.

Un insieme A su cui sia definita una relazione d'equivalenza R si può partizionare in sottoinsiemi, detti *classi d'equivalenza*, ciascuno dei quali è un sottoinsieme massimale che contiene solo elementi tra loro equivalenti. Dati un insieme A ed una relazione d'equivalenza R su A^2, l'insieme delle classi d'equivalenza di A rispetto a R è detto insieme *quoziente*, e viene normalmente denotato con A/R. I suoi elementi vengono denotati con $[a]$, dove $a \in A$ è un "rappresentante" della classe d'equivalenza: $[a]$ denota cioè l'insieme degli elementi equivalenti ad a.

Esempio 1.7 Dato un intero k, definiamo la relazione $\equiv_k$, detta *congruenza modulo* k, su $\mathbb{N}^2$ nel seguente modo:

$$n \equiv_k m \Longleftrightarrow \text{esistono } q, q', r, \text{ con } 0 \leq r < k, \text{ tali che } \begin{cases} n = qk + r \\ m = q'k + r \end{cases}$$

Essa è una relazione d'equivalenza e le sue classi d'equivalenza sono dette *classi resto* rispetto alla divisione per k.

Definizione 1.8 *Data una relazione d'equivalenza $R \subseteq A^2$, si dice* indice *di R, e si denota con* $\mathrm{ind}(R)$*, il numero di elementi di A/R.*

Esercizio 1.6 Dato un intero k, qual è l'indice della relazione $\equiv_k$?

Di particolare interesse godono, in informatica, le relazioni rappresentabili mediante grafi.

Definizione 1.9 *Dato un insieme finito V ed una relazione binaria $E \subseteq V \times V$, la coppia $\langle V, E \rangle$ si definisce* grafo orientato.

Per visualizzare un grafo orientato si usa rappresentare gli elementi di V con punti (detti *nodi*) e le coppie $\langle i, j \rangle$ di E con frecce da i a j (dette *archi*).

Esempio 1.8 Sia dato l'insieme $V = \{A, B, C, D\}$. Consideriamo la relazione

$$E = \{\langle A, B\rangle, \langle A, C\rangle, \langle A, D\rangle, \langle B, C\rangle, \langle C, D\rangle, \langle B, B\rangle, \langle B, A\rangle\}.$$

Essa si può rappresentare con il grafo orientato di Figura 1.1.

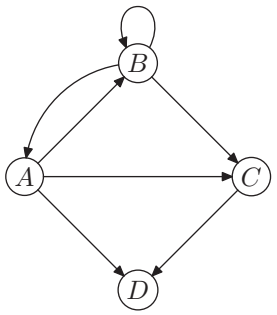


FIGURA 1.1 Esempio di relazione rappresentata mediante grafo orientato.

Se la relazione E gode della proprietà simmetrica il grafo può essere rappresentato senza assegnare un orientamento agli archi. In tal caso la coppia $\langle V, E \rangle$ si definisce *grafo non orientato* o semplicemente *grafo*.

1.1.4 Operazioni tra relazioni

Le operazioni che si possono definire per le relazioni non differiscono molto da quelle note per insiemi generici. Ad esempio l'unione è definita da

$$R_1 \cup R_2 = \{\langle x, y\rangle \mid \langle x, y\rangle \in R_1 \vee \langle x, y\rangle \in R_2\},$$

mentre la complementazione è definita da

$$\overline{R} = \{\langle x, y\rangle \mid \langle x, y\rangle \notin R\}.$$

Un'operazione che possiede caratteristiche interessanti è la chiusura transitiva.

Definizione 1.10 *Sia R una relazione su A^2; si definisce* chiusura transitiva *di R, denotata con R^+, la relazione*

$$R^+ = \{\langle x, y\rangle \mid \exists y_1, \ldots, y_n \in A, \text{ con } n \geq 2, y_1 = x, y_n = y, \text{ tali che } \langle y_i, y_{i+1}\rangle \in R, i = 1, \ldots, n-1\}.$$

In alcuni casi è interessante definire la chiusura transitiva e riflessiva.

Definizione 1.11 *Sia R una relazione su A^2; si definisce* chiusura transitiva e riflessiva *di R, denotata con R^*, la relazione*

$$R^* = R^+ \cup \{\langle x, x\rangle \mid x \in A\}.$$

Esercizio 1.7 Sia dato un grafo orientato $G = \langle V, E\rangle$ in cui V è l'insieme dei nodi ed $E \subseteq V \times V$ è l'insieme degli archi orientati. Dimostrare che se R è la relazione tale che xRy se e solo se $x = y$ oppure è possibile raggiungere y a partire da x percorrendo gli archi secondo il loro orientamento, allora R è la chiusura transitiva e riflessiva della relazione E.

Esercizio 1.8 Dimostrare che dato un insieme finito V ed una qualsiasi relazione binaria simmetrica E definita su V^2, E^* è una relazione d'equivalenza.

Esercizio 1.9 Dato un grafo orientato $G = \langle V, E\rangle$, cosa rappresentano le classi d'equivalenza del grafo $G^* = \langle V, E^*\rangle$?

1.1.5 Funzioni

Un particolare tipo di relazioni sono le relazioni funzionali.

Definizione 1.12 *Si dice che $R \subseteq X_1 \times \ldots \times X_n$ ($n \geq 2$) è una* relazione funzionale *tra una $(n-1)$-pla di elementi e l'n-esimo elemento, se $\forall\langle x_1, \ldots, x_{n-1}\rangle \in X_1 \times \ldots \times X_{n-1}$ esiste al più un elemento $x_n \in X_n$ tale che $\langle x_1, \ldots, x_n\rangle \in R$.*

In tal caso si definisce *funzione* (o anche *applicazione*, o *corrispondenza univoca*) la legge che all'elemento $\langle x_1, \ldots, x_{n-1}\rangle \in X_1 \times \ldots \times X_{n-1}$ associa, se esiste, quell'unico elemento $x_n \in X_n$ tale che $x_1, \ldots, x_n \in R$. La notazione usata per indicare l'esistenza di una relazione di tipo funzionale tra gli elementi $x_1, \ldots, x_n$ è

$$f(x_1, \ldots, x_{n-1}) = x_n.$$

In tal caso diciamo che la funzione f ha arità $n-1$. Per convenzione si ammette che una funzione possa anche avere arità 0. In tal modo un valore costante a può essere considerato come il risultato della applicazione di una funzione 0-aria: $f() = a$.

La notazione generalmente usata per indicare tra quali insiemi viene realizzata la corrispondenza, cioè per stabilire il ruolo degli insiemi in gioco è:

$$f : X_1 \times \cdots \times X_{n-1} \mapsto X_n.$$

L'espressione $X_1 \times \cdots \times X_{n-1} \mapsto X_n$ viene detta *tipo* della funzione f[2]. Nel caso in cui $X_1 = \cdots = X_{n-1} = X_n = X$ il tipo della funzione può essere indicato da $X^{n-1} \mapsto X$.

Definizione 1.13 *Data una funzione* $f : X_1 \times \cdots \times X_{n-1} \mapsto X_n$*, l'insieme* $X_1 \times \cdots \times X_{n-1}$ *viene detto* dominio *della funzione,* $\mathrm{dom}(f)$*, e l'insieme* X_n *viene detto* codominio*,* $\mathrm{cod}(f)$*.*

Definizione 1.14 *Data una funzione* $f : X_1 \times \cdots \times X_{n-1} \mapsto X_n$ *si chiama* dominio di definizione *della funzione* f*, e lo si indica con la notazione* $\mathrm{def}(f)$*, il sottoinsieme di* $\mathrm{dom}(f)$ *così definito:*

$$\mathrm{def}(f) = \{\langle x_1, \ldots, x_{n-1}\rangle \in \mathrm{dom}(f) \ \mid \ \exists x_n \in \mathrm{cod}(f) f(x_1, \ldots, x_{n-1}) = x_n\}$$

Il dominio di definizione di f è cioè l'insieme di tutte le $(n-1)$-ple $x_1, \ldots, x_{n-1}$ per cui $f(x_1, \ldots, x_{n-1})$ è definita.

Definizione 1.15 *Si definisce* immagine *della funzione* f*, e lo si indica con la notazione* $\mathrm{imm}(f)$*, il sottoinsieme di* X_n *così definito:*

$$\mathrm{imm}(f) = \{x_n \in X_n \ \mid \ \exists \langle x_1, \ldots, x_{n-1}\rangle \in \mathrm{dom}(f) f(x_1, \ldots, x_{n-1}) = x_n\}.$$

L'immagine di f è quindi l'insieme di tutti i valori assunti da f.

Definizione 1.16 *Data una funzione* f *e considerato un generico elemento* $x_n \in \mathrm{cod}(f)$ *si dice* controimmagine *(o* fibra*) di* x_n*, e lo si indica con* $f^{-1}(x_n)$*, il seguente sottoinsieme del dominio di* f

$$\begin{aligned} f^{-1}(x_n) \ &= \ \{\langle x_1, \ldots x_{n-1}\rangle \ \mid \ \langle x_1, \ldots x_{n-1}\rangle \in \mathrm{def}(f) \wedge \\ &\qquad f(x_1, \ldots x_{n-1}) = x_n\}. \end{aligned}$$

[2]Si noti che frequentemente, nell'ambito dei linguaggi di programmazione, per "tipo di una funzione" si intende il tipo del valore restituito da essa.

La controimmagine di x_n è cioè l'insieme di tutte le $(n-1)$-ple per cui f assume valore x_n.

Si noti che il dominio di definizione di una funzione può o meno coincidere con il dominio. Infatti, in molti casi è necessario considerare funzioni che per alcuni valori del dominio non sono definite. Ad esempio la funzione $f(x) = 1/\sin(x)$ ha come dominio l'insieme $\mathbb{R}$ ma come dominio di definizione l'insieme $\mathbb{R} - \{k\pi \mid k \in \mathbb{Z}\}$.

Definizione 1.17 *Una funzione $f : X_1 \times \ldots \times X_{n-1} \mapsto X_n$ viene detta* totale *se* $\mathrm{def}(f) = \mathrm{dom}(f)$. *Nel caso più generale in cui* $\mathrm{def}(f) \subseteq \mathrm{dom}(f)$, *$f$ viene detta funzione* parziale.

Chiaramente tutte le funzioni sono parziali (anche se ciò non viene sempre esplicitamente dichiarato) e le funzioni totali sono un caso particolare. Analogamente a quanto accade per il dominio di una funzione possiamo osservare che anche l'immagine di una funzione può coincidere o meno con il codominio relativo.

Definizione 1.18 *Una funzione $f : X_1 \times \cdots \times X_{n-1} \mapsto X_n$ viene detta* suriettiva *se*

$$\mathrm{imm}(f) = \mathrm{cod}(f).$$

Definizione 1.19 *Una funzione f si dice* iniettiva *o* uno-ad-uno (1:1) *se fa corrispondere ad elementi diversi del dominio di definizione elementi diversi del codominio, come esplicitato di seguito*

$$\forall\langle x'_1, \ldots, x'_{n-1}\rangle \in X_1 \times \ldots \times X_{n-1}, \forall\langle x''_1, \ldots, x''_{n-1}\rangle \in X_1 \times \ldots \times X_{n-1},$$
$$\langle x'_1, \ldots, x'_{n-1}\rangle \neq \langle x''_1, \ldots, x''_{n-1}\rangle \Longleftrightarrow f(x'_1, \ldots, x'_{n-1}) \neq f(x''_1, \ldots, x''_{n-1}).$$

Equivalentemente, possiamo anche notare che, in tal caso,

$$f(x'_1, \ldots, x'_{n-1}) = f(x''_1, \ldots, x''_{n-1}) \Longleftrightarrow \langle x'_1, \ldots, x'_{n-1}\rangle = \langle x''_1, \ldots, x''_{n-1}\rangle.$$

Naturalmente la fibra di ogni elemento dell'immagine di una funzione iniettiva ha cardinalità unitaria.

Definizione 1.20 *Una funzione si dice* biiettiva *o* biunivoca *se è allo stesso tempo iniettiva, suriettiva e totale. Una funzione biiettiva si dice anche* biiezione.

Esercizio 1.10 Dimostrare che esiste una biiezione tra l'insieme dei sottoinsiemi di un insieme S finito di cardinalità $|S| = n$ e le sequenze binarie di lunghezza n.

Un'importante proprietà delle funzioni iniettive è espressa dal seguente teorema[3].

[3] Il nome letteralmente significa "Principio della piccionaia" e si riferisce all'impossibilità che due piccioni occupino la stessa cella di una piccionaia. Per questo teorema, come d'altronde accadrà per altri teoremi nel corso del volume, manteniamo la dicitura inglese perché con tale dicitura il "principio" è universalmente noto e citato.

Teorema 1.2 (Pigeonhole Principle) *Dati due insiemi finiti A e B, tali che*

$$0 <| B |<| A |,$$

non esiste alcuna funzione iniettiva totale $f : A \mapsto B$.

Dimostrazione. Si procede per induzione, ragionando sulla cardinalità dell'insieme B.

Passo base

Se $| B |= 1$ (cioè $B = \{b\}$ per un opportuno elemento b) allora $| A |\geq 2$ ($A = \{a_1, a_2, \ldots\}$); per qualunque funzione totale f deve necessariamente valere $f(a_1) = f(a_2) = b$. cioè f non è iniettiva.

Passo induttivo

Supponiamo il teorema vero per $| B |\leq n$ e consideriamo il caso in cui $| B |= n + 1$ e $| A |\geq n + 2$. Scelto arbitrariamente un elemento $b \in B$ si hanno due casi:

- Se $| f^{-1}(b) |\geq 2$, il teorema è dimostrato in quanto la f non è iniettiva.
- Se $| f^{-1}(b) |\leq 1$, allora si considerino gli insiemi $A' = A - \{f^{-1}(b)\}$ e $B' = B - \{b\}$. Vale naturalmente $| A' |\geq n + 1 >| B' |= n$. Si consideri inoltre la funzione $f' : A' \mapsto B'$ tale che $\forall a \in A'\ f'(a) = f(a)$ (detta *restrizione* della f ad A'). Per ipotesi induttiva f' non può essere iniettiva, il che equivale a dire che esistono $a', a'' \in A'$ tali che $a' \neq a''$ e $f'(a') = f'(a'')$. Dato che, per costruzione, abbiamo che $a', a'' \in A$, $f'(a') = f(a')$ e $f'(a'') = f(a'')$, ne deriva che esistono $a', a'' \in A$ tali che $a' \neq a''$ e $f(a') = f(a'')$.

□

Si noti che il Pigeonhole principle esclude, in particolare, che si possa avere una biiezione tra un insieme A ed un suo sottoinsieme proprio $B \subset A$. È importante osservare che tale situazione non si verifica invece se consideriamo insiemi infiniti. Vedremo infatti, nella prossima sezione, che è possibile stabilire una corrispondenza biunivoca tra gli elementi di un insieme infinito (ad esempio i naturali) e quelli di un suo sottoinsieme proprio (ad esempio i numeri pari).

Esercizio 1.11 Dato un insieme finito B definiamo k-partizione una famiglia di k insiemi non vuoti $A_1, \ldots, A_k$ tali che $A_1 \cup \cdots \cup A_k = B$ e $\forall i, j$ se $i \neq j$ allora $A_i \cap A_j = \emptyset$. Dimostrare che se $| B |> 2$ non esiste una biiezione tra le 2-partizioni e le 3-partizioni di B.

Esercizio 1.12 Dimostrare che in ogni grafo semplice (privo cioè di archi multipli fra coppie di nodi e di archi riflessivi) esistono almeno due nodi con lo stesso grado (con lo stesso numero di archi incidenti).

1.1.6 Cardinalità di insiemi infiniti e numerabilità

All'inizio del Capitolo abbiamo introdotto informalmente la definizione di cardinalità di un insieme infinito (vedi Sezione 1.1.1). Al fine di poter estendere agli insiemi infiniti il concetto di cardinalità si rende ora necessario definire in modo più formale tale concetto, basandoci sulla possibilità di confrontare la "numerosità" di due insiemi.

Definizione 1.21 *Due insiemi A e B si dicono* equinumerosi *se esiste una biiezione tra di essi.*

Esempio 1.9 Gli insiemi

$$\{\text{lunedì}, \text{martedì}, \ldots, \text{domenica}\} \text{ e } \{5, 7, 31, 50, 64, 70, 75\}$$

sono equinumerosi.

Esercizio 1.13 Dimostrare che la relazione di equinumerosità è una relazione d'equivalenza.

Prima di considerare come il concetto di equinumerosità ci permette di definire la cardinalità di insiemi infiniti, osserviamo che, utilizzando tale concetto, è possibile ridefinire in modo formale la cardinalità di insiemi finiti, nel seguente modo.

Definizione 1.22 *Dato un insieme finito A, la sua cardinalità $|A|$ è così definita:*

$$|A| = \begin{cases} 0 & \textit{se } A = \emptyset \\ n & \text{se } A \text{ è equinumeroso a } \{0, 1, \ldots, n-1\}, \text{ con } n \geq 1. \end{cases}$$

Il concetto di cardinalità può essere utilizzato per dare una definizione insiemistica del concetto di "numero cardinale": il numero n può infatti essere definito come la classe d'equivalenza di tutti gli insiemi equinumerosi a $\{0, \ldots, n-1\}$. In particolare, il numero 0 è la classe d'equivalenza dell'insieme vuoto.

Definizione 1.23 *Un insieme si dice* numerabile *se esso è equinumeroso a* $\mathbb{N}$.

Definizione 1.24 *Un insieme si dice* contabile *se esso è finito o numerabile.*

Mentre nel caso degli insiemi finiti la cardinalità può essere espressa appunto dal numero cardinale corrispondente, nel caso di insiemi infiniti si fa generalmente uso di simboli aggiuntivi introdotti *ad hoc*. In particolare, per indicare la cardinalità degli insiemi infiniti equinumerosi ad $\mathbb{N}$ si utilizza il simbolo $\aleph_0$ (aleph[4] zero)

In base alle definizioni poste si può facilmente dimostrare il seguente risultato.

Teorema 1.3 *Se un insieme A è equinumeroso a un insieme B, con $B \subseteq C$, dove C è un insieme contabile, allora anche A è contabile.*

Dimostrazione. La dimostrazione è lasciata come esercizio al lettore (Esercizio 1.14). □

Esercizio 1.14 Dimostrare il Teorema 1.3.

Esempio 1.10 L'insieme $\mathbb{Z}$ degli interi relativi risulta essere numerabile (cioè $|\mathbb{Z}| = \aleph_0$) poiché i suoi elementi possono essere posti in corrispondenza biunivoca con $\mathbb{N}$ tramite la biiezione $f : \mathbb{Z} \mapsto \mathbb{N}$ definita nel seguente modo:

$$f(i) = \begin{cases} -2i & \text{se } i \leq 0 \\ 2i - 1 & \text{se } i > 0. \end{cases}$$

Il precedente esempio ci permette di osservare un'importante caratteristica degli insiemi infiniti. Infatti l'insieme $\mathbb{N}$ è propriamente contenuto nell'insieme $\mathbb{Z}$: ciononostante i due insiemi risultano equinumerosi. Si osservi che ciò non si può verificare nel caso di insiemi finiti come visto in base al "Pigeonhole principle".

Esempio 1.11 L'insieme delle coppie di naturali $\mathbb{N}^2$ è numerabile. La corrispondenza biunivoca può essere stabilita con la seguente biiezione, frequentemente chiamata *funzione coppia di Cantor*

$$p(i, j) = \frac{(i + j)(i + j + 1)}{2} + i.$$

Il metodo con cui la biiezione p pone in corrispondenza coppie di naturali con i naturali è illustrato in Figura 1.2.

Più in generale è possibile mostrare che, per ogni $n \in \mathbb{N}$, se A è contabile, anche A^n lo è.[5]

Il procedimento di enumerazione illustrato in Figura 1.2 fu originariamente introdotto da Cantor per mostrare la numerabilità dell'insieme $\mathbb{Q}$ dei numeri

[4] Prima lettera dell'alfabeto ebraico.

[5] La funzione $p(i, j)$ è frequentemente denominata *funzione coppia di Cantor*.

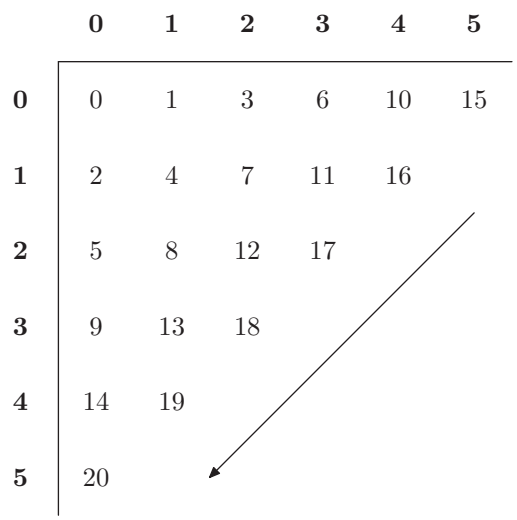


FIGURA 1.2 Dimostrazione della numerabilità delle coppie di naturali.

razionali. A tal fine dobbiamo innanzi tutto ricordare che i numeri razionali corrispondono alle classi d'equivalenza della relazione binaria R definita sull'insieme $\mathbb{Z} \times (\mathbb{Z}^+)$:

$R(\langle a, b\rangle, \langle c, d\rangle)$ se e solo se $ad = bc$.

Per tale motivo, l'insieme $\mathbb{Q}$ è dunque equinumeroso all'insieme $\mathbb{Z} \times (\mathbb{Z}^+)/_R$. D'altronde poiché $\mathbb{Z}$ è contabile, anche $\mathbb{Z}^2$ lo è e così anche l'insieme $\mathbb{Z} \times (\mathbb{Z}^+)/_R$ che è equinumeroso ad un sottoinsieme proprio di $\mathbb{Z}^2$. Ciò prova che $\mathbb{Q}$ è contabile. Si noti che il procedimento di Cantor può essere ulteriormente generalizzato per mostrare il seguente risultato.

Teorema 1.4 *L'unione di una quantità contabile di insiemi contabili è ancora un insieme contabile.*

Dimostrazione. L'enumerazione può essere effettuata applicando ancora il metodo di Cantor, come si può vedere in Figura 1.3, dove si suppone che la riga i-esima contenga gli elementi dell'i-esimo insieme. □

Esercizio 1.15 Si consideri l'insieme delle equazioni di grado intero, a coefficienti interi:

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = 0.$$

L'insieme dei numeri algebrici reali è per definizione l'insieme delle soluzioni reali di tali equazioni. Dimostrare che tale insieme è numerabile.

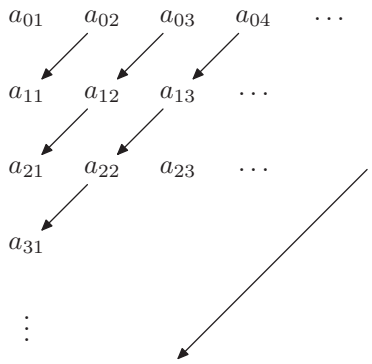


FIGURA 1.3 Tecnica per enumerare gli elementi dell'unione di una quantità contabile di insiemi contabili.

1.1.7 Insiemi non numerabili

Dopo aver introdotto alcuni esempi di insiemi numerabili, è utile mostrare l'esistenza di insiemi non numerabili. A tal fine useremo una tecnica che sarà ripetutamente utilizzata in questo volume, la cosiddetta *tecnica di diagonalizzazione*, introdotta da Cantor proprio per illustrare la non numerabilità dei numeri reali, e diffusamente utilizzata per provare risultati importanti nella teoria della calcolabilità.

Data una lista di oggetti, la tecnica di diagonalizzazione consiste nel creare un controesempio, cioè un oggetto non appartenente alla lista, mediante un procedimento che deliberatamente lo costruisce garantendo che esso sia diverso da tutti gli altri oggetti appartenenti alla lista; tale oggetto è appunto detto *oggetto diagonale*.

In termini più formali, data una sequenza $\Phi_0, \Phi_1, \ldots$, dove ciascun Φ_i è a sua volta una sequenza di infiniti elementi $a_{i0}, a_{i1}, \ldots, a_{in}, \ldots$, la diagonalizzazione consiste nel creare una sequenza diagonale Φ che si differenzia, per costruzione, da ogni Φ_i. In particolare, l'elemento i-esimo di Φ sarà diverso dall'elemento i-esimo a_{ii} di Φ_i.

Teorema 1.5 *L'insieme* $\mathbb{R}$ *dei reali non è numerabile.*

Dimostrazione. Innanzi tutto osserviamo che l'insieme aperto $(0, 1)$ e l'insieme $\mathbb{R}$ sono equinumerosi (una possibile biiezione è $1/(2^x+1)$, che ha dominio $\mathbb{R}$ e codominio $(0, 1)$). Basta dunque mostrare che l'insieme dei reali in $(0, 1)$ non è numerabile. A tal fine, consideriamo dapprima l'insieme delle sequenze infinite di cifre decimali che rappresentano la mantissa dei reali in $(0, 1)$ e

mostriamo che tale insieme non è numerabile. Per farlo si supponga per assurdo di aver trovato una qualsiasi corrispondenza tra i naturali e le sequenze: questa corrispondenza definirebbe una enumerazione Φ_i.

Introduciamo ora la sequenza Φ avente come i-esima cifra, per $i = 0, 1, 2, \ldots$, il valore ottenuto sommando $1(\bmod 10)$ alla i-esima cifra di Φ_i. Con riferimento, ad esempio, alla Figura 1.4 si ottiene $\Phi = 124\ldots$.

La sequenza Φ viene a costituire elemento diagonale dell'enumerazione $\Phi_0, \Phi_1, \ldots$ in quanto differisce da ogni altra sequenza Φ_i nella posizione i.

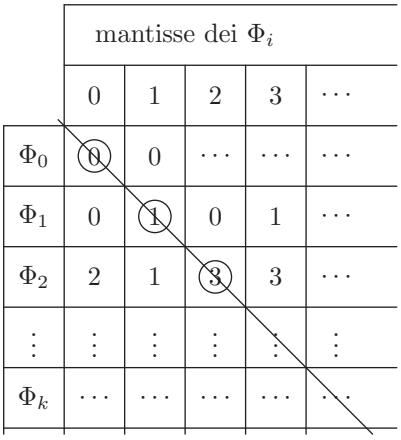


FIGURA 1.4 Costruzione per diagonalizzazione del numero reale $0.124\ldots$ non appartenente all'enumerazione.

In altre parole, dopo aver supposto per assurdo di poter enumerare tutte le rappresentazioni decimali di reali nell'intervallo $(0, 1)$, è stato possibile costruire per diagonalizzazione un'ulteriore rappresentazione che, seppure relativa ad un reale in $(0, 1)$, non appartiene all'enumerazione, il che contrasta con l'ipotesi che l'insieme delle rappresentazioni dei reali sia numerabile.

La non numerabilità dei reali in $(0, 1)$ deriva da quanto detto ed osservando inoltre che ogni numero reale ha al più due rappresentazioni distinte (ad esempio, $0.01000\ldots$ e $0.00999\ldots$) □

Un secondo importante esempio di insieme non numerabile è l'insieme delle parti di $\mathbb{N}$, $\mathcal{P}(\mathbb{N})$.

Teorema 1.6 *L'insieme delle parti di* $\mathbb{N}$, $\mathcal{P}(\mathbb{N})$, *non è numerabile.*

Dimostrazione. Supponiamo per assurdo che $\mathcal{P}(\mathbb{N})$ sia numerabile e sia

$P_0, P_1, \ldots$ una sua enumerazione. A ciascun P_i, con $i = 0, 1, 2, \ldots$, associamo una sequenza b_{i0}, b_{i1}, b_{i2}, $\ldots$, dove

$$b_{ij} = \begin{cases} 0 \text{ se } j \notin P_i \\ 1 \text{ se } j \in P_i. \end{cases}$$

Costruiamo ora l'insieme diagonale P. La sequenza associata a P è $p_0, p_1, \ldots$, dove $p_i = 1 - b_{ii}$ per $i = 0, 1, 2, \ldots$. L'insieme P è perfettamente definito dalla sequenza $p_0, p_1, \ldots$, ma differisce da ciascuno degli insiemi P_i poiché, per costruzione, $i \in P \Longleftrightarrow i \notin P_i$. Avendo dunque supposto che sia possibile enumerare gli elementi di $\mathcal{P}(\mathbb{N})$, si è riusciti a costruire un insieme $P \in \mathcal{P}(\mathbb{N})$ che non fa parte della enumerazione, il che falsifica tale ipotesi. □

Chiaramente il risultato di non numerabilità ora trovato non vale solo per l'insieme delle parti di $\mathbb{N}$, ma per l'insieme delle parti di ogni insieme di cardinalità $\aleph_0$.

1.1.8 Cardinalità transfinita

Nella sezione precedente abbiamo visto esempi di insiemi la cui cardinalità è superiore ad $\aleph_0$. Vediamo ora come possa essere caratterizzata la cardinalità di tali insiemi. A tal fine consideriamo innanzitutto la seguente classe di funzioni a valore binario.

Definizione 1.25 *Si definisce* funzione caratteristica $f_S(x)$ *di un insieme* $S \subseteq \mathbb{N}$ *la funzione totale* $f_S : \mathbb{N} \mapsto \{0, 1\}$

$$f_S(x) = \begin{cases} 1 & \textit{se } x \in S \\ 0 & \textit{se } x \notin S. \end{cases} \tag{1.1}$$

È utile introdurre questa classe di funzioni perché ciò consente di ricondurre il problema del riconoscimento di insiemi al problema del calcolo di una funzione.

Si può facilmente mostrare che anche l'insieme delle funzioni caratteristiche su $\mathbb{N}$ non è numerabile. Per dimostrarlo si usa, al solito, il procedimento diagonale.

Teorema 1.7 *L'insieme delle funzioni caratteristiche* $\{f \mid f : \mathbb{N} \mapsto \{0, 1\}\}$ *non è numerabile.*

Dimostrazione. Si supponga per assurdo di avere una corrispondenza tra l'insieme delle funzioni caratteristiche e i naturali, come in Figura 1.5. A partire dalle funzioni di questa enumerazione, si può costruire una nuova funzione, definita come di seguito:

$$\hat{f}(i) = 1 - f_i(i).$$

Questa $\hat{f}(i)$ assume ovviamente solo valori in $\{0, 1\}$. La funzione così costruita è evidentemente diversa da tutte quelle enumerate per almeno un valore dell'argomento, il che dimostra l'asserto. □

	0	1	2	3	4	$\cdots$	j	$\cdots$
f_0	0	0	0	0	0	$\cdots$	$f_0(j)$	$\cdots$
f_1	0	1	0	1	0	$\cdots$	$f_1(j)$	$\cdots$
f_2	0	1	1	0	1	$\cdots$	$f_2(j)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	
f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	$f_i(3)$	$f_i(4)$	$\cdots$	$f_i(j)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	

FIGURA 1.5 Enumerazione delle funzioni caratteristiche.

Il risultato trovato è, in effetti, ovvio, nel momento in cui si osserva che esiste una corrispondenza biunivoca naturale tra sottoinsiemi di $\mathbb{N}$ e funzioni caratteristiche.

Come abbiamo visto in Sezione 1.1.1, dato un insieme finito A di cardinalità n, l'insieme $\mathcal{P}(A)$ ha cardinalità 2^n. Per analogia, dato un insieme A numerabile, e quindi di cardinalità $\aleph_0$, si dice che l'insieme $\mathcal{P}(A)$ ha cardinalità $2^{\aleph_0}$ (o, con un leggero abuso di notazione, $2^{\mathbb{N}}$). Gli insiemi aventi cardinalità $2^{\aleph_0}$ vengono detti insiemi *continui*. L'osservazione precedente circa la corrispondenza tra l'insieme $\mathcal{P}(\mathbb{N})$ e l'insieme $\{f \mid f : \mathbb{N} \mapsto \{0,1\}\}$ ci permette di asserire che quest'ultimo insieme è un insieme continuo.

Tale risultato può essere in realtà esteso a tutte le funzioni intere. Infatti da un lato vale che:

$$|\{f \mid f : \mathbb{N} \mapsto \mathbb{N}\}| \geq |\{f \mid f : \mathbb{N} \mapsto \{0,1\}\}| = 2^{\aleph_0}.$$

D'altra parte ogni funzione totale intera può essere vista come un insieme di coppie di interi, cioè come un elemento di $\mathcal{P}(\mathbb{N}^2)$, e poiché sappiamo che $\mathbb{N}^2$ è equinumeroso a $\mathbb{N}$, è chiaro che l'insieme delle funzioni intere ha cardinalità non superiore a $2^{\aleph_0}$. Ne risulta che $|\{f \mid f : \mathbb{N} \mapsto \mathbb{N}\}| = 2^{\aleph_0}$.

Si noti che il problema dell'esistenza di un insieme avente cardinalità compresa tra $\aleph_0$ e $2^{\aleph_0}$ (noto come *problema del continuo*) è tuttora un problema aperto.[6] Raffiniamo ora il risultato del Teorema 1.5 dimostrando che l'insieme dei reali ha cardinalità $2^{\aleph_0}$.

[6] Se chiamiamo $\aleph_1$ il primo cardinale transfinito maggiore di $\aleph_0$ il problema del continuo si può formulare come il problema di stabilire se $\aleph_1 = 2^{\aleph_0}$.

Teorema 1.8 *L'insieme* $\mathbb{R}$ *dei reali è un insieme continuo.*

Dimostrazione. È sufficiente mostrare che tale proprietà è goduta dall'insieme dei reali compresi fra 0 e 1, poiché, come abbiamo visto, esiste una corrispondenza biunivoca tra $\mathbb{R}$ e l'insieme dei reali in $(0,1)$ (vedi dimostrazione del Teorema 1.5). Osserviamo che ad ogni sottoinsieme dei naturali è possibile associare una unica stringa binaria che ne descrive la funzione caratteristica e che tale stringa, a sua volta, può essere vista come la mantissa di un unico reale in $(0,1)$. D'altro canto, come osservato nella dimostrazione del Teorema 1.5, ad ogni reale corrispondono al più due diverse mantisse e quindi due diversi sottoinsiemi dei naturali. Ne deriva quindi che i reali in $(0,1)$ sono almeno la metà dei sottoinsiemi dei naturali e non più di questi ultimi. Il teorema è dimostrato constatando che la presenza di un fattore moltiplicativo finito non modifica una cardinalità transfinita. □

Per indicare la cardinalità degli insiemi infiniti sono stati introdotti da Cantor i cosiddetti *cardinali transfiniti*. $\aleph_0$ e $2^{\aleph_0}$ sono esempi di cardinali transfiniti. Cantor ha anche mostrato che esistono infiniti cardinali transfiniti, in base alla proprietà che se un insieme A ha cardinalità transfinita ϕ, l'insieme $\mathcal{P}(A)$ ha cardinalità transfinita 2^{ϕ}. Tale risultato è basato su un uso della tecnica di diagonalizzazione estesa ai numeri transfiniti.

Teorema 1.9 *L'insieme delle funzioni reali* $\{f \mid f : \mathbb{R} \mapsto \mathbb{R}\}$ *ha cardinalità* $2^{2^{\aleph_0}}$.

Dimostrazione. Poiché ad ogni elemento dell'insieme

$$F = \{f \mid f : \mathbb{R} \mapsto \mathbb{R}\}$$

si può associare un elemento di $\mathcal{P}(\mathbb{R}^2)$ abbiamo che $|F| \leq |\mathcal{P}(\mathbb{R}^2)|$. D'altra parte $\mathbb{R}^2$ è equinumeroso a $\mathbb{R}$: infatti basta verificare, per quanto visto nella dimostrazione del Teorema 1.5, che $|(0,1)| = |(0,1)^2|$. Ciò è sicuramente vero, se consideriamo la biiezione che associa ad ogni elemento a di $(0,1)$ la coppia di $(0,1)^2$ avente per mantisse le cifre ottenute estraendo le cifre dispari e pari, rispettivamente, della mantissa di a. Dunque

$$|F| \leq |\mathcal{P}(\mathbb{R}^2)| = |\mathcal{P}(\mathbb{R})| = 2^{2^{\aleph_0}}.$$

Possiamo poi osservare che esiste una ovvia corrispondenza biunivoca fra $\mathcal{P}(\mathbb{R})$ e $F' = \{f \mid f : \mathbb{R} \mapsto \{0,1\}\} \subset F$. Ne segue che $|F| \geq |F'| = 2^{2^{\aleph_0}}$.

Da quanto visto, risulta che deve necessariamente essere $|F| = 2^{2^{\aleph_0}}$. □

In Tabella 1.2 sono riassunti vari risultati di cardinalità ottenuti su insiemi di interesse.

insiemi	cardinalità
$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{N}^n$, $\mathbb{Z}^n$, $\mathbb{Q}^n$	$\aleph_0$
$\mathcal{P}(\mathbb{N})$, $\{f \mid f : \mathbb{N} \mapsto \{0,1\}\}$, $\{f \mid f : \mathbb{N} \mapsto \mathbb{N}\}$, $\mathbb{R}$, $\mathbb{R}^n$	$2^{\aleph_0}$
$\mathcal{P}(\mathbb{R})$, $\{f \mid f : \mathbb{R} \mapsto \{0,1\}\}$, $\{f \mid f : \mathbb{R} \mapsto \mathbb{R}\}$	$2^{2^{\aleph_0}}$

Tabella 1.2 Cardinalità di alcuni insiemi notevoli.

Esercizio 1.16 Determinare la cardinalità dei seguenti insiemi:

1. l'insieme delle relazioni n-arie, per n qualunque, su un insieme finito A,
2. l'insieme delle relazioni n-arie, per n qualunque, sull'insieme dei naturali,
3. l'insieme delle relazioni binarie sull'insieme dei reali.

1.2 Strutture algebriche elementari

Nel seguito della trattazione avremo necessità di considerare insiemi su cui sono definite operazioni aventi particolari proprietà. Come si ricorda, un insieme chiuso rispetto a una o più specifiche operazioni è generalmente definito *algebra*. Prima di tutto è dunque utile richiamare definizioni e proprietà delle strutture algebriche cui si farà riferimento nel seguito.

1.2.1 Semigruppi, monoidi, gruppi, semianelli

Supponiamo dato un insieme universo U, che per ipotesi contiene tutti gli elementi degli insiemi cui faremo riferimento nel seguito.

Definizione 1.26 *Dato un insieme non vuoto $S \subseteq U$, si definisce* operazione binaria $\circ$ *su S una funzione $\circ : S \times S \mapsto U$.*

Definizione 1.27 *Un insieme non vuoto S si dice* chiuso *rispetto ad una operazione binaria $\circ$ su S se* $\mathrm{imm}(\circ) \subseteq S$.

Se un insieme S è chiuso rispetto a $\circ$, ne segue quindi che applicando tale operazione ad elementi di S si ottengono ancora elementi dell'insieme stesso.

Si consideri ora un insieme S chiuso rispetto ad un'operazione binaria $\circ$.

Definizione 1.28 *La coppia* $\langle S, \circ \rangle$ *viene detta* semigruppo *se l'operazione binaria* $\circ$ *soddisfa la seguente proprietà:*

$$\forall x \forall y \forall z \in S \ \ (x \circ (y \circ z)) = (x \circ y) \circ z) \quad \text{(associatività)}.$$

Se inoltre vale la seguente proprietà:

$$\forall x \forall y \in S \ \ (x \circ y) = (y \circ x) \quad \text{(commutatività)}$$

il semigruppo è detto commutativo.

Esempio 1.12 La coppia $\langle \mathbb{N}, + \rangle$, dove $+$ è l'usuale operazione di somma, è un semigruppo commutativo, in quanto, chiaramente, l'operazione di somma di naturali è associativa e commutativa.

Definizione 1.29 *La terna* $\langle S, \circ, e \rangle$ *viene detta* monoide *se* $\langle S, \circ \rangle$ *è un semigruppo, e se* $e \in S$ *è tale che:*

$$\forall x \in S \ \ (e \circ x) = (x \circ e) = x$$

L'elemento e *viene detto* elemento neutro *o* unità *del monoide. Se* $\circ$ *è anche commutativa, il monoide viene detto* commutativo.

Esempio 1.13 Le terne $\langle \mathbb{N}, +, 0 \rangle$ e $\langle \mathbb{N}, *, 1 \rangle$, dove $+$ e $*$ sono le usuali operazioni di somma e prodotto, sono monoidi commutativi. Infatti, oltre alle proprietà di associatività e commutatività della somma e del prodotto di naturali, si ha anche che $\forall x \in \mathbb{N} \ \ (x + 0) = (x * 1) = x$.

Definizione 1.30 *La terna* $\langle S, \circ, e \rangle$ *viene detta* gruppo *se* $\langle S, \circ, e \rangle$ *è un monoide ed inoltre l'operazione* $\circ$ *ammette inverso, cioè se*

$$\forall x \in S \ \exists y \in S \ \ (x \circ y) = (y \circ x) = e.$$

L'elemento y *viene detto* inverso *di* x*, e si denota come* x^{-1}.

Se il monoide $\langle S, \circ, e \rangle$ è commutativo il gruppo viene detto *commutativo* (o *abeliano*).

Esempio 1.14 Le terne $\langle \mathbb{N}, +, 0 \rangle$ e $\langle \mathbb{N}, *, 1 \rangle$ non sono gruppi, in quanto l'insieme $\mathbb{N}$ non è chiuso rispetto all'inverso di $+$ e di $*$. Al contrario, le terne $\langle \mathbb{Z}, +, 0 \rangle$ e $\langle \mathbb{Q}, *, 1 \rangle$ sono gruppi abeliani.

Esercizio 1.17 Dimostrare che la seguente definizione di elemento inverso

$$\forall x \in S \ \exists y \in S \ \ (x \circ y) = e,$$

è equivalente a quella data.

Osserviamo ora che, se $\circ$ è un'operazione associativa, allora $((x \circ y) \circ z) = (x \circ (y \circ z))$ e potremo non considerare l'ordine di applicazione dell'operazione, scrivendo semplicemente $x \circ y \circ z$.

Definizione 1.31 *Dati un semigruppo $\langle S, \circ\rangle$ e un insieme $X \subseteq S$, si definisce* chiusura *di X l'insieme*

$$X^T = \{x \in S \mid \exists \langle x_1, \ldots, x_n\rangle \in X^n \; x_1 \circ x_2 \circ \cdots \circ x_n = x, \; n \geq 1\}.$$

Vale a dire che X^T è l'insieme degli elementi di S che possono essere ottenuti, applicando l'operazione $\circ$ un numero arbitrario di volte, a partire dagli elementi in X. Un importante caso particolare è quando $X^T = S$.

Definizione 1.32 *Dati un semigruppo $\langle S, \circ\rangle$ e un insieme $X \subseteq S$, gli elementi di X si dicono* generatori *di S se $X^T = S$.*

Definizione 1.33 *Dati un insieme S ed una operazione associativa $\circ$, definiamo* semigruppo libero *sulla coppia $\langle S, \circ\rangle$ il semigruppo $\langle S^+, \circ^+\rangle$, dove:*

1. *S^+ è l'insieme di tutte le espressioni $x = x_1 \circ x_2 \circ \ldots \circ x_n$, per ogni $n \geq 1$, con $x_1, \ldots, x_n \in S$;*
2. *l'operazione $\circ^+$ è definita nel modo seguente: se $x = x_1 \circ \ldots \circ x_n$ e $y = y_1 \circ \ldots \circ y_n$, allora $x \circ^+ y = x_1 \circ \ldots \circ x_n \circ y_1 \circ \ldots \circ y_n$.*

Per semplicità, gli elementi di S^+ vengono usualmente denotati senza indicare il simbolo d'operazione $\circ$ (pertanto, anziché $x_1 \circ x_2 \circ x_3$ scriviamo ad esempio $x_1x_2x_3$).

Se estendiamo S^+ introducendo un elemento aggiuntivo ε, detto *parola vuota*, possiamo definire sull'insieme risultante $S^* = S^+ \cup \{\varepsilon\}$ l'operazione $\circ^*$, estensione di $\circ^+$, tale che, $\forall x, y \in S^+ \; x \circ^* y = x \circ^+ y$ e $\forall x \in S^* \; (\varepsilon \circ^* x = x \circ^* \varepsilon = x)$.

La terna $\langle S^*, \circ^*, \varepsilon\rangle$ è allora un monoide e viene detto *monoide libero*.

Esercizio 1.18 Data una sequenza di tre caratteri, si considerino le operazioni di rotazione verso destra (d), di rotazione verso sinistra (s) e l'operazione neutra (i) che non opera alcuna rotazione. Dimostrare che la coppia $\langle \{d, s, i\}, \cdot \rangle$,, dove l'operazione $\cdot$ è la composizione delle rotazioni, è un gruppo commutativo. Dimostrare che d è un generatore del gruppo. Definire il monoide libero associato alla coppia $\langle \{d\}, \cdot \rangle$.

Se consideriamo coppie di operazioni definite su S possiamo individuare le seguenti strutture algebriche più complesse.

Definizione 1.34 *La quintupla $\langle S, +, *, 0, 1\rangle$ è detta* semianello *se valgono le seguenti proprietà:*

- *$\langle S, +, 0\rangle$ è un monoide commutativo;*
- *$\langle S, *, 1\rangle$ è un monoide;*
- *l'operazione $*$ gode della proprietà distributiva rispetto all'operazione $+$, cioè $\forall x, y, z \in S \;\; x * (y + z) = (x * y) + (x * z)$;*

- *lo* 0 *è un elemento* assorbente *per il prodotto, cioè* $\forall x \in S \quad (x * 0) = (0 * x) = 0$.

Il semianello è detto commutativo *se l'operazione* $*$ *è anch'essa commutativa.*

Esempio 1.15 Esempi di semianello sono le strutture algebriche seguenti:

- $\langle \mathcal{P}(S), \cup, \cap, \emptyset, S \rangle$, per ogni insieme S,
- $\langle \mathbb{Q}, +, *, 0, 1 \rangle$,
- $\langle \{\text{FALSE}, \text{TRUE}\}, \vee, \wedge, \text{FALSE}, \text{TRUE} \rangle$ (detto *semianello booleano*).

In tutti i casi si tratta di semianelli commutativi perché le operazioni di prodotto sono commutative.

Esercizio 1.19 Dimostrare che la quintupla $\langle \mathbb{N} \cup \{\infty\}, \min, +, \infty, 0 \rangle$ è un semianello commutativo (si assuma che $\forall x \in \mathbb{N} \quad x + \infty = \infty$ e $\min(x, \infty) = x$).

Tra le strutture algebriche, le algebre di Boole rivestono un interesse particolare in informatica.

Definizione 1.35 *Un'*algebra di Boole *è un semianello* $\langle B, +, *, 0, 1 \rangle$ *con le seguenti proprietà aggiuntive:*

- $0 \neq 1$*;*
- *per ogni* $x \in B$ *esiste* $x^{-1} \in B$ *(inverso) tale che* $x * x^{-1} = 0$ *e* $x + x^{-1} = 1$*;*
- *per ogni* $x \in B$, $x * x = x$.

1.2.2 Congruenze, monoide quoziente e gruppo quoziente

Introduciamo ora alcuni importanti concetti riguardanti relazioni tra gli elementi di strutture algebriche.

Definizione 1.36 *Dato un semigruppo* $\langle S, \circ \rangle$*, una* congruenza $\equiv$ *è una relazione d'equivalenza su* S *che soddisfa la seguente proprietà:*

$$\forall x, y \in S \quad x \equiv y \Longleftrightarrow \forall z \in S \ ((x \circ z \equiv y \circ z) \wedge (z \circ x \equiv z \circ y)).$$

Esempio 1.16 Nell'Esempio 1.7 è stata definita la relazione d'equivalenza $\equiv_k$ delle classi resto rispetto alla divisione per k. È facile osservare che tale relazione è una congruenza rispetto al semigruppo commutativo $\langle \mathbb{N}, + \rangle$: infatti, se $n \equiv_k m$, abbiamo che $\forall l \ (n + l \equiv_k m + l)$ e, chiaramente, anche $l + n \equiv_k l + m$. Viceversa, se $\forall l \ (n + l \equiv_k m + l)$ allora abbiamo, nel caso particolare $l = 0$, $n \equiv_k m$.

Si noti che nel caso di semigruppi non commutativi possiamo definire anche congruenze destre (sinistre), cioè relazioni tali che

$$x \equiv y \Longleftrightarrow \forall z \in S \ (x \circ z \equiv y \circ z) \quad (x \equiv y \Longleftrightarrow \forall z \in S \ (z \circ x = z \circ y)).$$

Esempio 1.17 Dato un semigruppo $\langle S, \circ \rangle$, una relazione d'equivalenza definita nel modo seguente,

$$xRy \Longleftrightarrow \forall z \ (x \circ z = y \circ z)$$

è una relazione di congruenza destra. Infatti essa è chiaramente una relazione d'equivalenza, ed inoltre, poiché $xRy \Longleftrightarrow x \circ z = y \circ z$ per ogni z, abbiamo che $xRy \Rightarrow \forall z \ \ (x \circ z)R(y \circ z)$. D'altra parte, se $\forall z \ \ (x \circ z)R(y \circ z)$ allora $\forall z \forall z' \ (x \circ z \circ z' = y \circ z \circ z')$ e quindi, assumendo $u = z \circ z'$ abbiamo $\forall u \ (x \circ u = y \circ u)$ che per definizione implica xRy.

Dati un semigruppo $\langle S, \circ \rangle$ e una congruenza C su S, denotiamo con S mod C o S/C l'insieme delle classi d'equivalenza della congruenza. Sia $[x]$ la classe d'equivalenza dell'elemento $x \in S$: se definiamo $[x] \circ [y] = [x \circ y]$, la struttura $\langle S/C, \circ \rangle$ è un semigruppo, ed è chiamato *semigruppo quoziente*. Inoltre, dato il monoide $\langle S, \circ, 1 \rangle$, la struttura $\langle S/C, \circ, [1] \rangle$ è chiamata *monoide quoziente*. Dato il gruppo $\langle S, \circ, 1 \rangle$, se definiamo $[x]^{-1} = [x^{-1}]$, abbiamo che la struttura $\langle S/C, \circ, 1 \rangle$ è un gruppo, detto *gruppo quoziente*. Il numero di elementi di una struttura algebrica (semigruppo, monoide, gruppo) ottenuta come struttura "quoziente", viene chiamato *indice* della struttura.

Esempio 1.18 Consideriamo ad esempio il semigruppo $\langle \mathbb{Z}, + \rangle$ e la congruenza $\equiv_k$ (con $k \geq 2$). Allora, per ogni $x \in \mathbb{Z}$, la classe di equivalenza $[x]$ è data dall'insieme $\{y \in \mathbb{Z} \mid \exists i \in \mathbb{Z} \ \ (x - y) = ik\}$ degli interi che differiscono da x per un multiplo di k. Il relativo semigruppo quoziente (in effetti un gruppo quoziente) è costituito dall'insieme $\mathbb{Z}_k$ degli interi positivi minori o uguali a k e dall'operazione $+$ definita *modulo* k, come $\forall x, y, z \in \mathbb{Z}_k \ \ (x + y) = z \Longleftrightarrow x + y = z - ik, \ k \in \mathbb{Z}$.

Esercizio 1.20 Si consideri il gruppo dei razionali $\langle \mathbb{Q}, +, 0 \rangle$. Dimostrare che la relazione $aRb \Longleftrightarrow a - b \in \mathbb{Z}$ è una congruenza. Definire inoltre il gruppo quoziente e mostrare che il suo indice non è finito.

1.3 Caratteristiche elementari dei linguaggi

Fra le strutture matematiche utilizzate nell'ambito dell'informatica, i linguaggi formali svolgono un ruolo particolarmente importante. In questa sezione introduciamo alcuni concetti relativi ai linguaggi ed alle principali operazioni che nel seguito avremo occasione di utilizzare.

1.3.1 Alfabeto, stringa, linguaggio

Definiamo innanzitutto i concetti più elementari di alfabeto e stringa.

Definizione 1.37 *Un insieme finito non vuoto* Σ *di simboli (detti* caratteri*) prende il nome di* alfabeto.

Esempio 1.19 Ad esempio $\{0,1\}$ è l'alfabeto binario, mentre

$$\{A,\ldots,Z,a,\ldots,z,+,-,*,/,<,>,.,:,;,=,',\uparrow,),(,],[,0,1,\ldots,9\}$$

oltre ai simboli 'spazio', 'virgola', 'parentesi graffa aperta' e 'parentesi graffa chiusa', costituisce l'alfabeto utilizzato nel linguaggio di programmazione Pascal.

Se consideriamo i caratteri di un alfabeto Σ come i generatori di un semigruppo, possiamo fare riferimento al monoide libero corrispondente (vedi Definizione 1.33).

Definizione 1.38 *Dato un alfabeto Σ, denotiamo come $\langle \Sigma^*, \circ, \varepsilon\rangle$ il* monoide libero definito su Σ. *Tale monoide è chiamato anche* monoide sintattico. *Gli elementi di Σ^* vengono detti* parole *o* stringhe. *L'elemento ε viene detto* parola vuota. *L'operazione $\circ : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$ definita sul monoide è chiamata* concatenazione *e consiste nel giustapporre due parole di Σ^*:*

$$x_{i_1}\ldots x_{i_n} \circ y_{j_1}\ldots y_{j_m} = x_{i_1}\ldots x_{i_n} y_{j_1}\ldots y_{j_m},$$

con $x_{i_1},\ldots,x_{i_n},y_{j_1},\ldots,y_{j_m} \in \Sigma$.

Dato un alfabeto Σ ed il monoide sintattico definito su di esso vale naturalmente la proprietà:

$$\forall x \;\; x \circ \varepsilon = \varepsilon \circ x = x.$$

La concatenazione di due stringhe x e y è frequentemente indicata omettendo il simbolo $\circ$, cioè scrivendo xy anziché $x \circ y$.

Con la notazione $|\, x \,|$ indichiamo la *lunghezza* di una parola x, ovvero il numero di caratteri che la costituiscono. Chiaramente $|\, \varepsilon \,| = 0$. Si osservi inoltre che la concatenazione non gode della proprietà commutativa e quindi in generale:

$$x \circ y \neq y \circ x.$$

Un caso particolare di concatenazione è quello in cui la stringa viene concatenata con sé stessa: con x^h si denota la concatenazione di x con sé stessa iterata h volte. Per convenzione con x^0 si intende la stringa vuota.

Normalmente si rappresentano per convenzione i caratteri di Σ con le prime lettere minuscole dell'alfabeto latino e le parole di Σ^* con le ultime lettere, sempre minuscole, dell'alfabeto latino.

Definizione 1.39 *Dato un alfabeto Σ, si definisce* linguaggio *un qualsivoglia sottoinsieme di Σ^*.*

Si noti che poiché $\Sigma \subseteq \Sigma^*$, un alfabeto è a sua volta un linguaggio.

Definizione 1.40 *Si chiama* linguaggio vuoto, *e lo si indica con Λ, il linguaggio che non contiene stringa alcuna.*

Si noti che $\Lambda \neq \{\varepsilon\}$.

Esempio 1.20 Dato l'alfabeto $\{a, b\}$, l'insieme $\{a^n b^n \mid n \geq 1\}$ è il linguaggio composto da tutte le stringhe costituite dalla concatenazione di un certo numero di a, seguita dalla concatenazione dello stesso numero di b.

Data una stringa x, risulterà interessante considerare la stringa ottenuta invertendo l'ordine dei caratteri di x. Più formalmente, abbiamo la seguente definizione.

Definizione 1.41 *Data una stringa x, chiamiamo* inversa *(o* riflessa*) di x la stringa $\tilde{x}$ definita nel seguente modo:*

$$\tilde{x} = \begin{cases} x & se\ x = \varepsilon\ o\ x = a \\ a\tilde{y} & se\ x = ya \end{cases}$$

per ogni $a \in \Sigma$.

Infine, dato un alfabeto Σ, definiamo l'insieme delle stringhe palindrome su Σ come $\{x \mid x = \tilde{x}\}$.

1.3.2 Operazioni su linguaggi

Dati due linguaggi L_1 ed L_2 si possono definire su essi varie operazioni: in particolare introdurremo in questa sezione le operazioni binarie di intersezione, unione e concatenazione (o prodotto), nonché quelle unarie di complementazione ed iterazione. Intersezione, unione e complementazione sono normali operazioni su insiemi, mentre concatenazione ed iterazione sono definite sulla base della struttura algebrica del monoide $\langle \Sigma^*, \circ, \varepsilon \rangle$.

Definizione 1.42 *L'*intersezione *di due linguaggi L_1 e L_2 è il linguaggio $L_1 \cap L_2$ costituito dalle parole di L_1 e di L_2, cioè $L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$.*

Definizione 1.43 *L'*unione *di due linguaggi L_1 e L_2 è il linguaggio $L_1 \cup L_2$ costituito dalle parole appartenenti ad almeno uno fra L_1 ed L_2, cioè $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$.*

Si noti che$L_1 \cap \Lambda = \Lambda$ e $L_1 \cup \Lambda = L_1$.

Definizione 1.44 *Il* complemento *di un linguaggio L_1 è il linguaggio $\overline{L_1} = \Sigma^* - L_1$ costituito dalle parole appartenenti a Σ^* ma non ad L_1, cioè $\overline{L_1} = \{x \in \Sigma^* \mid x \notin L_1\}$.*

Definizione 1.45 *La* concatenazione *(o* prodotto*) di due linguaggi L_1 e L_2 è il linguaggio $L_1 \circ L_2$[7] delle parole costituite dalla concatenazione di una stringa di L_1 e di una stringa di L_2, cioè*

$$L_1 \circ L_2 = \{x \in \Sigma^* \mid \exists y_1 \in L_1\ \exists y_2 \in L_2\ (x = y_1 \circ y_2)\}.$$

[7] Si osservi che, anche se per indicare la concatenazione tra stringhe e tra linguaggi viene adottato lo stesso simbolo, le due operazioni sono comunque diverse e, peraltro, il contesto consente di evitare ogni ambiguità.

Si noti che $L \circ \{\varepsilon\} = \{\varepsilon\} \circ L = L$, e che $L \circ \Lambda = \Lambda \circ L = \Lambda$.

Esempio 1.21 Dati $L_1 = \{a^n \mid n \geq 1\}$ e $L_2 = \{b^m \mid m \geq 1\}$,

$$L_1 \circ L_2 = \{a^n b^m \mid n, m \geq 1\}.$$

Esempio 1.22 Dati $L_1 = \{\text{ASTRO}, \text{FISIO}\}$ ed $L_2 = \{\text{LOGIA}, \text{NOMIA}\}$, abbiamo che $L_1 \circ L_2$ è costituito dalle stringhe ASTROLOGIA, ASTRONOMIA, FISIOLOGIA e FISIONOMIA.

Esercizio 1.21 Dimostrare che la quintupla

$$\langle \mathcal{P}(\Sigma^*), \cup, \circ, \Lambda, \{\varepsilon\} \rangle$$

è un semianello non commutativo.

Il concetto di potenza inteso come iterazione della concatenazione viene esteso ai linguaggi mediante la seguente definizione.

Definizione 1.46 *La* potenza L^h *di un linguaggio è definita come*

$$L^h = L \circ L^{h-1}, h \geq 1$$

con la convenzione secondo cui $L^0 = \{\varepsilon\}$.

Si noti che, in base alla suddetta convenzione, $\Lambda^0 = \{\varepsilon\}$. Con questa definizione, l'insieme delle stringhe di lunghezza h sull'alfabeto Σ si può indicare con Σ^h.

Definizione 1.47 *Il linguaggio* L^* *definito da*

$$L^* = \bigcup_{h=0}^{\infty} L^h$$

prende il nome di chiusura riflessiva del linguaggio L *rispetto all'operazione di concatenazione, mentre l'operatore "*$*$*" prende il nome di* iterazione *o* stella di Kleene.

Si noti che, dato un qualunque linguaggio L, $\varepsilon \in L^*$, e che $\Lambda^* = \{\varepsilon\}$.

Esempio 1.23 Se $L = \{aa\}$, allora $L^* = \{a^{2n} \mid n \geq 0\}$.

Esempio 1.24 Dati i linguaggi $L_1 = \{\text{BIS}\}$ ed $L_2 = \{\text{NONNO}\}$, il linguaggio $L_1^* \circ L_2$ contiene le stringhe NONNO, BISNONNO, BISBISNONNO, ...

Definizione 1.48 *Si indica con* L^+ *la* chiusura (non riflessiva) *definita da*

$$L^+ = \bigcup_{h=1}^{\infty} L^h$$

Risulta ovviamente $L^* = L^+ \cup \{\varepsilon\}$.

Esempio 1.25 Se $L = \{aa\}$, allora $L^+ = \{a^{2n} \mid n \geq 1\}$.

Si noti che le Definizioni 1.47 e 1.48 sono coerenti con quanto specificato nella Definizione 1.38. In particolare si noti che Σ^* può essere ottenuto come chiusura riflessiva del linguaggio Σ.

1.3.3 Espressioni regolari

Le considerazioni finora svolte relativamente ai linguaggi sono state effettuate senza fornire alcuno strumento specifico di descrizione dei linguaggi stessi. Nel seguito introdurremo vari formalismi per la rappresentazione di linguaggi: nella presente sezione introduciamo un primo strumento, di natura algebrica, che verrà frequentemente utilizzato a tale scopo. Tale strumento è costituito dalle cosiddette *espressioni regolari* e consente di descrivere tutti i linguaggi appartenenti a un'importante classe.

Definizione 1.49 *Dato un alfabeto* Σ *e dato l'insieme di simboli*

$$\{+, ^*, (,), \cdot, \emptyset\}$$

si definisce espressione regolare *sull'alfabeto* Σ *una stringa*

$$r \in (\Sigma \cup \{+, *, (,), ., \emptyset\})^+$$

tale che valga una delle seguenti condizioni:

1. $r = \emptyset$
2. $r \in \Sigma$
3. $r = (s + t)$, *oppure* $r = (s \cdot t)$, *oppure* $r = s^*$, *dove* s *e* t *sono espressioni regolari sull'alfabeto* Σ.

Come si è detto, le espressioni regolari consentono di rappresentare linguaggi mediante una opportuna interpretazione dei simboli che le compongono. Nella Tabella 1.3 si mostra la corrispondenza tra un'espressione regolare e il linguaggio che essa rappresenta. Per convenzione, se s e t sono due espressioni

Espr. regolari	Linguaggi
$\emptyset$	Λ
a	$\{a\}$
$(s + t)$	$\mathcal{L}(s) \cup \mathcal{L}(t)$
$(s \cdot t)$	$\mathcal{L}(s) \circ \mathcal{L}(t)$
s^*	$(\mathcal{L}(s))^*$

Tabella 1.3 Espressioni regolari e corrispondenti linguaggi. $\mathcal{L}(r)$ denota il linguaggio rappresentato dall'espressione regolare r.

regolari, al posto di $(s \cdot t)$ si può scrivere (st). Inoltre, assumendo che il simbolo $*$ abbia precedenza sul simbolo $\cdot$ e questo abbia precedenza sul simbolo $+$, e tenendo conto dell'associatività di queste operazioni, si possono spesso eliminare alcune parentesi.

Esempio 1.26 L'espressione regolare $\Big(a + \big(b \cdot (c \cdot d)\big)\Big)$ definita sull'alfabeto $\Sigma = \{a, b, c, d\}$ può essere sostituita da $a + bcd$.

Va detto che anche per le espressioni regolari si può introdurre la potenza, scrivendo ad esempio $(r)^3$ per indicare l'espressione regolare rrr e la chiusura (non riflessiva) scrivendo $(r)^+$ per indicare $r(r)^*$, dove r è una qualunque espressione regolare.[8] Si noti che per rappresentare la stringa vuota può essere utile a volte usare il simbolo ε nelle espressioni regolari, con lo scopo di indicare il linguaggio $\{\varepsilon\}$. Tuttavia ciò non è formalmente necessario in quanto, per le proprietà viste, abbiamo che il linguaggio rappresentato da $\emptyset^*$ è esattamente $\{\varepsilon\}$.

Esempio 1.27 L'espressione regolare $(a + b)^* a$ rappresenta il linguaggio

$$\begin{aligned}
\mathcal{L}\left((a+b)^*a\right) &= \mathcal{L}\left(\left(a+b\right)^*\right) \circ \mathcal{L}\left(a\right) \\
&= \left(\mathcal{L}\left(a+b\right)\right)^* \circ \mathcal{L}\left(a\right) \\
&= \left(\mathcal{L}\left(a\right) \cup \mathcal{L}\left(b\right)\right)^* \circ \{a\} \\
&= \left(\{a\} \cup \{b\}\right)^* \circ \{a\} \\
&= \{a, b\}^* \circ \{a\} \\
&= \{x \mid x \in \{a, b\}^+, x \text{ termina con } a\}.
\end{aligned}$$

Esempio 1.28 Posto $c = \{0, 1, \ldots, 9\}$, l'espressione regolare che rappresenta i numeri reali nella forma $c_{i1}c_{i2}\ldots c_{in}.c_{f1}c_{f2}\ldots c_{fm}$ (dove c_{ij} rappresenta la j-esima cifra prima del punto decimale, e c_{fk} la k-esima cifra dopo il punto decimale) è

$$\big((1 + 2 + \ldots + 9)(0 + 1 + \ldots + 9)^* + 0\big).(0 + 1 + \ldots + 9)^+.$$

Nei capitoli seguenti si vedrà che i linguaggi rappresentabili mediante espressioni regolari appartengono ad una classe molto particolare, detta appunto classe dei *linguaggi regolari*, che gode di molte proprietà notevoli, fra cui la chiusura rispetto alle cinque operazioni definite all'inizio della Sezione 1.3.2.

Esercizio 1.22 Determinare l'espressione regolare che, sull'alfabeto $\{a, b\}$, definisce l'insieme delle stringhe il cui terzultimo carattere è una b.

Esercizio 1.23 Determinare il linguaggio definito dall'espressione regolare $a^*\big((aa)^*b + (bb)^*a\big)b^*$.

Esercizio 1.24 Sia data la mappa stradale (con tratti stradali a senso unico e contrassegnati da caratteri dell'alfabeto) schematicamente indicata in Figura 1.6.

Fornire un'espressione regolare che definisca tutti i percorsi, anche passanti più volte per uno stesso nodo, tra A e B.

[8] Tuttavia tali arricchimenti non accrescono, come è evidente, la potenza descrittiva delle espressioni regolari, ma solo la loro sinteticità.

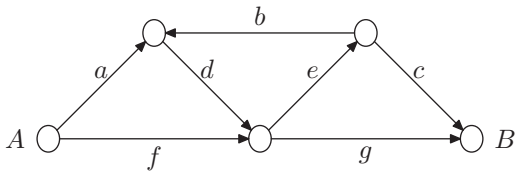


FIGURA 1.6 Mappa stradale (relativa all'Esercizio 1.24).

1.3.4 Cardinalità dei linguaggi

Tra i problemi più basilari che debbono essere affrontati nell'informatica vi è quello di riconoscere se una stringa appartenga o meno ad un determinato linguaggio, opportunamente definito. Ad esempio, quando un compilatore analizza un programma in linguaggio C o Pascal deve verificare che tale programma sia sintatticamente corretto, o, in altri termini, che la stringa costituita dal programma stesso appartenga al linguaggio dei programmi C o Pascal sintatticamente corretti. Ebbene, il problema di riconoscere l'appartenenza di una stringa ad un linguaggio non può essere sempre risolto mediante un algoritmo. Infatti, i concetti definiti precedentemente ci permettono di valutare la cardinalità dell'insieme di tutti i linguaggi definiti su un dato alfabeto, e di osservare, in modo intuitivo, l'esistenza di linguaggi per i quali non esiste alcun algoritmo di riconoscimento.

Definizione 1.50 *Dato un alfabeto* $\Sigma = \{a_1, \ldots, a_n\}$*, si definisce* ordinamento lessicografico *delle stringhe di* Σ^* *l'ordinamento* $<$ *ottenuto stabilendo un (qualunque) ordinamento tra i caratteri di* Σ *(conveniamo, senza perdita alcuna di generalità, di modificare la numerazione dei caratteri in modo da assicurare che* $a_1 < a_2 < \ldots < a_n$*) e definendo l'ordinamento di due stringhe* $x, y \in \Sigma^*$ *in modo tale che* $x < y$ *se e solo se una delle condizioni seguenti è verificata*

1. $|x| < |y|$;
2. $|x| = |y|$ *ed esiste* $z \in \Sigma^*$ *tale che* $x = za_iu$ *e* $y = za_jv$*, con* $u, v \in \Sigma^*$ *e* $i < j$.

Esempio 1.29 Dato l'alfabeto $\Sigma = \{a, b\}$, dove assumiamo $a < b$, le stringhe di Σ^* sono enumerate in modo lessicografico come segue: ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, ...

L'esistenza dell'ordinamento lessicografico per le stringhe di Σ^* conduce direttamente ad una enumerazione, il che ci consente di affermare che la cardinalità di Σ^* è $\aleph_0$ e, in base a quanto visto in precedenza, che l'insieme di tutti i linguaggi su Σ, essendo equinumeroso a $\mathcal{P}(\Sigma^*)$, ha cardinalità $2^{\aleph_0}$, e non è quindi numerabile.

Consideriamo ora la cardinalità dell'insieme dei programmi che possono essere scritti utilizzando un qualunque linguaggio di programmazione, ad esempio il Pascal. Se Σ_P è l'alfabeto del linguaggio Pascal, i programmi sono stringhe di Σ_P^* che sono accettate (senza segnalazione di errori sintattici) da un compilatore Pascal. Naturalmente la correttezza sintattica non implica la correttezza di comportamento del programma rispetto ai calcoli che vogliamo eseguire, ma ciò, per il momento, non ci interessa.

Utilizzando ancora l'ordinamento lessicografico, osserviamo che i programmi Pascal possono essere enumerati nel seguente modo: vengono generate via via le stringhe di Σ_P^* in ordine lessicografico, ogni stringa viene sottoposta al compilatore, il quale ne verifica la correttezza sintattica, e la stringa entra a far parte dell'enumerazione se e solo se tale verifica dà esito positivo. È chiaro dunque che anche l'insieme dei programmi Pascal ha cardinalità (al più) $\aleph_0$.

Poniamoci ora il seguente problema. È possibile che, dato un qualunque $L \subseteq \Sigma^*$, esista un programma Pascal che, data in input una qualunque stringa $x \in \Sigma^*$, ne decida l'appartenenza a L, vale a dire restituisca il valore TRUE se la stringa appartiene ad L e restituisca il valore FALSE altrimenti? Una semplice considerazione di cardinalità ci consente di osservare immediatamente che i programmi Pascal sono una quantità contabile e non possono quindi essere messi in corrispondenza con i linguaggi, che hanno la cardinalità del continuo. In altre parole, esistono più linguaggi *da riconoscere* che programmi che *riconoscono*. Devono perciò necessariamente esistere linguaggi per i quali non esiste alcun programma Pascal di riconoscimento.

Questo risultato, ottenuto in modo informale, ed ovviamente indipendente dal linguaggio di programmazione preso ad esempio, sarà successivamente confermato in modo più preciso quando dimostreremo che qualunque formalismo per il calcolo di funzioni o per il riconoscimento d'insiemi ha un potere computazionale limitato ed esistono quindi funzioni che esso non può calcolare ed insiemi che esso non può riconoscere.

1.4 Notazione asintotica

Un'ultima serie di concetti che desideriamo richiamare prima di passare alla trattazione degli argomenti principali di questo volume riguarda la notazione che esprime l'andamento asintotico di una funzione rispetto ad un'altra.

Definizione 1.51 *Data una funzione* $f : \mathbb{N} \mapsto \mathbb{N}$*, la notazione* $O(f(n))$ *denota l'insieme delle funzioni:*

$$\{g \mid (\exists c > 0)\ (\exists n_0 > 0)\ (\forall n \geq n_0)\ (g(n) \leq cf(n))\}\,.$$

Se una funzione $g(n)$ è tale che $g(n) \in O(f(n))$ si ha quindi che asintoticamente, per valori sufficientemente grandi, la funzione assume valori non superiori a quelli assunti dalla funzione $f(n)$, a meno di una costante moltiplicativa prefissata. In tal caso diciamo che $g(n)$ cresce *al più* come $f(n)$. Con un abuso di notazione scriveremo anche $g(n) = O(f(n))$.

Definizione 1.52 *Data una funzione $f : \mathbb{N} \mapsto \mathbb{N}$, la notazione $\Omega(f(n))$ denota l'insieme delle funzioni:*

$$\{g \mid (\exists c > 0)\ (\exists n_0 > 0)\ (\forall n \geq n_0)\ (g(n) \geq cf(n))\}\,.$$

Se $g(n) \in \Omega(f(n))$ si dice che $g(n)$ cresce *almeno* come $f(n)$. Con un abuso di notazione si scrive anche $g(n) = \Omega(f(n))$.

Definizione 1.53 *Data una funzione $f : \mathbb{N} \mapsto \mathbb{N}$, la notazione $\Theta(f(n))$ denota l'insieme delle funzioni:*

$$\{g \mid (\exists c_1 > 0)\ (\exists c_2 \geq c_1)\ (\exists n_0 > 0)\ (\forall n \geq n_0)(c_1 f(n) \leq g(n) \leq c_2 f(n))\}\,.$$

Se $g(n) \in \Theta(f(n))$ si dice che $g(n)$ e $f(n)$ crescono nello stesso modo. Con un abuso di notazione si scrive anche $g(n) = \Theta(f(n))$.

Definizione 1.54 *Date due funzione $f, g : \mathbb{N} \mapsto \mathbb{N}$, con la notazione $g(n) = o(f(n))$ indichiamo che:*

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

Se $g(n) = o(f(n))$ diciamo che $g(n)$ è un infinitesimo rispetto a $f(n)$ o, equivalentemente, che $f(n)$ è un infinito rispetto a $g(n)$. La stessa relazione può essere espressa mediante la notazione $f(n) = \omega(g(n))$.

Valgono ovviamente le seguenti proprietà:

$$f(n) = O(g(n)) \Longleftrightarrow g(n) = \Omega(f(n))$$

$$f(n) = \Theta(g(n)) \Longleftrightarrow g(n) = \Theta(f(n))$$

$$f(n) = \Theta(g(n)) \Longleftrightarrow \Big(f(n) = O((g(n))\Big) \wedge \Big(f(n) = \Omega(g(n))\Big).$$

Come vedremo, la notazione asintotica viene utilizzata in particolare per rappresentare il costo di esecuzione degli algoritmi e la complessità dei problemi. Nel seguito utilizzeremo tale notazione ancor prima di definire formalmente il concetto di complessità computazionale, per indicare il numero di passi eseguiti da un algoritmo ed esprimere quindi, anche informalmente, il suo costo di esecuzione. Nel dire, ad esempio, che un algoritmo ha un costo $O(n^2)$ per una stringa in input di n caratteri, intenderemo dire che esiste una opportuna costante c tale che, per ogni n sufficientemente grande, il numero di

passi eseguiti dall'algoritmo su ogni stringa di lunghezza n è minore o eguale a cn^2.

Esercizio 1.25 Esprimere con le notazioni O, Ω, Θ e o l'andamento asintotico delle seguenti funzioni:

$$\begin{aligned} f(n) &= 3n^2 + 2\log n \\ g(n) &= 2\sqrt{n} + 5\frac{n^{4/3}}{n} \end{aligned}$$

Capitolo 2

Linguaggi formali

Tra i concetti principali alla base dell'informatica, il concetto di linguaggio riveste un ruolo particolarmente importante. Anche se da un punto di vista matematico un linguaggio è semplicemente un insieme di stringhe su un dato alfabeto, nell'ambito dell'informatica siamo interessati a linguaggi, generalmente infiniti, le cui stringhe sono caratterizzate da qualche particolare proprietà: ad esempio, il linguaggio dei programmi C è costituito da tutte le stringhe che soddisfano la proprietà di poter essere analizzate da un compilatore C senza che venga rilevato alcun errore sintattico.

Lo studio formale dei linguaggi è una parte importante dell'informatica teorica e trova applicazione in varie direzioni. La prima, più evidente, è appunto lo studio delle proprietà sintattiche dei programmi, cioè lo studio dei metodi di definizione della sintassi di un linguaggio di programmazione, dei metodi per la verifica che un programma soddisfi le proprietà sintattiche volute e dei metodi di traduzione da un linguaggio ad un altro (tipicamente da un linguaggio di programmazione ad alto livello al linguaggio di macchina).

Inoltre, al di là di questa importante applicazione, la familiarità con i concetti di generazione e riconoscimento di linguaggi è importante in quanto stringhe di caratteri aventi struttura particolare vengono frequentemente utilizzate, in informatica, per rappresentare vari aspetti relativi all'elaborazione di dati, come ad esempio sequenze di operazioni eseguite da un programma, sequenze di passi elementari eseguiti da una macchina, protocolli di comunicazione, sequenze di azioni eseguite nell'interazione con un'interfaccia utente, ecc.

Infine, è importante tenere presente che, nella formulazione dei problemi trattati mediante metodi algoritmici (come il calcolo di funzioni, la risoluzione di problemi di ottimizzazione, ecc.) ci si riconduce frequentemente al problema standard di decidere l'appartenenza di una stringa ad un dato linguaggio: ad esempio, anziché porsi il problema di calcolare una funzione $f : \mathbb{N} \mapsto \mathbb{N}$, ci si

può porre il problema di riconoscere il linguaggio $\{a^n b^{f(n)} \mid n \geq 0\}$.

La definizione di linguaggi si può effettuare mediante strumenti diversi. Un primo esempio di strumento formale per la definizione di linguaggi sono le espressioni regolari, già incontrate in Sezione 1.3.3. Come vedremo in seguito, tuttavia, le espressioni regolari consentono di definire linguaggi di tipo particolare, con una struttura piuttosto semplice, e non sono idonee a rappresentare linguaggi più complessi, come ad esempio i linguaggi di programmazione.

Un approccio alternativo è il cosiddetto approccio *generativo*, dove si utilizzano opportuni strumenti formali, le *grammatiche formali* appunto, che consentono di costruire le stringhe di un linguaggio tramite un insieme prefissato di regole, dette *regole di produzione*.

Altro approccio di interesse, infine, è quello *riconoscitivo*, che consiste nell'utilizzare macchine astratte, dette *automi riconoscitori*, che definiscono algoritmi di riconoscimento dei linguaggi stessi, vale a dire algoritmi che per un dato linguaggio $L \subseteq \Sigma^*$ stabiliscono se una stringa $x \in \Sigma^*$ appartiene a L o meno.

2.1 Grammatiche di Chomsky

In generale, con il termine di *grammatica* si intende un formalismo che permette di definire un insieme di stringhe mediante l'imposizione di un particolare metodo per la loro costruzione. Tale costruzione si effettua partendo da una stringa particolare e *riscrivendo* via via parti di essa secondo una qualche regola specificata nella grammatica stessa.

Le grammatiche formali presentate in questo capitolo sono denominate *Grammatiche di Chomsky* perché furono introdotte appunto dal linguista Noam Chomsky con lo scopo di rappresentare i procedimenti sintattici elementari che sono alla base della costruzione di frasi della lingua inglese. Pur essendosi presto rivelate uno strumento inadeguato per lo studio del linguaggio naturale, le grammatiche formali hanno un ruolo fondamentale nello studio delle proprietà sintattiche dei programmi e dei linguaggi di programmazione.

Vediamo ora, in modo più preciso, cos'è una grammatica formale e come si caratterizzano le stringhe da essa generate.

Definizione 2.1 *Una* grammatica formale $\mathcal{G}$ *è una quadrupla*

$$\mathcal{G} = \langle V_T, V_N, P, S \rangle$$

in cui

1. V_T *è un insieme finito e non vuoto di simboli detto* alfabeto terminale*, i cui elementi sono detti* caratteri terminali*, o* terminali*;*

2. V_N *è un insieme finito e non vuoto di simboli, detto* alfabeto non terminale *i cui elementi sono detti* caratteri non terminali *(o* non terminali*, o* variabili*, o* categorie sintattiche*);*

3. *P è una relazione binaria di cardinalità finita su*

$$(V_T \cup V_N)^* \circ V_N \circ (V_T \cup V_N)^* \times (V_T \cup V_N)^*.$$

P è detta insieme delle regole di produzione *(o* delle produzioni*, o delle* regole sintattiche*). Una coppia $\langle \alpha, \beta \rangle \in P$, si indica generalmente con la notazione $\alpha \longrightarrow \beta$;*

4. *$S \in V_N$ è detto* assioma *ed è il simbolo non terminale di inizio, ossia la categoria sintattica più generale.*

Si noti la presenza di almeno un carattere non terminale nel membro sinistro di ogni produzione: le produzioni di una grammatica rappresentano le regole mediante le quali una stringa non composta tutta di caratteri terminali può venire trasformata (riscritta) in un'altra. Il linguaggio generato dalla grammatica è l'insieme delle stringhe costituite da soli terminali ai quali si può pervenire partendo dall'assioma e applicando una sequenza, arbitrariamente lunga, di passi di riscrittura.

Esempio 2.1 Si consideri la grammatica $\mathcal{G} = \langle \{a, b\}, \{S, B, C\}, P, S \rangle$, le cui regole di produzione sono

1. $S \longrightarrow aS$
2. $S \longrightarrow B$
3. $B \longrightarrow bB$
4. $B \longrightarrow bC$
5. $C \longrightarrow cC$
6. $C \longrightarrow c.$

Con questa grammatica si possono generare le stringhe del linguaggio

$$L(\mathcal{G}) = \{a^n b^m c^h \mid n \geq 0,\ m, h \geq 1\}.$$

Ad esempio, per generare la stringa *aabccc* si deve applicare prima due volte la regola 1, poi la regola 2, poi la regola 4, poi due volte la regola 5 ed infine la regola 6. Si noti che per generare la stringa non è stata mai usata la regola 3.

Normalmente un insieme di produzioni aventi stessa parte sinistra, del tipo cioè

$$\begin{aligned} \alpha &\longrightarrow \beta_1 \\ \alpha &\longrightarrow \beta_2 \\ &\dots \\ \alpha &\longrightarrow \beta_n, \end{aligned}$$

viene convenzionalmente indicato, in maniera più compatta, con la notazione

$$\alpha \longrightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n.$$

Inoltre, l'unione $V_T \cup V_N$ viene indicata con V. Ancora per convenzione, si usano le maiuscole dell'alfabeto latino per denotare i caratteri di V_N, le minuscole iniziali dell'alfabeto latino per denotare i caratteri di V_T, le minuscole finali per denotare stringhe di V_T^* e le minuscole dell'alfabeto greco per indicare stringhe di V^*.

Definizione 2.2 *Una regola del tipo $\alpha \longrightarrow \varepsilon$, dove $\alpha \in V^* \circ V_N \circ V^*$, prende il nome di* ε-produzione *o* ε-regola.

La derivazione è il passo formale di base che consente di generare stringhe utilizzando una grammatica.

Definizione 2.3 *Sia data una grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$. La* derivazione diretta *(rispetto a $\mathcal{G}$) è una relazione su $(V^* \circ V_N \circ V^*) \times V^*$, rappresentata con il simbolo $\underset{\mathcal{G}}{\Longrightarrow}$ e così definita: la coppia $\langle \phi, \psi\rangle$ appartiene alla relazione, e scriviamo $\phi \underset{\mathcal{G}}{\Longrightarrow} \psi$ (ψ deriva direttamente da ϕ tramite $\mathcal{G}$) se esistono $\alpha \in V^* \circ V_N \circ V^*$ e $\beta, \gamma, \delta \in V^*$ tali che $\phi = \gamma\alpha\delta,\ \psi = \gamma\beta\delta$ e $\alpha \longrightarrow \beta \in P$.*

Definizione 2.4 *Data una grammatica $\mathcal{G}$, una* derivazione *(in $\mathcal{G}$) è una sequenza di stringhe $\phi_1, \ldots, \phi_n \in V^*$ tali che $\forall i \in \{1, \ldots, n-1\}$ $\phi_i \underset{\mathcal{G}}{\Longrightarrow} \phi_{i+1}$.*

La relazione di *derivabilità* (rispetto a $\mathcal{G}$) è la chiusura transitiva e riflessiva della derivazione diretta: essa si rappresenta con la notazione $\overset{*}{\underset{\mathcal{G}}{\Longrightarrow}}$. Si osservi che scrivendo $\phi \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} \psi$ esprimiamo l'esistenza di (almeno) una derivazione da ϕ a ψ.

Definizione 2.5 *Data una grammatica $\mathcal{G}$, si definisce* forma di frase *(in $\mathcal{G}$) una qualunque stringa $\phi \in V^*$ tale che $S \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} \phi$.*

Definizione 2.6 *Il* linguaggio generato da una grammatica $\mathcal{G}$ *è l'insieme*

$$L(\mathcal{G}) = \left\{ x \mid x \in V_T^* \wedge S \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x \right\}.$$

Il linguaggio generato da una grammatica formale è dunque un insieme di stringhe di caratteri terminali, ognuna delle quali si può ottenere a partire dall'assioma mediante l'applicazione di un numero finito di passi di derivazione diretta. Equivalentemente, possiamo definire il linguaggio generato da una grammatica come l'insieme di tutte e sole le forme di frase composte da soli simboli terminali.

Quando risulterà chiaro dal contesto a quale grammatica ci riferiamo, anziché $\underset{\mathcal{G}}{\Longrightarrow}$ o $\overset{*}{\underset{\mathcal{G}}{\Longrightarrow}}$ scriveremo semplicemente $\Longrightarrow$ o $\overset{*}{\Longrightarrow}$.

È bene notare che talvolta si scrive $\alpha \stackrel{i}{\Longrightarrow} \beta$ per indicare che la stringa β è ottenibile da α mediante l'applicazione di i (o meno) passi di produzione diretta.

Esempio 2.2 Si supponga di voler generare il linguaggio

$$L = \{a^n b^n c^n \mid n \geq 1\}.$$

A tal fine si può utilizzare una grammatica $\mathcal{G}$ in cui $V_T = \{a, b, c\}$ e $V_N = \{S, B, C, F, G\}$ e le regole di P sono le seguenti:

$$\begin{array}{rcl} S & \longrightarrow & aSBC \\ CB & \longrightarrow & BC \\ SB & \longrightarrow & bF \\ FB & \longrightarrow & bF \\ FC & \longrightarrow & cG \\ GC & \longrightarrow & cG \\ G & \longrightarrow & \varepsilon. \end{array}$$

Per generare la stringa *aabbcc* si può procedere come segue:

$$\begin{array}{rcl} S & \Longrightarrow & aSBC \\ & \Longrightarrow & aaSBCBC \\ & \Longrightarrow & aaSBBCC \\ & \Longrightarrow & aabFBCC \\ & \Longrightarrow & aabbFCC \\ & \Longrightarrow & aabbcGC \\ & \Longrightarrow & aabbccG \\ & \Longrightarrow & aabbcc. \end{array}$$

Ciò mostra che $S \stackrel{*}{\Longrightarrow} aabbcc$, ed in particolare che $S \stackrel{i}{\Longrightarrow} aabbcc$ per ogni $i \geq 8$.

Come è facile osservare, non è vero che ogni sequenza di derivazioni dirette conduce prima o poi ad una stringa del linguaggio generato dalla grammatica.

Esempio 2.3 Se si considera la grammatica dell'Esempio 2.2, è facile trovare una sequenza di produzioni che genera una stringa in cui sono presenti anche dei non terminali ed alla quale non possono essere applicate ulteriori produzioni. Ad esempio,

$$\begin{array}{rcl} S & \Longrightarrow & aSBC \\ & \Longrightarrow & aaSBCBC \\ & \Longrightarrow & aabFCBC \\ & \Longrightarrow & aabcGBC \\ & \Longrightarrow & aabcBC. \end{array}$$

Per lo stesso motivo possono esistere grammatiche che generano il linguaggio vuoto, che cioè non generano alcuna stringa di V_T^*.

Esempio 2.4 La grammatica $\mathcal{G} = \langle \{a, b\}, \{S, A\}, P, S \rangle$ con produzioni P

$$\begin{array}{rcl} S & \longrightarrow & Ab \\ A & \longrightarrow & Sa \end{array}$$

genera il linguaggio vuoto Λ.

Si osservi che tutti gli enunciati del tipo "la grammatica $\mathcal{G}$ genera il linguaggio L" (dove L viene descritto tramite una proprietà caratteristica) spesso introdotti in questo volume dovrebbero essere, almeno in linea di principio, formalmente dimostrati provando che la grammatica genera tutte e sole le parole del linguaggio.

Esempio 2.5 Data la grammatica $\mathcal{G} = \langle \{a, b, c\}, \{S, A\}, P, S \rangle$ con produzioni P

$$\begin{array}{rcl} S & \longrightarrow & aSc \mid A \\ A & \longrightarrow & bAc \mid \varepsilon, \end{array}$$

dimostriamo che $\mathcal{G}$ genera il linguaggio

$$L = \left\{ a^n b^m c^{n+m} \mid n, m \geq 0 \right\}.$$

Proviamo dapprima che $\mathcal{L}(\mathcal{G}) \subseteq L$. A tale scopo è immediato verificare che tutte le forme di frase costruite da $\mathcal{G}$ sono del tipo

- $a^k S c^k$, $k \geq 0$, oppure
- $a^k b^j A c^{k+j}$, $k \geq 0$ e $j \geq 0$.

Ne segue che ogni parola z generata da $\mathcal{G}$ è ottenuta tramite una derivazione del tipo

$$S \stackrel{k}{\Longrightarrow} a^k S c^k \stackrel{1}{\Longrightarrow} a^k A c^k \stackrel{j}{\Longrightarrow} a^k b^j A c^{k+j} \stackrel{1}{\Longrightarrow} a^k b^j c^{k+j} = z.$$

Ogni parola derivata dunque appartiene a L.

Proviamo ora che $L \subseteq \mathcal{L}(\mathcal{G})$. A tale scopo basta osservare che ogni stringa di $z \in L$ è del tipo $a^n b^m c^{n+m}$ e che essa viene generata da $\mathcal{G}$ attraverso la seguente derivazione:

$$S \stackrel{m}{\Longrightarrow} a^m S c^m \stackrel{1}{\Longrightarrow} a^m A c^m \stackrel{n}{\Longrightarrow} a^m b^n A c^{m+n} \stackrel{1}{\Longrightarrow} a^m b^n c^{m+n} = z.$$

Esiste dunque in $\mathcal{G}$ una derivazione che costruisce z.

Come si è visto nell'esempio precedente, la dimostrazione del fatto che una grammatica genera un dato linguaggio può essere eccessivamente tediosa, soprattutto se si considera che in molti casi (come accade nell'esempio stesso) tale proprietà della grammatica si può verificare facilmente in modo intuitivo. Per tale motivo, nel seguito, non forniremo più dimostrazioni formali di tale

tipo e ci affideremo all'intuizione del lettore, il quale potrà comunque convincersi con facilità che le grammatiche date generano effettivamente i linguaggi voluti.

Esercizio 2.1 Data la grammatica $\mathcal{G} = \langle \{a\}, \{S, I, F, M\}, P, S \rangle$ con produzioni P

$$\begin{array}{rcl} S & \longrightarrow & a \mid aa \mid IaF \\ aF & \longrightarrow & Maa \mid MaaF \\ aM & \longrightarrow & Maa \\ IM & \longrightarrow & Ia \mid aa, \end{array}$$

provare che $\mathcal{G}$ genera il linguaggio $\{a^{2^n} \mid n \geq 0\}$.

Esercizio 2.2 Definire una grammatica che il linguaggio $\{a^n b^m c^p \mid n = m \vee m = p\}$ e dimostrare la sua correttezza.

In generale uno stesso linguaggio può essere generato da (infinite) grammatiche diverse.

Definizione 2.7 *Due grammatiche $\mathcal{G}_1$ e $\mathcal{G}_2$ si dicono* equivalenti *se $L(\mathcal{G}_1) = L(\mathcal{G}_2)$.*

Esercizio 2.3 Dimostrare che la grammatica con produzioni

$$S \longrightarrow aS \mid b$$

e la grammatica con produzioni

$$\begin{array}{rcl} S & \longrightarrow & b \mid Ab \\ A & \longrightarrow & Aa \mid a \end{array}$$

sono equivalenti.

Introducendo restrizioni sulla struttura delle produzioni si ottengono vari tipi di grammatiche. Nel resto di questa sezione vedremo le caratteristiche di tali grammatiche e la classificazione di linguaggi cui esse danno luogo.

2.1.1 Grammatiche di tipo 0

Le grammatiche di tipo 0, dette anche *non limitate*, definiscono la classe di linguaggi più ampia possibile.[1] In esse le produzioni sono del tipo più generale:

$$\alpha \longrightarrow \beta,\ \alpha \in V^* \circ V_N \circ V^*,\ \beta \in V^*.$$

[1] Più ampia nell'ambito dei linguaggi descrivibili con grammatiche. Esistono tuttavia linguaggi per cui non esiste una grammatica corrispondente. Questo aspetto sarà trattato nel Capitolo 7.

Si noti che queste grammatiche ammettono anche derivazioni che "accorciano" le forme di frase , come ad esempio quelle che si ottengono applicando le ε-produzioni.

Esempio 2.6 Le produzioni

$$\begin{aligned} S &\longrightarrow aSa \mid aAb \mid aAa \mid \varepsilon \\ aAa &\longrightarrow a \mid \varepsilon \\ aaAb &\longrightarrow b \mid \varepsilon \end{aligned}$$

appartengono ad una grammatica di tipo 0. Come conseguenza immediata, la grammatica considerata nell'Esempio 2.2 risulta essere di tipo 0.

Esempio 2.7 La grammatica

$$\mathcal{G} = \langle \{a, b\}, \{S, A\}, P, S \rangle,$$

in cui P è

$$\begin{aligned} S &\longrightarrow aAb \\ aA &\longrightarrow aaAb \\ A &\longrightarrow \varepsilon, \end{aligned}$$

è anch'essa una grammatica di tipo 0 e genera il linguaggio $L = \{a^n b^n \mid n \geq 1\}$.

I linguaggi generabili da grammatiche di tipo 0 si dicono *linguaggi di tipo 0*.

2.1.2 Grammatiche di tipo 1

Queste grammatiche, dette anche *contestuali* o *context sensitive* (CS), ammettono qualunque regola di produzione che non riduca la lunghezza delle stringhe, cioè produzioni del tipo:

$$\alpha \longrightarrow \gamma, \alpha \in V^* \circ V_N \circ V^*, \gamma \in V^+, |\alpha| \leq |\gamma| \, .$$

Esempio 2.8 Le produzioni

$$\begin{aligned} S &\longrightarrow aSa \mid aAb \mid aAa \\ aA &\longrightarrow aa \\ Ab &\longrightarrow aab \end{aligned}$$

appartengono ad una grammatica di tipo 1.

I linguaggi generabili da grammatiche di tipo 1 si dicono *linguaggi di tipo 1*, o *contestuali*, o *context sensitive* (CS).

Esempio 2.9 Come si è potuto vedere, in base all'Esempio 2.2, il linguaggio $\{a^n b^n c^n \mid n \geq 1\}$ è certamente di tipo 0. Lo stesso linguaggio può venir generato da una grammatica di tipo 1 equivalente avente le produzioni $S \longrightarrow aBSc \mid abc$, $Ba \longrightarrow aB$, $Bb \longrightarrow bb$. Dunque il linguaggio $\{a^n b^n c^n \mid n \geq 1\}$ è contestuale.

Il termine "linguaggio contestuale", deriva dal fatto che, storicamente, questi linguaggi sono stati definiti da Chomsky come la classe dei linguaggi generabili da grammatiche aventi produzioni "contestuali" del tipo

$$\beta_1 A \beta_2 \longrightarrow \beta_1 \gamma \beta_2,\ A \in V_N,\ \beta_1, \beta_2 \in V^*,\ \gamma \in V^+,$$

in cui si esprime il fatto che la produzione $A \longrightarrow \gamma$ può essere applicata solo se A si trova nel contesto $\langle \beta_1, \beta_2 \rangle$. Ad esempio, nella grammatica per il linguaggio $\{a^n b^n c^n \mid n \geq 1\}$ riportata nell'Esempio 2.9, la produzione $Bb \longrightarrow bb$ ha la struttura suddetta, cioè il carattere B può essere rimpiazzato dal carattere b solo se alla sua destra è presente il contesto b.

Esercizio 2.4 Mostrare che le grammatiche con produzioni contestuali possono generare tutti e soli i linguaggi di tipo 1.
[Suggerimento: Ridursi a una grammatica con produzioni senza simboli terminali nelle parti sinistre e sostituire ciascuna produzione non avente la struttura contestuale suddetta con un insieme di produzioni contestuali equivalente.]

2.1.3 Grammatiche di tipo 2

Queste grammatiche, dette anche *non contestuali* o *context free* (CF), ammettono produzioni del tipo:

$$A \longrightarrow \beta, \quad A \in V_N,\ \beta \in V^+$$

cioè produzioni in cui ogni non terminale A può essere riscritto in una stringa β indipendentemente dal contesto in cui esso si trova.

Esempio 2.10 Il linguaggio generato dalla grammatica dell'Esempio 2.7 può essere generato anche dalla grammatica equivalente di tipo 2 con produzioni:

$$S \longrightarrow aSb \mid ab.$$

Esempio 2.11 Un esempio di grammatica non contestuale è costituito dalla grammatica che, a partire dall'assioma E, genera espressioni aritmetiche di somme e moltiplicazioni in una variabile i, con le seguenti produzioni:

$$\begin{aligned} E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow i \mid (E). \end{aligned}$$

I linguaggi generabili da grammatiche di tipo 2 vengono detti *linguaggi di tipo 2* o *non contestuali* o *context free* (CF).

Esempio 2.12 Il linguaggio delle parentesi ben bilanciate è di tipo 2. Esso può essere generato dalla grammatica $\mathcal{G} = \langle \{(,)\}, \{S\}, P, S \rangle$, dove P è l'insieme delle produzioni

$$\begin{aligned} S &\longrightarrow () \\ S &\longrightarrow SS \\ S &\longrightarrow (S). \end{aligned}$$

Esercizio 2.5 Definire una grammatica non contestuale per generare tutte le stringhe palindrome, cioè quelle che risultano uguali se lette da destra verso sinistra o da sinistra verso destra.

2.1.4 Grammatiche di tipo 3

Queste grammatiche, dette anche *lineari destre* o *regolari*, ammettono produzioni del tipo:

$$A \longrightarrow \delta, \quad A \in V_N,\, \delta \in (V_T \circ V_N) \cup V_T.$$

Il termine "regolare" deriva dal fatto che, come vedremo più avanti (vedi Sezione3.6, tali linguaggi sono rappresentabili per mezzo di espressioni regolari. Il termine "lineare" deriva dal fatto che al lato destro di ogni produzione compare al più un simbolo non terminale.

I linguaggi generabili da grammatiche di tipo 3 vengono detti *linguaggi di tipo 3* o *regolari*.

Esempio 2.13 La grammatica definita da

$$\mathcal{G} = \langle \{a, b\}, \{S\}, P, S \rangle$$

in cui P contiene le produzioni

$$\begin{array}{lcl} S & \longrightarrow & aS \\ S & \longrightarrow & b \end{array}$$

è una grammatica di tipo 3 e genera il linguaggio regolare $L = \{a^n b \mid n \geq 0\}$.

Esercizio 2.6 Definire una grammatica regolare per il linguaggio delle stringhe sull'alfabeto $\{a, b\}$ che contengono un numero dispari di a.

Esercizio 2.7 Con riferimento all'Esercizio 1.24 e alla Figura 1.6, definire una grammatica regolare che generi le stringhe corrispondenti a tutti i percorsi, anche passanti più volte per uno stesso nodo, tra A e B.

In modo del tutto analogo, i linguaggi regolari si possono definire anche mediante grammatiche *lineari sinistre* caratterizzate da regole del tipo:

$$A \longrightarrow \delta, \quad A \in V_N,\, \delta \in (V_N \circ V_T) \cup V_T.$$

Esempio 2.14 Come si è visto nell'Esercizio 2.3, il linguaggio $\{a^n b \mid n \geq 0\}$ può anche essere generato attraverso le produzioni

$$\begin{array}{lcl} S & \longrightarrow & Ab \mid b \\ A & \longrightarrow & Aa \mid a. \end{array}$$

Vedremo nel seguito (Esercizio 3.11) che per ogni grammatica lineare destra ne esiste una lineare sinistra equivalente e viceversa.

2.1.5 Gerarchia di Chomsky

Si verifica immediatamente che per ogni $0 \leq n \leq 2$, ogni grammatica di tipo $n+1$ è anche di tipo n, e pertanto l'insieme dei linguaggi di tipo n contiene tutti i linguaggi di tipo $n+1$, formando quindi una gerarchia, detta *Gerarchia di Chomsky*. Peraltro è possibile mostrare che il contenimento è stretto, come illustrato in Figura 2.1.

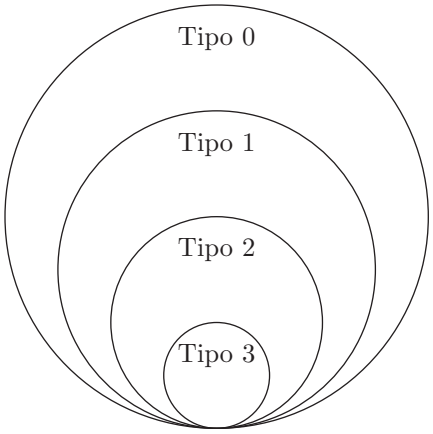


FIGURA 2.1 Relazioni tra i tipi di linguaggi.

Definizione 2.8 *Un linguaggio L viene detto* strettamente *di tipo n se esiste una grammatica $\mathcal{G}$ di tipo n che genera L e non esiste alcuna grammatica $\mathcal{G}'$ di tipo $m > n$ che possa generarlo.*

Esempio 2.15 Il linguaggio $L = \{a^n b^n \mid n \geq 1\}$ può essere generato sia dalla grammatica dell'Esempio 2.7, che è di tipo 0, perché ammette ε-produzioni, sia dalla grammatica dell'Esempio 2.10 che è una grammatica non contestuale, per cui il linguaggio è di tipo 2. Tuttavia è possibile dimostrare che non esiste nessuna grammatica di tipo 3 che lo può generare (vedi Teorema 3.5): pertanto esso è strettamente di tipo 2.

Le proprietà di inclusione stretta delle varie classi di linguaggi, tra cui quelle citate nell'esempio precedente, saranno affrontate successivamente.

Per comodità del lettore la classificazione delle grammatiche viene riportata nella Tabella 2.1. Le grammatiche di tipo 3, le più povere, permettono di generare linguaggi molto limitati. Ad esempio, con queste grammatiche è possibile generare il linguaggio $L = \{a^n b \mid n \geq 0\}$, mentre non si può generare il linguaggio $L = \{a^n b^n \mid n \geq 0\}$, per il quale, come vedremo nella Sezione 3.4, è invece necessaria una grammatica di tipo 2.

TIPO	PRODUZIONI	DOVE
TIPO 0 non limitate	$\alpha \longrightarrow \beta$	$\alpha \in V^* \circ V_N \circ V^*,\ \beta \in V^*$
TIPO 1 contestuali	$\alpha \longrightarrow \beta$	$\mid \alpha \mid \leq \mid \beta \mid,$ $\alpha \in V^* \circ V_N \circ V^*,\ \beta \in V^+$
TIPO 2 non contestuali	$A \longrightarrow \beta$	$A \in V_N,\ \beta \in V^+$
TIPO 3 regolari	$A \longrightarrow \delta$	$A \in V_N,\ \delta \in (V_T \cup (V_T \circ V_N))$

Tabella 2.1 Classificazione delle grammatiche secondo Chomsky.

Le grammatiche non contestuali consentono di generare strutture sintattiche anche piuttosto articolate come quelle utilizzate dai linguaggi di programmazione (vedi Sezione 2.4).

Ovviamente le grammatiche di tipo 1 sono ancora più potenti e permettono di descrivere linguaggi ancora più complessi, come ad esempio il linguaggio $\{a^n b^n c^n \mid m \geq 0\}$, che è strettamente di tipo 1.

Esempio 2.16 La seguente grammatica contestuale genera il linguaggio $\{ww \mid w \in (a+b)^+\}$.

$$S \longrightarrow aAS \mid bBS \mid A_0a \mid B_0b \tag{2.1}$$

$$\begin{aligned} Aa &\longrightarrow aA \\ Ab &\longrightarrow bA \\ Ba &\longrightarrow aB \\ Bb &\longrightarrow bB \end{aligned} \tag{2.2}$$

$$\begin{aligned} AA_0 &\longrightarrow A_0a \\ BA_0 &\longrightarrow A_0b \\ AB_0 &\longrightarrow B_0a \\ BB_0 &\longrightarrow B_0b \end{aligned} \tag{2.3}$$

$$\begin{aligned} A_0 &\longrightarrow a \\ B_0 &\longrightarrow b \end{aligned} \tag{2.4}$$

Consideriamo ad esempio una possibile derivazione, in questa grammatica, della stringa *abbabb*:

$$\begin{array}{lllllll} S & \Longrightarrow & aAS & \Longrightarrow & aAbBS & \Longrightarrow & aAbBB_0b \\ & \Longrightarrow & aAbB_0bb & \Longrightarrow & abAB_0bb & \Longrightarrow & abB_0abb \\ & \Longrightarrow & abbabb. & & & & \end{array} \tag{2.5}$$

L'ultimo esempio verrà utilizzato per mostrare come un linguaggio può essere generato da grammatiche di tipo diverso.

Esempio 2.17 Il linguaggio

$$L = \left\{(x\#)^+ \;\mid\; x = \text{ permutazione di } \langle a, b, c\rangle\right\}$$

permette di generare stringhe costituite dalla concatenazione di un numero qualunque, maggiore di 0, di permutazioni dei terminali a, b, c, terminate dal carattere $\#$. È semplice descrivere una grammatica di tipo 1 in grado di generare questo linguaggio: essa è la grammatica

$$\mathcal{G} = \langle\{a, b, c, \#\}, \{S, A, B, C\}, P, S\rangle,$$

dove P contiene le produzioni

$$\begin{array}{rcl} S & \longrightarrow & ABC\#S \;\mid\; ABC\# \\ AB & \longrightarrow & BA \\ AC & \longrightarrow & CA \\ BC & \longrightarrow & CB \\ A & \longrightarrow & a \\ B & \longrightarrow & b \\ C & \longrightarrow & c. \end{array}$$

Per lo stesso linguaggio si può definire la seguente grammatica di tipo 2:

$$\mathcal{G}' = \langle\{a, b, c, \#\}, \{S, E\}, P', S\rangle,$$

dove P' contiene le produzioni

$$\begin{array}{rcl} S & \longrightarrow & E\#S \;\mid\; E\# \\ E & \longrightarrow & abc \;\mid\; acb \;\mid\; cba \;\mid\; bac \;\mid\; bca \;\mid\; cab. \end{array}$$

Si può infine trovare anche una grammatica regolare che genera lo stesso linguaggio, come la seguente:

$$\mathcal{G}'' = \langle\{a, b, c, \#\}, \{S, R, X, Y, Z, X', Y', Z', X'', Y'', Z''\}, P'', S\rangle,$$

dove P'' contiene le produzioni

$$\begin{array}{lcl} S & \longrightarrow & aX \mid bY \mid cZ \\ X & \longrightarrow & bX' \mid cX'' \\ X' & \longrightarrow & cR \\ X'' & \longrightarrow & bR \\ Y & \longrightarrow & aY' \mid cY'' \\ Y' & \longrightarrow & cR \\ Y'' & \longrightarrow & aR \\ Z & \longrightarrow & aZ' \mid bZ'' \\ Z' & \longrightarrow & bR \\ Z'' & \longrightarrow & aR \\ R & \longrightarrow & \#S \mid \#. \end{array}$$

Come si vede, a parità di linguaggio, l'utilizzazione di una grammatica di tipo più alto può comportare la necessità di un numero maggiore di produzioni.

Esercizio 2.8 Cosa succederebbe se considerassimo permutazioni di $4, 5, \ldots, n$ caratteri anziché 3? Come varierebbe il numero delle produzioni necessarie nelle tre grammatiche?

2.2 Grammatiche con ε-produzioni

In base alle definizioni date finora, non è possibile generare la stringa vuota con grammatiche di tipo 1, 2 o 3. D'altra parte in molti casi è possibile generare linguaggi contenenti anche la stringa vuota senza ricorrere all'intero potere generativo delle grammatiche di tipo 0. Come vedremo, tali linguaggi possono essere generati apportando lievi modifiche al tipo di produzioni ammesse per le grammatiche non contestuali e regolari. Queste modifiche consistono semplicemente nell'aggiunta di opportune ε-produzioni. Inoltre, è interessante valutare l'impatto determinato dall'introduzione di ε-produzioni anche per le grammatiche contestuali.

Innanzi tutto possiamo mostrare che, dato un qualunque linguaggio di tipo 1, 2 o 3, è possibile modificare la grammatica che lo genera in modo da ottenere lo stesso linguaggio arricchito della sola stringa vuota; a tal fine basta semplicemente limitare le ε-produzioni al solo assioma, procedendo come descritto di seguito.

Se una grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ di tipo 1, 2 o 3 genera un linguaggio L, per poter generare il linguaggio $L \cup \{\varepsilon\}$ è sufficiente utilizzare la grammatica

$$\mathcal{G}' = \langle V_T, V_N \cup \{S'\}, P', S'\rangle,$$

dove

$$P' = P \cup \{S' \longrightarrow \varepsilon\} \cup \{S' \longrightarrow \beta \mid S \longrightarrow \beta \in P\}.$$

Esempio 2.18 Consideriamo le produzioni della grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$, già introdotta nell'Esempio 2.9,

$$\begin{array}{rcl} S & \longrightarrow & aBSc \mid abc \\ Ba & \longrightarrow & aB \\ Bb & \longrightarrow & bb. \end{array}$$

Risulta $L(\mathcal{G}) = \{a^n b^n c^n \mid n \geq 1\}$. È immediato verificare che la grammatica $\mathcal{G}' = \langle V_T, V_N \cup \{S'\}, P', S'\rangle$ con produzioni P'

$$\begin{array}{rcl} S' & \longrightarrow & aBSc \mid abc \mid \varepsilon \\ S & \longrightarrow & aBSc \mid abc \\ Ba & \longrightarrow & aB \\ Bb & \longrightarrow & bb \end{array}$$

genera $\{a^n b^n c^n \mid n \geq 0\}$.

Come si può osservare, le ε-produzioni si possono anche aggiungere direttamente all'assioma delle grammatiche di tipo 1, 2 o 3 (senza introdurre un nuovo assioma, come fatto sopra), senza che questo abbia conseguenze particolarmente significative, purché l'assioma non appaia mai al lato destro di una produzione. In effetti, è interessante osservare che tale condizione è essenziale per limitare le conseguenze dell'inserimento di ε-produzioni. Infatti, consideriamo ad esempio la grammatica con produzioni

$$\begin{array}{rcl} S & \longrightarrow & Ub \\ U & \longrightarrow & ab \mid S \end{array}$$

che genera le stringhe del tipo ab^*bb. Se aggiungiamo la ε-produzione $S \longrightarrow \varepsilon$ è immediato verificare che la nuova grammatica non solo genera la stringa vuota, ma anche tutte le stringhe del tipo bb^*.

Esaminiamo ora, più in generale, le conseguenze provocate dalla presenza indiscriminata di ε-produzioni all'interno di una grammatica. A tal proposito, si può mostrare che la situazione è diversa a seconda che si trattino grammatiche di tipo 1, 2 o 3. Vediamo innanzi tutto come, nel primo caso, l'aggiunta non controllata di ε-produzioni aumenti in modo sostanziale il potere generativo della grammatica.

Esempio 2.19 Si consideri una grammatica

$$\mathcal{G} = \langle V_T, V_N, P, S\rangle$$

che abbia tutte produzioni di tipo 1 tranne una sola regola del tipo $\alpha \longrightarrow \beta$ con $\mid \alpha \mid \geq \mid \beta \mid$: sia essa la regola $AB \longrightarrow C$. Si può costruire allora la grammatica $\mathcal{G}' = \langle V_T, V_N \cup \{H\}, P', S\rangle$ con produzioni o di tipo 1 o ε-produzioni, che rispetto a $\mathcal{G}$ ha un non terminale in più ed un insieme di regole di produzione P':

$$P' = P - \{AB \longrightarrow C\} \cup \{AB \longrightarrow CH, H \longrightarrow \varepsilon\}.$$

Osservando che alla singola derivazione $AB \underset{\mathcal{G}}{\Longrightarrow} C$ corrisponde la sequenza di derivazioni $AB \underset{\mathcal{G}'}{\Longrightarrow} CH \underset{\mathcal{G}'}{\Longrightarrow} C$, e che l'introduzione del simbolo non terminale H non estende l'insieme delle stringhe generabili, ne deriva che $L(\mathcal{G}) = L(\mathcal{G}')$.

Mediante una semplice generalizzazione dell'esempio precedente possiamo dimostrare il seguente teorema.

Teorema 2.1 *Data una grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ di tipo 0, esiste una grammatica $\mathcal{G}'$, ottenuta estendendo con opportune ε-produzioni una grammatica di tipo 1, equivalente a $\mathcal{G}$.*

Dimostrazione. La grammatica $\mathcal{G}' = \langle V'_T, V'_N, P', S'\rangle$ è caratterizzata da: $V'_T = V_T$, $V'_N = V_N \cup \{X\}$, con $X \notin V_N$, $S' = S$ e P' ottenuto da P aggiungendo la produzione $X \longrightarrow \varepsilon$ e sostituendo ad ogni produzione $\phi \longrightarrow \psi$ con $|\,\phi\,| > |\,\psi\,| > 0$ la produzione

$$\phi \longrightarrow \psi\ \underbrace{X \ldots X}_{|\phi| - |\psi| \text{ volte}}.$$

È semplice verificare che con la grammatica $\mathcal{G}'$ sono derivabili tutte e sole le stringhe di V_T^* derivabili con la grammatica $\mathcal{G}$. □

Nel caso di grammatiche di tipo 2 o 3 abbiamo invece che l'aggiunta indiscriminata di ε-produzioni non altera il potere generativo delle grammatiche. Infatti si può dimostrare che data una grammatica del tipo suddetto estesa con ε-produzioni, ne possiamo sempre costruire una equivalente, dello stesso tipo, che usa ε-produzioni solo a partire dall'assioma (nel caso che ε appartenga al linguaggio da generare) o non ne usa affatto (in caso contrario). Vediamo innanzi tutto, mediante due esempi, come ciò sia possibile.

Esempio 2.20 Nella grammatica regolare avente le produzioni

$$\begin{aligned} S &\longrightarrow bX \mid aB \\ B &\longrightarrow cX \\ X &\longrightarrow \varepsilon \end{aligned}$$

la produzione vuota si può eliminare riscrivendo le produzioni nella forma

$$\begin{aligned} S &\longrightarrow b \mid aB \\ B &\longrightarrow c \end{aligned}$$

in cui la ε-produzione è completamente scomparsa.

Esempio 2.21 Nella grammatica non contestuale

$$\begin{aligned} S &\longrightarrow AB \mid aB \mid B \\ A &\longrightarrow ab \mid aB \\ B &\longrightarrow cX \mid X \\ X &\longrightarrow \varepsilon \end{aligned}$$

la ε-produzione si può far "migrare" verso l'assioma sostituendola all'indietro e ottenendo prima le produzioni

$$\begin{array}{lcl} S & \longrightarrow & AB \mid aB \mid B \\ A & \longrightarrow & ab \mid aB \\ B & \longrightarrow & c \mid \varepsilon \end{array}$$

e poi, ripetendo il procedimento, le produzioni

$$\begin{array}{lcl} S & \longrightarrow & AB \mid A \mid aB \mid a \mid B \mid \varepsilon \\ A & \longrightarrow & ab \mid aB \mid a \\ B & \longrightarrow & c. \end{array}$$

Non è difficile fornire anche una dimostrazione formale di quanto detto.

Teorema 2.2 *Data una grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ il cui insieme di produzioni P comprende soltanto produzioni di tipo non contestuale e produzioni vuote, esiste una grammatica non contestuale $\mathcal{G}'$ tale che $L(\mathcal{G}') = L(\mathcal{G}) - \{\varepsilon\}$.*

Dimostrazione. Il primo passo della dimostrazione consiste nel costruire $\mathcal{G}' = \langle V_T, V_N, P', S\rangle$. A tale scopo determiniamo innanzi tutto l'insieme $N \subseteq V_N$ dei simboli che si annullano, cioè i non terminali da cui è possibile derivare ε tramite $\mathcal{G}$. Ciò può essere fatto applicando l'Algoritmo 2.1. Tale algoritmo costruisce una sequenza $N_0, N_1, \ldots, N_k = N$ di sottoinsiemi di V_N, ponendo inizialmente $N_0 = \{A \in V_N \mid A \longrightarrow \varepsilon \in P\}$ e costruendo N_{i+1} a partire da N_i nel modo seguente:

$$N_{i+1} = N_i \cup \left\{ B \in V_N \mid (B \longrightarrow \beta \in P) \wedge (\beta \in N_i^+) \right\}.$$

L'algoritmo termina quando $N_{k+1} = N_k$, $k \geq 0$. Si noti che $\varepsilon \in L(\mathcal{G})$ se e solo se $S \in N$. Per costruire l'insieme P' delle produzioni di $\mathcal{G}'$ si può applicare

input grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$;
output insieme $N \subseteq V_N$ dei simboli che si annullano;
begin
  $N := \{A \in V_N \mid A \longrightarrow \varepsilon \in P\}$;
  repeat
    $\widehat{N} := N$;
    $N := \widehat{N} \cup \left\{ B \in V_N \mid (B \longrightarrow \beta \in P) \wedge (\beta \in \widehat{N}^+) \right\}$
  until $N = \widehat{N}$
end.

Algoritmo 2.1: Determina Simboli Annullabili

l'Algoritmo 2.2, il quale esamina ciascuna produzione $A \longrightarrow \alpha$ di P, con l'esclusione delle ε-produzioni. Se nessun simbolo di α si annulla l'algoritmo inserisce la produzione $A \longrightarrow \alpha$ in P'; altrimenti α contiene $k > 0$ simboli che si annullano. In tal caso l'algoritmo aggiunge a P' tutte le possibili produzioni ottenute da $A \longrightarrow \alpha$ eliminando da α uno dei sottoinsiemi di simboli che si annullano: i simboli di α che si annullano sono considerati con la propria molteplicità. Ad esempio, se $N = \{S, A, B\}$ ed esiste in P la produzione $A \longrightarrow BCBA$, allora vengono inserite in P' le produzioni $A \longrightarrow BCBA$, $A \longrightarrow CBA$, $A \longrightarrow BCA$, $A \longrightarrow BCB$, $A \longrightarrow CA$, $A \longrightarrow CB$, $A \longrightarrow BC$ e $A \longrightarrow C$. Fa eccezione, nel caso $k =\mid \alpha \mid$ (in cui cioè tutti i simboli a destra nella produzione si annullano), il sottoinsieme di cardinalità $\mid \alpha \mid$, la cui eliminazione darebbe luogo ad una ε-produzione, che non viene aggiunta.

Dunque, in corrispondenza alla produzione $A \longrightarrow \alpha$, l'algoritmo aggiunge a P' $\min\{2^{|\alpha|} - 1, 2^k\}$ produzioni.

La procedura viene ripetuta per ciascuna produzione non vuota di P. Per

input grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$, insieme $N \subseteq V_N$ dei simboli che si annullano;
output insieme P' delle produzioni di $\mathcal{G}'$;
begin
  $P' := \emptyset$;
  for each $A \longrightarrow \alpha \in P$ con $\alpha \neq \varepsilon$ **do**
    begin
      sia $\alpha = Z_1, \ldots, Z_t$;
      $J := \{i \mid Z_i \in N\}$;
      for each $J' \in 2^J$ **do**
        if $J' \neq \{1, \ldots, t\}$ **then**
          begin
            sia β la stringa ottenuta eliminando da α ogni Z_i con $i \in J'$;
            $P' := P' \cup \{A \longrightarrow \beta\}$
          end
    end
end.

Algoritmo 2.2: Elimina ε-produzioni

provare che $L(\mathcal{G}') = L(\mathcal{G}) - \{\varepsilon\}$ basta mostrare, per induzione sulla lunghezza della derivazione, che $\forall A \in V_N$ e $\forall w \in V_T^+$:

$$A \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} w \iff (w \neq \varepsilon \ \wedge \ A \underset{\mathcal{G}'}{\overset{*}{\Longrightarrow}} w).$$

Tale dimostrazione è lasciata al lettore come esercizio (Esercizio 2.9). □

Si osservi che, nel caso in cui $L(\mathcal{G})$ contiene ε, si può ottenere da $\mathcal{G}'$ una grammatica equivalente a $\mathcal{G}$ tramite la semplice introduzione di una ε-produzione sull'assioma di $\mathcal{G}'$, applicando la tecnica vista nell'Esempio 2.18.

Esempio 2.22 Consideriamo la grammatica $\mathcal{G} = \langle\{a, b\}, \{S, A, B\}, P, S\rangle$, le cui produzioni P sono:

$$\begin{array}{lcl} S & \longrightarrow & A \mid SSa \\ A & \longrightarrow & B \mid Ab \mid \varepsilon \\ B & \longrightarrow & S \mid ab \mid aA. \end{array}$$

Applicando la tecnica descritta nella dimostrazione del Teorema 2.2 costruiamo inizialmente gli insiemi $N_0 = \{A\}$, $N_1 = \{S, A\}$ e $N = N_2 = \{S, A, B\}$. La grammatica $\mathcal{G}' = \langle\{a, b\}, \{S, A, B\}, P', S\rangle$ che genera $L(\mathcal{G}) - \varepsilon$ ha le seguenti produzioni P':

$$\begin{array}{lcl} S & \longrightarrow & A \mid SSa \mid Sa \\ A & \longrightarrow & B \mid Ab \mid b \\ B & \longrightarrow & S \mid ab \mid aA \mid a. \end{array}$$

Una grammatica equivalente a $\mathcal{G}$ senza ε-produzioni (se non al più sull'assioma) è dunque $\langle\{a, b\}, \{T, S, A, B\}, P' \cup \{T \longrightarrow S, T \longrightarrow \varepsilon\}, T\rangle$.

Esercizio 2.9 Completare la dimostrazione del Teorema 2.2.

Nel caso particolare in cui la grammatica è regolare è possibile dimostrare un risultato analogo.

Teorema 2.3 *Data una grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ il cui insieme di produzioni P è partizionabile in un sottoinsieme di produzioni di tipo regolare e in un sottoinsieme di produzioni vuote, esiste una grammatica regolare $\mathcal{G}' = \langle V_T, V_N, P', S\rangle$ tale che $L(\mathcal{G}') = L(\mathcal{G}) - \{\varepsilon\}$.*

Dimostrazione. Innanzi tutto notiamo che $\varepsilon \in L(\mathcal{G})$ se e solo se $S \longrightarrow \varepsilon \in P$, per cui per dimostrare il teorema basta provare che $\mathcal{G}' = \langle V_T, V_N, P', S\rangle$ è equivalente alla grammatica $\mathcal{G}'' = \langle V_T, V_N, P - \{S \longrightarrow \varepsilon\}, S\rangle$. Sebbene la dimostrazione del Teorema 2.2 possa essere facilmente adattata al caso in esame, preferiamo qui procedere con una tecnica che sfrutta al meglio le peculiarità delle produzioni di tipo regolare.

Costruiamo dunque l'insieme P' delle produzioni di $\mathcal{G}'$. Inseriamo inizialmente in P' tutte le produzioni non vuote presenti in P. Quindi, per ciascuna produzione $A \longrightarrow \varepsilon \in P - \{S \longrightarrow \varepsilon\}$, consideriamo tutte le produzioni in P del tipo $B \longrightarrow aA$, con $B \in V_N$ e $a \in V_T$: per ciascuna di queste inseriamo in P' la produzione $B \longrightarrow a$.

Per completare la dimostrazione del teorema è sufficiente mostrare che $\forall A \in V_N$ e $\forall w \in V_T^*$

$$A \overset{*}{\underset{\mathcal{G}''}{\Longrightarrow}} w \iff A \overset{*}{\underset{\mathcal{G}'}{\Longrightarrow}} w.$$

Ciò può essere facilmente provato per induzione sulla lunghezza della derivazione. □

Esercizio 2.10 Data la seguente grammatica,

$$\begin{aligned} S &\longrightarrow ASB \mid XL \mid ASCL \\ X &\longrightarrow AB \mid \varepsilon \\ C &\longrightarrow b \\ B &\longrightarrow b \\ A &\longrightarrow a \\ L &\longrightarrow a \mid \varepsilon. \end{aligned}$$

Eliminare, se possibile, le ε-produzioni in essa presenti.

Nel seguito si adotterà la classificazione delle grammatiche formali, e dei corrispondenti linguaggi generati, proposta da Chomsky, con la modifica, alla luce di quanto visto nella presente sezione, che nel caso di grammatiche di tipo 2 e 3 è consentita la presenza di ε-produzioni, mentre nel caso di grammatiche di tipo 1 è ammessa la presenza di ε-produzioni solo sull'assioma, a condizione che questo non compaia mai nella parte destra di una produzione.

2.3 Linguaggi lineari

Nella Sezione 4.1 mostreremo che tutti i linguaggi non contestuali possono essere generati mediante grammatiche in cui il lato destro di una produzione contiene al più due non terminali. Una interessante restrizione della classe dei linguaggi CF è rappresentata dai linguaggi generati da grammatiche CF in cui ogni produzione ha al più un simbolo non terminale nella parte destra.

Definizione 2.9 *Si dice* lineare *una grammatica non contestuale in cui la parte destra di ogni produzione contenga al più un non terminale.*

Chiaramente, come visto nella Sezione 2.1, le grammatiche di tipo 3 sono tutte grammatiche lineari: in particolare esse sono *lineari destre* se le produzioni sono del tipo $A \longrightarrow aB$ e *lineari sinistre* se le produzioni sono del tipo $A \longrightarrow Ba$.

La classe dei linguaggi *lineari* è la classe dei linguaggi generabili con grammatiche lineari. Si può dimostrare che questa classe è strettamente contenuta in quella dei linguaggi non contestuali e contiene strettamente quella dei linguaggi regolari, come indicato in Figura 2.2. Il seguente esempio mostra una grammatica lineare per un linguaggio che, come vedremo, è strettamente non contestuale.

Esempio 2.23 La grammatica con produzioni

$$\begin{aligned} S &\longrightarrow aSb \\ S &\longrightarrow ab \end{aligned}$$

è una grammatica lineare. Essa genera il linguaggio $L = \{a^n\ b^n \mid n \geq 1\}$.

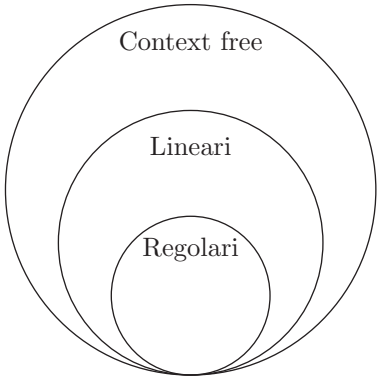


FIGURA 2.2 Relazione tra linguaggi lineari, regolari e context free.

Esercizio 2.11 Dimostrare che se consideriamo grammatiche che ammettono sia produzioni lineari destre che produzioni lineari sinistre possiamo generare tutti e soli i linguaggi lineari.

2.4 Forma Normale di Backus e diagrammi sintattici

Nelle sezioni precedenti abbiamo visto che le grammatiche formali sono strumenti di tipo generativo per la definizione di linguaggi. In particolare esse possono essere utilizzate per la definizione di linguaggi di programmazione, consentendo così di caratterizzare un linguaggio di programmazione come l'insieme di tutte le stringhe (programmi) derivabili dalla grammatica.

Per la definizione sintattica dei linguaggi di programmazione vengono adottate grammatiche non contestuali, tradizionalmente rappresentate mediante una notazione specifica, particolarmente suggestiva, denominata *Forma Normale di Backus* (*Backus Normal Form*, BNF, detta anche *Backus-Naur Form*).

La BNF è una notazione per grammatiche context free resa più espressiva e succinta mediante i seguenti accorgimenti (alcuni dei quali vagamente ispirati alle espressioni regolari):

1. I simboli non terminali sono sempre costituiti da stringhe che denominano delle categorie sintattiche racchiuse tra parentesi acute $<\ldots>$ (ad es.: $<$`espressione`$>$);

2. Il segno di produzione ($\longrightarrow$) viene sostituito dal simbolo $::=$ in modo da

non confonderlo con i simboli $\longrightarrow$, $=$, e $:=$, che sono simboli terminali usati in vari linguaggi di programmazione.

3. Le parentesi graffe $\{\ldots\}$ vengono impiegate per indicare l'iterazione illimitata ($0, 1, 2, \ldots, n, \ldots$ volte). Analogamente con $\{\ldots\}^n$ si può indicare l'iterazione per un numero di volte pari al più ad n. Ad esempio, le produzioni:

   $$\mathbf{A} ::= \mathbf{bA} \mid \mathbf{a}$$

   possono essere così riscritte:

   $$\mathbf{A} ::= \{\mathbf{b}\}\ \mathbf{a}$$

   e le produzioni:

   $$\mathbf{A} ::= \mathbf{x} \mid \mathbf{xy} \mid \mathbf{xyy} \mid \mathbf{xyyy} \mid \mathbf{xyyyy} \mid \mathbf{xyyyyy}$$

   possono essere così riscritte:

   $$\mathbf{A} ::= \mathbf{x}\ \{\mathbf{y}\}^5$$

4. Le parentesi quadre $[\ldots]$ vengono utilizzate per indicare l'opzionalità (possibile assenza di una parte di stringa). Ad esempio, le produzioni:

   $$\mathbf{A} ::= \mathbf{xy} \mid \mathbf{y}$$

   possono essere così riscritte:

   $$\mathbf{A} ::= [\mathbf{x}]\ \mathbf{y}$$

5. Le parentesi tonde $(\ldots)$ vengono utilizzate per indicare la fattorizzazione, vale a dire la messa in comune di una sottoespressione. Ad esempio, le produzioni:

   $$\mathbf{A} ::= \mathbf{xu} \mid \mathbf{xv} \mid \mathbf{xy}$$

   possono essere così riscritte:

   $$\mathbf{A} ::= \mathbf{x}\ (\mathbf{u} \mid \mathbf{v} \mid \mathbf{y})$$

Esempio 2.24 Possiamo esprimere la struttura di un identificatore nel linguaggio Pascal, definito informalmente come una sequenza di al più $n > 0$ caratteri alfanumerici di cui il primo necessariamente alfabetico, tramite la BNF nel seguente modo:

<identificatore> ::= <alfabetico> $\{<$ alfanumerico $>\}^{n-1}$

<alfanumerico> ::= <alfabetico> | <cifra>

<alfabetico> ::= **A** | **B** | $\cdots$ | **Z** | **a** | **b** | $\cdots$ | **z**

<cifra> ::= **0** | **1** | $\cdots$ | **9**

Esempio 2.25 Riportiamo di seguito un frammento della BNF per il linguaggio Pascal:

```
<programma> ::= <intestazione> ; <blocco>.

<intestazione> ::= program <identificatore> [ <lista par.> ]

<lista par.> ::= <lista identif.>

<lista identif.> ::= <identificatore> {, <identificatore> }

<blocco> ::= [< sezione dichiarazione etichette >]
             [< sezione definizione costanti >]
             [< sezione definizione tipi >]
             [< sezione dichiarazione variabili >]
             [< sezione dichiarazione procedure e funzioni >]
             [< sezione istruzioni >]

...

<sezione definizione tipi> ::= type <identificatore>=<tipo>
                               {; < identificatore > = < tipo >}

...

<proposizione-while> ::= while <predicato> do <proposizione>

<proposizione-if> ::= if <predicato> then <proposizione>
                                     else <proposizione>

...

<predicato> ::= <espressione-booleana>

<proposizione> ::= <proposizione-semplice> |
                   <proposizione-composta>

<proposizione composta> ::= begin <proposizione>
                            {;  < proposizione >} end

<proposizione-semplice> ::= <assegnazione> |
                            <proposizione-if> |
```

<proposizione-while>

<assegnazione> ::= <identificatore> := <espressione>

Un'altra notazione molto utilizzata per descrivere i linguaggi di programmazione è costituita dai *diagrammi sintattici*, anch'essi riconducibili ad un meccanismo di tipo generativo, ma influenzati dal concetto di automa (vedi Sezione 3.1).

Tali diagrammi sono sostanzialmente dei grafi orientati con un ingresso ed un'uscita, e i cui nodi sono collegati da archi etichettati con uno o più caratteri terminali o non terminali. I caratteri terminali vengono usualmente inclusi in un cerchio o in una forma di tipo ellittico e quelli non terminali vengono inclusi in un rettangolo: ad ogni simbolo non terminale corrisponde a sua volta un diagramma sintattico. Quando un arco di un diagramma sintattico è etichettato con un simbolo non terminale si deve immaginare che tale arco sia sostituito dal diagramma sintattico corrispondente al non terminale stesso. Ogni cammino in un diagramma sintattico definisce una forma di frase.

Esempio 2.26 Il diagramma sintattico dato in Figura 2.3 equivale alla grammatica:

$$\begin{array}{lcl} A & \longrightarrow & aABc \mid aBc \\ B & \longrightarrow & bB \mid b \end{array}$$

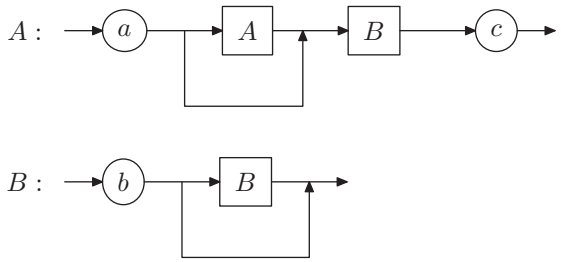


FIGURA 2.3 Diagramma sintattico di un semplice linguaggio.

Esercizio 2.12 Fornire la forma di Backus e i diagrammi sintattici per il linguaggio delle espressioni aritmetiche (vedi Esempio 2.11).

2.5 Accettazione e riconoscimento di linguaggi

Le grammatiche formali introdotte in questa sezione costituiscono uno strumento generativo per la rappresentazione e la classificazione di linguaggi. Esse

non consentono tuttavia di impostare in modo algoritmico il problema del riconoscimento di un linguaggio, e cioè il problema di decidere, dato un linguaggio L e una stringa x, se $x \in L$. Tale problema riveste importanza fondamentale: si ricorda, a tal proposito, che il problema di stabilire se un programma scritto in un linguaggio di programmazione sia sintatticamente corretto può essere formalizzato come il problema di decidere se una particolare stringa (appunto, il programma sorgente) appartenga o meno a un determinato linguaggio (ad esempio, il Pascal, visto come l'insieme dei programmi Pascal sintatticamente corretti), definito per mezzo di una opportuna grammatica (ad esempio, la definizione del Pascal in BNF).

Notiamo inoltre che l'importanza di affrontare i problemi di riconoscimento di linguaggi deriva anche dal fatto che ogni problema decisionale, cioè un qualunque problema consistente nel determinare se una certa proprietà sia verificata o meno sull'istanza fornita in input, può essere visto come problema di riconoscimento di un particolare linguaggio (ad esempio, il problema di decidere se un dato numero intero sia un numero primo può essere espresso come il problema di riconoscere tutte le stringhe di 0 e 1 che costituiscono la rappresentazione binaria di un numero primo).

2.5.1 Automi e computazioni

Il concetto fondamentale per quanto riguarda il riconoscimento di linguaggi è il concetto di *automa*, inteso come dispositivo astratto che, data un stringa x fornitagli in input, può eseguire una *computazione* ed eventualmente, se la computazione termina, restituisce, secondo una qualche modalità, un valore che, nel caso del problema del riconoscimento, sarà booleano. In generale, la

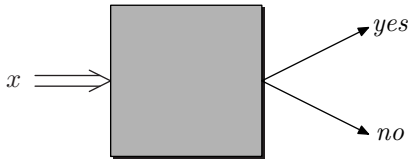


FIGURA 2.4 Schema generale di automa.

struttura di un automa comprende un dispositivo interno, che ad ogni istante assume uno *stato* in un possibile insieme predefinito: lo stato rappresenta essenzialmente l'informazione associata al funzionamento interno.

Inoltre, un automa può utilizzare uno o più dispositivi di memoria, sui quali è possibile memorizzare delle informazioni, sotto forma di stringhe di caratteri da alfabeti anch'essi predefiniti. In genere si assume che tali dispositivi di memoria siano *nastri*, sequenze cioè di celle, ognuna delle quali può contenere un carattere: uno tra tali nastri contiene inizialmente l'input fornito

all'automa. I caratteri vengono letti o scritti per mezzo di *testine* che possono muoversi lungo i nastri, posizionandosi sulle diverse celle (vedi Figura 2.5). Come si vedrà nei capitoli successivi, diversi tipi di automi possono essere definiti facendo ipotesi diverse sulle capacità di movimento, lettura e scrittura delle testine sui vari nastri. Il funzionamento di un automa è definito rispetto

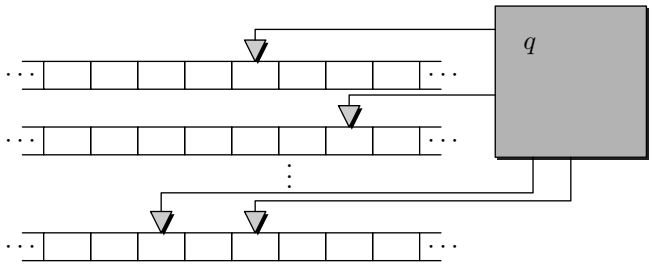


FIGURA 2.5 Schema particolareggiato di automa.

---

ai due concetti di *configurazione* e di *funzione di transizione*.

Una configurazione rappresenta l'insieme delle informazioni che determinano, in un certo istante, il comportamento (o l'insieme di comportamenti, nel caso degli automi non deterministici introdotti nella Sezione 2.5.2) futuro dell'automa: se si considera nuovamente lo schema generale di automa in Figura 2.5, ne risulta che una configurazione è composta dalle seguenti informazioni:

1. stato interno dell'automa;
2. contenuto di tutti i nastri di memoria;
3. posizione di tutte le testine sui nastri.

La definizione di un automa permette, come vedremo, di associare ad una stringa in input una ed una sola configurazione, detta *configurazione iniziale*, che rappresenta la situazione complessiva in cui si trova l'automa all'inizio, nel momento in cui la stringa di input gli viene sottoposta.

Esempio 2.27 In Sezione 3.1 verrà introdotto un particolare tipo di automa, detto *Automa a stati finiti*. Gli automi di questo tipo non possono memorizzare informazioni, a parte la stringa di input; inoltre, il nastro contenente la stringa di input può essere scandito dalla relativa testina una sola volta, dall'inizio alla fine della stringa stessa. Per tali automi una configurazione è rappresentata dalle seguenti componenti:

1. la stringa x di input;
2. la posizione del carattere di x che viene attualmente letto;

3. lo stato interno dell'automa.

In tali automi, un particolare stato q_0 è specificato essere lo *stato iniziale*. La configurazione iniziale associata alla stringa di input x è allora data da:

1. la stringa x;
2. la prima posizione della stringa;
3. lo stato q_0.

La *funzione di transizione*, che è parte della definizione della struttura dell'automa, induce una *relazione di transizione* tra configurazioni, che associa ad una configurazione un'altra (o più di una) *configurazione successiva*.[2] L'applicazione della funzione di transizione ad una configurazione si dice *transizione* o *mossa* o *passo computazionale* dell'automa. In generale, per ogni automa $\mathcal{A}$, date due configurazioni c_i, c_j di $\mathcal{A}$, useremo nel seguito la notazione $c_i \underset{\mathcal{A}}{\vdash\!\!\!-\!\!\!-} c_j$ per indicare che c_i e c_j sono correlate dalla relazione di transizione, vale a dire che c_j deriva da c_i per effetto dell'applicazione della funzione di transizione di $\mathcal{A}$.[3]

Esempio 2.28 La funzione di transizione, nel caso di automi a stati finiti, associa ad una coppia "stato attuale - carattere letto" un nuovo stato. Tale funzione consente di associare ad una configurazione la configurazione successiva nel modo seguente. Se la funzione associa lo stato q_j alla coppia composta dallo stato q_i e dal carattere a allora, data una configurazione composta da:

1. la stringa x;
2. la posizione i-esima;
3. lo stato q_i;

dove l'i-esimo carattere di x è il carattere a, la configurazione successiva sarà composta da:

1. la stringa x;
2. la posizione $i+1$-esima;
3. lo stato q_j.

Alcune configurazioni, aventi strutture particolari e dipendenti dalla struttura dell'automa, sono inoltre considerate come *configurazioni di accettazione*. Tutte le altre configurazioni sono definite come *Configurazioni di non accettazione*, o di *rifiuto*.

[2] Come vedremo nei capitoli successivi, la funzione di transizione, pur consentendo di associare configurazioni a configurazioni, non è definita sull'insieme delle possibili configurazioni, ma su domini e codomini che rappresentano parti di configurazioni, quelle che effettivamente determinano e sono determinate dalla transizione.

[3] Se l'automa cui si fa riferimento è chiaro dal contesto potremo anche usare la notazione semplificata $c_i \vdash\!\!\!-\!\!\!- c_j$.

Esempio 2.29 Nel caso di automi a stati finiti viene definito, dato un automa, un insieme F di stati, detti *finali*. Una configurazione di accettazione corrisponderà allora alla situazione in cui l'automa, letta tutta la stringa di input, si trova in uno stato finale. Ne deriva quindi che una configurazione di accettazione ha la struttura:

1. la stringa x;
2. la prima posizione successiva alla stringa;
3. uno stato finale $q \in F$.

Una configurazione non di accettazione corrisponderà, al contrario, alla situazione in cui l'automa non ha letto tutta la stringa di input oppure, se l'ha letta, non si trova in uno stato finale. Una configurazione di rifiuto ha quindi la struttura

1. la stringa x
2. posizione su un carattere della stringa
3. uno stato qualunque $q \in Q$,

o la struttura

1. la stringa x
2. la prima posizione successiva alla stringa
3. uno stato non finale $q \notin F$.

Un automa esegue una computazione applicando iterativamente, ad ogni istante, la propria funzione di transizione alla configurazione attuale, a partire dalla configurazione iniziale.

Possiamo quindi vedere una computazione eseguita da un automa $\mathcal{A}$ a partire da una configurazione iniziale c_0 come una sequenza di configurazioni $c_0, c_1, c_2, \ldots$ tale che, per $i = 0, 1, \ldots$, si ha $c_i \underset{\mathcal{A}}{\vdash} c_{i+1}$. Indicheremo anche nel seguito con la notazione $\overset{*}{\underset{\mathcal{A}}{\vdash}}$ la chiusura transitiva e riflessiva della relazione $\underset{\mathcal{A}}{\vdash}$. È facile allora rendersi conto che, dato un automa $\mathcal{A}$ e due configurazioni c_i, c_j di $\mathcal{A}$, si ha $c_i \overset{*}{\underset{\mathcal{A}}{\vdash}} c_j$ se e solo se esiste una computazione che porta $\mathcal{A}$ da c_i a c_j.

Se la sequenza di configurazioni $c_0, c_1, c_2, \ldots, c_n$ ha lunghezza finita e se è massimale, nel senso che non esiste nessuna configurazione c tale che $c_n \underset{\mathcal{A}}{\vdash} c$, allora diciamo che la computazione *termina*: in tal caso, diciamo che essa è una *computazione di accettazione* se termina in una configurazione di accettazione, mentre, nel caso contrario in cui termini in una configurazione non di accettazione, diremo che è una *computazione di rifiuto*.

2.5.2 *Automi deterministici e non deterministici*

In questa sezione introdurremo due concetti importanti, relativi alle struttura degli automi. Tali concetti, che saranno successivamente ripresi e ridiscussi in modo estensivo per i vari tipi di automi che saranno considerati, sono quelli di *determinismo* e *non determinismo*, ed hanno una grande rilevanza per quanto riguarda la classificazione dei linguaggi.

Al fine di introdurre questi concetti, osserviamo che, per quanto esposto nella sezione precedente, possiamo considerare un automa come un dispositivo che, data una stringa di input, associa ad essa (una o più) computazioni.

Un automa, in particolare, è detto *deterministico* se ad ogni stringa di input associa una sola computazione, e quindi una singola sequenza di configurazioni (vedi Figura 2.6). Possiamo anche dire che un automa deterministico, data una stringa di input, può eseguire una sola computazione: se tale computazione termina in una configurazione di accettazione, allora la stringa viene accettata.

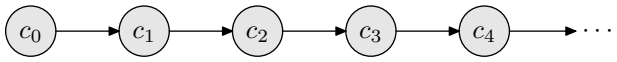


FIGURA 2.6 Computazione di un automa deterministico.

Esercizio 2.13 Dimostrare che condizione necessaria e sufficiente affinché un automa sia deterministico è che la relativa funzione di transizione sia una funzione dall'insieme delle possibili configurazioni in sé stesso.

Dato un automa deterministico $\mathcal{A}$ e data una stringa x di input ad $\mathcal{A}$, se indichiamo con $c_0(x)$ la configurazione iniziale di $\mathcal{A}$ corrispondente alla stringa x, avremo che $\mathcal{A}$ accetta x se e solo se esiste una configurazione di accettazione c di $\mathcal{A}$ per la quale $c_0(x) \overset{*}{\underset{\mathcal{A}}{\vdash}} c$. Il linguaggio accettato da $\mathcal{A}$ sarà allora l'insieme $L(\mathcal{A})$ di tutte le stringhe x accettate da $\mathcal{A}$.

Nel caso in cui si verifichi inoltre che per ogni stringa la computazione eseguita dall'automa $\mathcal{A}$ sulla stringa stessa termina, e quindi se ogni stringa viene o accettata o rifiutata, allora diciamo che il linguaggio $L(\mathcal{A})$ è riconosciuto (o deciso) da $\mathcal{A}$.

Un automa è, al contrario, detto *non deterministico* se esso associa ad ogni stringa di input un numero qualunque, in generale maggiore di uno, di computazioni.[4] Automi di questo tipo corrispondono al caso generale in cui la funzione di transizione associa a qualche configurazione c più di una configurazione successiva: in tal modo, per ogni computazione che conduce alla

[4]Si osservi che un automa deterministico risulta quindi essere un caso particolare di automa non deterministico.

configurazione c sono definite continuazioni diverse, che definiscono diverse computazioni.

Indichiamo con il termine *grado di non determinismo* di un automa il massimo numero di configurazioni che la funzione di transizione associa ad una configurazione. Ciò evidentemente corrisponde al massimo numero di configurazioni differenti che si possono raggiungere in un singolo passo di computazione a partire da una stessa configurazione.

Esempio 2.30 In Figura 2.7 viene riportato il caso in cui, per effetto della funzione di transizione che associa alla configurazione c_3 la coppia di configurazioni c_4, c_5, la computazione c_0, c_1, c_2, c_3 ha due distinte possibili continuazioni, dando luogo quindi alle due computazioni $c_0, c_1, c_2, c_3, c_4, c_5, \ldots$ e $c_0, c_1, c_2, c_3, c_6, c_7, \ldots$.

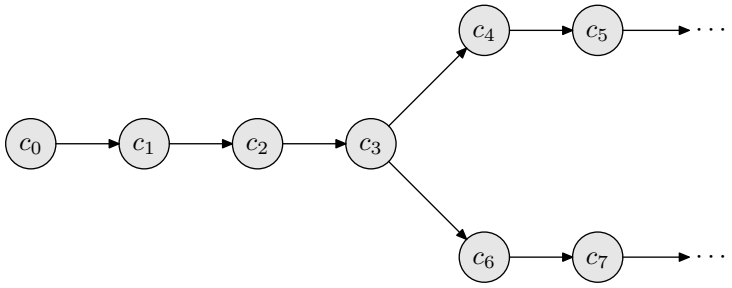


FIGURA 2.7 Continuazioni diverse di una stessa computazione in un automa non deterministico.

Esercizio 2.14 Dimostrare che due qualunque computazioni associate da un automa non deterministico ad una stessa stringa di input devono necessariamente avere un prefisso uguale.

Un diverso modo di introdurre il non determinismo, che citiamo qui ma non considereremo nel seguito, consiste nel prevedere la presenza di ε-transizioni, transizioni cioè che un automa può eseguire senza leggere alcun carattere in input. Ci si può rendere conto facilmente che il fatto che da una configurazione c una automa possa passare o meno in un'altra configurazione c' senza leggere caratteri in input introduce due diverse possibili continuazioni della computazione attuale, l'una da c e l'altra da c'.

Esempio 2.31 Si consideri la situazione in Figura 2.8, in cui si assume che la configurazione attuale sia c_0 e che il prossimo carattere in input sia il carattere a. Si assuma inoltre che, per definizione della funzione di transizione dell'automa, la lettura di a fa passare dalla configurazione c_0 alla configurazione c_3, e che è però definita una possibile ε-transizione da c_0 a c_1, da cui la lettura di a fa passare l'automa nella configurazione c_2.

Come si può osservare ciò dà luogo a due diverse continuazioni: la prima attraverso c_3, nel caso in cui la ε-transizione non viene eseguita, la seconda attraverso c_1 e c_2, nel caso in cui essa venga eseguita.

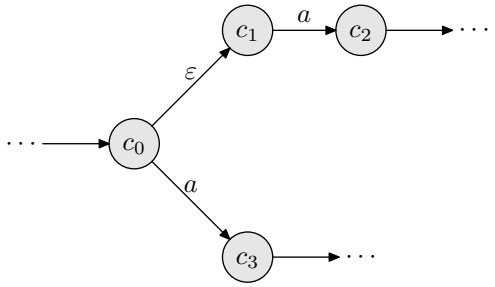


FIGURA 2.8 Esempio di non determinismo introdotto da ε-transizione.

Dalle osservazioni precedenti, possiamo osservare che, mentre un automa deterministico associa alla stringa di input una computazione avente struttura lineare, un automa non deterministico associa ad essa una struttura più complessa, ad albero, con i nodi che rappresentano configurazioni dell'automa (con la radice associata alla configurazione iniziale) e gli archi che indicano la possibile transizione tra configurazioni. In questa struttura, detta *albero di computazione*, ogni computazione corrisponde ad un cammino avente origine dalla radice stessa (vedi Figura 2.9).

L'automa non deterministico può quindi eseguire una qualunque computazione, associata ad una sequenza di "scelte su quale transizione applicare in corrispondenza ad ogni applicazione della funzione di transizione. L'automa accetta la stringa di input se, tra tutte le computazioni possibili, ne esiste almeno una di accettazione, vale a dire se esiste, nell'albero di computazione, un cammino dalla radice ad un nodo rappresentante una configurazione di accettazione.

Si osservi la asimmetria tra accettazione e rifiuto di una stringa, introdotta dal non determinismo: infatti, nel caso deterministico la stringa viene accettata se la singola computazione definita è di accettazione, e rifiutata se essa termina in una configurazione non di accettazione. Al contrario, nel caso non deterministico la stringa viene accettata se *una qualunque* delle computazioni definite è di accettazione, mentre non lo viene se *tutte* le possibili computazioni che terminano non sono di accettazione.

Un modo alternativo, a cui faremo a volte riferimento nel seguito, di vedere il comportamento di un automa non deterministico, consiste nel considerare che l'automa esegua una sola *computazione non deterministica*, per la quale, ad ogni passo, assume non una sola, ma un insieme di configurazioni, transitando,

ad ogni passo, non da configurazione a configurazione ma da un insieme di configurazioni ad un insieme di configurazioni.

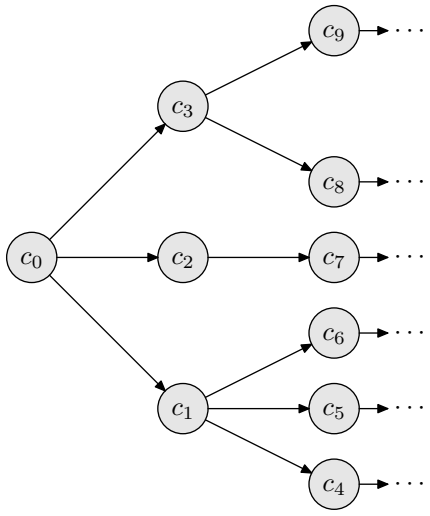


FIGURA 2.9 Albero di computazione di un automa non deterministico.

Esempio 2.32 L'albero di computazione riportato in Figura 2.9 definisce le singole computazioni aventi come prefissi le sequenze di configurazioni $c_0, c_1, c_4, \ldots$; $c_0, c_1, c_5, \ldots$; $c_0, c_1, c_6, \ldots$; $c_0, c_2, c_7, \ldots$; $c_0, c_3, c_8, \ldots$; $c_0, c_3, c_9, \ldots$ corrispondenti per l'appunto ai cammini con origine nella radice c_0 dell'albero. Si osservi che il grado di non determinismo di un automa è pari al massimo grado di uscita per un nodo dell'albero di computazione relativo all'automa stesso. Il massimo grado di non determinismo nell'albero in Figura 2.10 è pari a 3 (configurazioni c_0 e c_1): ne deriva quindi che il grado di non determinismo dell'automa a cui si riferisce l'albero di computazione è almeno pari a 3.

Il concetto di non determinismo, che qui incontriamo per la prima volta, non deve essere confuso con il non determinismo presente nel mondo reale. Nella realtà, in varie circostanze, accade che un osservatore, in base alla conoscenza dello stato attuale di un sistema e di eventuali stimoli, non sia in grado di determinare univocamente in quale stato il sistema si troverà. Ciò può verificarsi o per caratteristiche intrinseche del sistema, come accade nel caso di fenomeni quantistici, oppure perché l'osservatore non ha un modello adeguato a rappresentare la complessità del fenomeno osservato e quindi a tener conto di tutti i diversi fattori che influenzano l'evoluzione del sistema, come accade ad esempio nei sistemi termodinamici o, più banalmente, nella complessità della vita quotidiana. In tali casi si usa supplire alle carenze di conoscenza del sistema facendo ricorso al concetto di probabilità.

Il non determinismo che abbiamo qui introdotto, e che introdurremo nel corso della trattazione seguente nei vari modelli di calcolo che considereremo per modificarne il potere computazionale, ha caratteristiche del tutto diverse. Nel nostro caso il non determinismo è sostanzialmente un artificio matematico che ci consente di rappresentare un calcolo, anziché come una traiettoria in uno spazio di stati, come un *albero di traiettorie*, analogamente a come possono essere previste evoluzioni diverse di una vicenda in una narrazione a finale multiplo. In particolare, come visto, ciò che ci interessa è poter definire un concetto di accettazione legato al fatto che almeno uno dei rami dell'albero conduca ad uno stato finale.

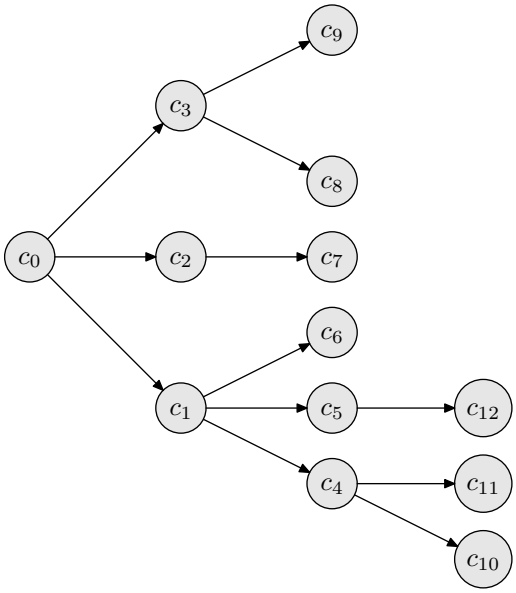


FIGURA 2.10 Esempio di computazione di un automa non deterministico.

2.5.3 Automi e classi di linguaggi

Una prima osservazione riguardante il rapporto fra generazione di linguaggi e problemi decisionali è già stata fatta nella Sezione 1.3.4 dove, sulla base di semplici considerazioni di cardinalità, abbiamo mostrato che esistono infiniti linguaggi per cui non esistono algoritmi (né automi) di riconoscimento.[5] Una

[5]Si noti che le considerazioni relative alla numerabilità dell'insieme degli algoritmi si applicano anche agli automi. In effetti, un automa può essere visto come un particolare tipo di algoritmo.

considerazione analoga può essere fatta in relazione alle grammatiche: poiché una grammatica può essere rappresentata essa stessa come una stringa finita di caratteri, l'insieme delle grammatiche di tipo 0 è un insieme numerabile. Poiché d'altra parte l'insieme dei linguaggi ha la cardinalità del continuo esistono infiniti linguaggi a cui non corrisponde alcuna grammatica.

A tal proposito, secondo una terminologia consolidata, un problema decisionale (o un linguaggio) viene detto decidibile se esiste un algoritmo che risolve ogni istanza del problema. In alcune situazioni, come vedremo meglio nei prossimi capitoli, non esistono algoritmi di decisione ma solo algoritmi in grado di riconoscere correttamente un'istanza positiva del problema. In queste situazioni si parla di semidecidibilità.

Definizione 2.10 *Un problema decisionale (o un linguaggio) è detto* decidibile *se esiste un algoritmo (in particolare, un automa) che per ogni istanza del problema risponde correttamente* VERO *oppure* FALSO. *In caso contrario il problema è detto* indecidibile.

Definizione 2.11 *Un problema decisionale (o un linguaggio) è detto* semidecidibile *se esiste un algoritmo (in particolare, un automa) che per tutte e sole le istanze positive del problema risponde correttamente* VERO.

Possiamo fin da ora anticipare che, come vedremo nei capitoli successivi i linguaggi di tipo 1, 2 e 3 sono decidibili, mentre quelli di tipo 0 sono semidecidibili. Nei capitoli successivi infatti introdurremo vari tipi di automi che possono essere utilizzati per risolvere problemi di decisione relativi alle diverse classi di linguaggi.

In particolare vedremo che:

- Per i linguaggi di tipo 3 esistono dispositivi di riconoscimento che operano con memoria costante e tempo lineare (*Automi a stati finiti*).
- Per i linguaggi strettamente di tipo 2, il riconoscimento può avvenire sempre in tempo lineare ma con dispositivi di tipo non deterministico[6] dotati di una memoria a pila (*Automi a pila non deterministici*).
- Per i linguaggi strettamente di tipo 1, l'esigenza di tempo e di memoria è ancora maggiore. Per essi si può mostrare che il riconoscimento può essere effettuato con una macchina non deterministica che fa uso di una quantità di memoria che cresce linearmente con la lunghezza della stringa da esaminare (*Macchina di Turing non deterministica "linear bounded"*, o *Automa lineare*).
- Infine, per i linguaggi strettamente di tipo 0, si farà riferimento a un dispositivo di calcolo costituito da un automa che opera con quantità di tempo e di memoria potenzialmente illimitate (*Macchina di Turing*).

[6] Vedremo nel seguito (Sezione 2.5.2) cosa tale termine stia a significare.

Nei Capitoli 3 e 4 ci soffermeremo sui metodi di riconoscimento e sulle proprietà delle due classi di linguaggi che hanno maggiore attinenza con i linguaggi di programmazione, cioè i linguaggi di tipo 3 e di tipo 2.

Alle macchine di Turing ed alla caratterizzazione del loro potere computazionale sarà dedicato il Capitolo 5 e, in quel contesto, sarà ripreso anche lo studio delle proprietà dei linguaggi di tipo 0 e di tipo 1.

Capitolo 3

Linguaggi regolari

I linguaggi di tipo 3 costituiscono una classe particolarmente interessante per diversi motivi. Innanzi tutto, i componenti sintattici più elementari di un linguaggio di programmazione — identificatori, costanti, parole chiave, ecc. — sono generalmente definibili con grammatiche di tipo 3. In secondo luogo, nonostante l'estrema semplicità delle grammatiche di tale tipo, si può mostrare che la classe di linguaggi che esse generano gode di numerose proprietà algebriche. In particolare, tali proprietà consentono di associare ad ogni linguaggio di tipo 3 un'espressione regolare che ne descrive le stringhe.

Nel corso di questa sezione introdurremo innanzi tutto i dispositivi di riconoscimento per i linguaggi di tipo 3, detti automi a stati finiti, e successivamente discuteremo le principali proprietà della classe dei linguaggi regolari.

3.1 Automi a stati finiti

La tipologia più semplice di dispositivo per il riconoscimento di linguaggi che prenderemo in esame, è costituita dai cosiddetti *Automi a stati finiti* (ASF). Questi automi, già introdotti in modo informale in Sezione 2.5, si possono pensare come dispositivi che, mediante una testina, leggono la stringa di input scritta su un nastro e la elaborano facendo uso di un elementare meccanismo di calcolo e di una memoria finita e limitata. L'esame della stringa avviene un carattere alla volta, mediante una sequenza di passi di computazione, ognuno dei quali comporta lo spostamento della testina sul carattere successivo e l'aggiornamento dello stato della memoria (Figura 3.1). In questa sezione caratterizzeremo, in particolare, gli *Automi a stati finiti deterministici*: nella successiva Sezione 3.2 tratteremo il caso più generale degli *Automi a stati finiti non deterministici*.

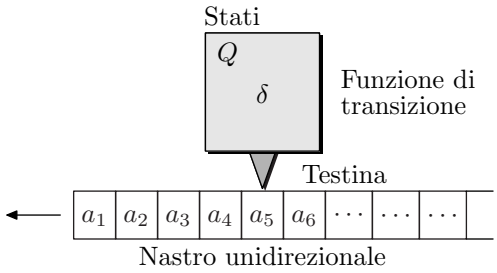


FIGURA 3.1 Schema di automa a stati finiti.

Definizione 3.1 *Un* automa a stati finiti deterministico *(ASFD) è una quintupla* $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$*, dove* $\Sigma = \{a_1, \ldots, a_n\}$ *è l'*alfabeto *di input,* $Q = \{q_0, \ldots, q_m\}$ *è un insieme finito e non vuoto di* stati*,* $F \subseteq Q$ *è un insieme di* stati finali*,* $q_0 \in Q$ *è lo* stato iniziale *e* $\delta : Q \times \Sigma \mapsto Q$ *è la* funzione (totale) di transizione *che ad ogni coppia* $\langle$*stato*, *carattere in input*$\rangle$ *associa uno stato successivo.*

Il comportamento dell'automa così definito è caratterizzato in particolare dalla funzione di transizione. Essa è una funzione con dominio finito, per cui può venire rappresentata mediante una tabella, detta *tabella di transizione* (o, equivalentemente, *matrice di transizione*), alle cui righe vengono associati gli stati, alle colonne i caratteri in input, ed i cui elementi rappresentano il risultato dell'applicazione della δ allo stato identificato dalla riga ed al carattere associato alla colonna della tabella.

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Tabella 3.1 Esempio di tabella di transizione di un ASFD.

Un'altra rappresentazione, molto più evocativa, è costituita dal cosiddetto *diagramma degli stati*, (o *grafo di transizione*) in cui l'automa è rappresentato mediante un grafo orientato: i nodi rappresentano gli stati, mentre gli archi rappresentano le transizioni, per cui sono etichettati con il carattere la cui lettura determina la transizione. Gli stati finali sono rappresentati da nodi con un doppio circolo, mentre quello iniziale è individuato tramite una freccia

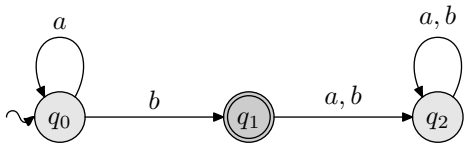


FIGURA 3.2 Diagramma degli stati dell'ASFD di Tabella 3.1.

entrante. Essenzialmente, un ASFD esegue una computazione nel modo seguente: a partire dalla situazione in cui lo stato attuale è q_0, l'automa ad ogni passo legge il carattere successivo della stringa in input ed applica la funzione δ alla coppia formata dallo stato attuale e da tale carattere per determinare lo stato successivo. Terminata la lettura della stringa, essa viene accettata se lo stato attuale è uno stato finale, rifiutata altrimenti.[1] Al fine di formalizzare meglio il modo di operare di un ASFD, possiamo osservare, ricordando quanto detto nella Sezione 2.5, che il comportamento futuro di un automa a stati finiti deterministico dipende esclusivamente dallo stato attuale e dalla stringa da leggere. Da ciò, e da quanto esposto sopra, conseguono le seguenti definizioni.

Definizione 3.2 *Dato un automa a stati finiti* $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$*, una configurazione di* $\mathcal{A}$ *è una coppia* (q, x)*, con* $q \in Q$ *e* $x \in \Sigma^*$*.*

Definizione 3.3 *Una configurazione* $\langle q, x \rangle$*,* $q \in Q$ *ed* $x \in \Sigma^*$*, di* $\mathcal{A}$*, è detta:*

- iniziale *se* $q = q_0$*;*
- finale *se* $x = \varepsilon$*;*
- accettante *se* $x = \varepsilon$ *e* $q \in F$*.*

La funzione di transizione permette di definire la relazione di transizione $\vdash\!\!\!-$ tra configurazioni, che associa ad una configurazione la configurazione successiva, nel modo seguente.

Definizione 3.4 *Dato un ASFD* $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ *e due configurazioni* (q, x) *e* (q', y) *di* $\mathcal{A}$*, avremo che* $(q, x) \underset{\mathcal{A}}{\vdash\!\!\!-} (q', y)$ *se e solo se valgono le due condizioni:*

1. *esiste* $a \in \Sigma$ *tale che* $x = ay$*;*
2. $\delta(q, a) = q'$*.*

[1] Si noti che un ASFD termina sempre le sue computazioni.

A questo punto possiamo dire che, dato un automa a stati finiti deterministico $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, una stringa $x \in \Sigma^*$ è accettata da $\mathcal{A}$ se e solo se $(q_0, x) \overset{*}{\underset{\mathcal{A}}{\vdash}} (q, \varepsilon)$, con $q \in F$, e possiamo definire il linguaggio riconosciuto[2] da $\mathcal{A}$ come

$$L(\mathcal{A}) = \left\{ x \in \Sigma^* \mid (q_0, x) \overset{*}{\underset{\mathcal{A}}{\vdash}} (q, \varepsilon), q \in F \right\}.$$

Esempio 3.1 La stringa *aab* è accettata dall'automa a stati finiti deterministico descritto in Figura 3.2. Infatti, a partire dalla configurazione iniziale (q_0, aab) l'automa raggiunge la configurazione di accettazione (q_1, ε) per mezzo della computazione $(q_0, aab) \vdash\!\!\!- (q_0, ab) \vdash\!\!\!- (q_0, b) \vdash\!\!\!- (q_1, \varepsilon)$.

Un modo alternativo ed ampiamente utilizzato di descrivere il comportamento di un automa a stati finiti deterministico e di definire il linguaggio da esso riconosciuto utilizza una estensione della funzione di transizione da caratteri a stringhe.

Definizione 3.5 *La* funzione di transizione estesa *di un automa a stati finiti deterministico* $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ *è la funzione* $\overline{\delta} : Q \times \Sigma^* \mapsto Q$*, definita nel seguente modo:*

$$\begin{aligned} \overline{\delta}(q, \varepsilon) &= q \\ \overline{\delta}(q, xa) &= \delta\left(\overline{\delta}(q, x), a\right), \end{aligned}$$

dove $a \in \Sigma$*,* $x \in \Sigma^*$*.*

Dalla definizione precedente, avremo quindi che, dato uno stato q ed una stringa di input $x \in \Sigma^*$, lo stato q' è tale che $q' = \overline{\delta}(q, x)$ se e solo se la computazione eseguita dall'automa a partire da q ed in conseguenza della lettura della stringa x conduce l'automa stesso nello stato q'. Più formalmente, possiamo anche dire che $q' = \overline{\delta}(q, x)$ se e solo se esiste $y \in \Sigma^*$ tale che $(q, xy) \overset{*}{\vdash\!\!\!-} (q', y)$. Di conseguenza, avremo che una stringa $x \in \Sigma^*$ è accettata da un ASFD $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ se e solo se $\overline{\delta}(q_0, x) \in F$.

È allora possibile introdurre la seguente definizione di linguaggio riconosciuto da un ASFD.

Definizione 3.6 *Il* linguaggio riconosciuto *da un automa a stati finiti deterministico* $\mathcal{A}$ *è l'insieme*

$$L(\mathcal{A}) = \left\{ x \in \Sigma^* \mid \overline{\delta}(q_0, x) \in F \right\}.$$

[2] Si osservi che, dato che ogni computazione di un ASFD termina, ogni automa a stati finiti deterministico riconosce un linguaggio.

Esempio 3.2 Al fine di verificare che la stringa aab è accettata dall'ASFD di Figura 3.2, deriviamo il valore di $\delta(q_0, aab)$ nel modo seguente:

$$\overline{\delta}(q_0, aab) = \delta\left(\overline{\delta}(q_0, aa), b\right) = \delta\left(\delta\left(\overline{\delta}(q_0, a), a\right), b\right) = \delta\left(\delta\left(\delta\left(\overline{\delta}(q_0, \varepsilon), a\right), a\right), b\right) =$$

$$= \delta\left(\delta\left(\delta\left(q_0, a\right), a\right), b\right) = \delta\left(\delta\left(q_0, a\right), b\right) = \delta\left(q_0, b\right) = q_1.$$

Esercizio 3.1 Si consideri il linguaggio $L = \{a^n b \mid n \geq 0\}$. Come sappiamo, questo linguaggio è generato dalla grammatica $\mathcal{G}$ le cui produzioni sono:

$$S \longrightarrow aS \mid b.$$

Si dimostri che il linguaggio L è il linguaggio riconosciuto dall'automa di Figura 3.2.

Esempio 3.3 Si consideri il linguaggio delle parole sull'alfabeto $\{a, b\}$ che contengono un numero pari di a o un numero pari di b. Un automa che riconosce tale linguaggio è l'automa $\mathcal{A} = \langle\{a, b\}, \{q_0, q_1, q_2, q_3\}, \delta, q_0, \{q_0, q_1, q_2\}\rangle$, dove la funzione di transizione è rappresentata in Tabella 3.2.

δ	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

Tabella 3.2 Tabella di transizione di un ASFD che riconosce parole con un numero pari di a o di b.

La funzione di transizione è inoltre rappresentata, sotto forma di grafo di transizione, in Figura 3.3.

Esempio 3.4 Un altro semplice esempio di automa a stati finiti deterministico è dato dall'automa che riconosce gli identificatori di un linguaggio di programmazione, dove un identificatore è definito come una stringa di lunghezza arbitraria avente come primo carattere un carattere alfabetico, ed i cui restanti caratteri siano tutti alfanumerici.

Un automa $\mathcal{A}$ che riconosce tali identificatori ha alfabeto $\Sigma = \{a, \ldots, z, 0, \ldots, 9\}$ e tre stati, due di quali, q_0 e q_1, usati rispettivamente per accettare il primo carattere ed i caratteri successivi al primo, ed il terzo, lo stato q_2, che riveste il ruolo di stato di errore. Il grafo di transizione di questo automa è riportato in Figura 3.4.

Esercizio 3.2 Definire un ASFD che riconosce le stringhe su $\{a, b\}$ caratterizzate dal fatto che il secondo carattere è b.

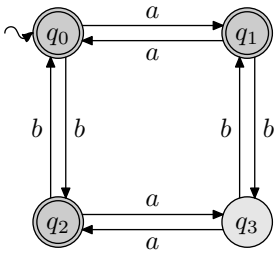


FIGURA 3.3 Diagramma degli stati di un ASFD che riconosce parole con un numero pari di a o di b.

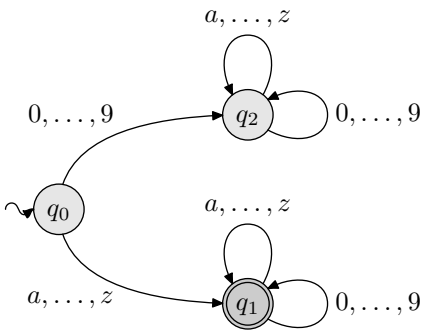


FIGURA 3.4 Diagramma degli stati di un ASFD che riconosce identificatori.

Esercizio 3.3 Definire un ASFD che riconosce le stringhe su $\{a, b\}$ caratterizzate dal fatto che il penultimo carattere è b.

Con riferimento alla Definizione 3.1 di ASFD, ricordiamo che la funzione di transizione δ deve essere una funzione totale.[3] Tuttavia, se rilassiamo il vincolo di totalità della δ ammettendo che possa essere parziale, non viene modificata sostanzialmente l'espressività del modello di macchina in quanto ogni ASFD con funzione di transizione $\delta : Q \times \Sigma \mapsto Q$ non totale può essere trasformato in un ASFD con funzione di transizione totale ed equivalente $\delta' : Q \cup \{\overline{q}\} \times \Sigma \mapsto Q \cup \{\overline{q}\}$ definita nel modo seguente:

1. Se $\delta(q, a)$ è definito allora $\delta'(q, a) = \delta(q, a)$;
2. Se $\delta(q, a)$ non è definito allora $\delta'(q, a) = \overline{q}$;
3. $\delta'(\overline{q}, a) = \overline{q}$ per ogni carattere $a \in \Sigma$.

Esercizio 3.4 Dimostrare che, dato un ASFD con funzione di transizione parziale, l'ASFD con funzione di transizione totale, costruito come sopra illustrato, riconosce lo stesso linguaggio.

Consideriamo ora l'insieme $\mathcal{R}$ dei linguaggi riconoscibili da automi a stati finiti deterministici, cioè l'insieme

$$\mathcal{R} = \{L \mid L \subseteq \Sigma^* \text{ ed esiste un automa } \mathcal{A} \text{ tale che } L = L(\mathcal{A})\}.$$

Nel seguito studieremo varie proprietà dei linguaggi contenuti nell'insieme $\mathcal{R}$. In particolare, dimostreremo che questa classe di linguaggi coincide con quella dei linguaggi generati dalle grammatiche di tipo 3 e con quella dei linguaggi definiti da espressioni regolari. Il nome di *linguaggi regolari*, generalmente attribuito ai linguaggi di tipo 3, deriva appunto da queste proprietà.

3.2 Automi a stati finiti non deterministici

In questa sezione prenderemo in esame la classe più estesa degli automi a stati finiti, costituita dalla classe degli *automi a stati finiti non deterministici*, che risulterà particolarmente utile per studiare le proprietà dei linguaggi regolari. Tale estensione si ottiene, coerentemente con quanto esposto nella Sezione 2.5.2, definendo l'insieme delle transizioni non mediante una funzione, ma attraverso una relazione o, equivalentemente, come una funzione nell'insieme

[3] Queste considerazioni possono essere immediatamente applicate anche al caso di automi a stati finiti non deterministici.

delle parti dell'insieme degli stati. In altre parole tali automi sono caratterizzati dal fatto che, ad ogni passo, una applicazione della funzione di transizione definisce più stati anziché uno solo.

Definizione 3.7 *Un* automa a stati finiti non deterministico *(ASFND) è una quintupla* $\mathcal{A}_N = \langle \Sigma, Q, \delta_N, q_0, F \rangle$*, in cui* $\Sigma = \{a_1, \ldots, a_n\}$ *è l'*alfabeto di input*,* $Q = \{q_0, \ldots, q_m\}$ *è l'insieme finito e non vuoto degli* stati *interni,* $q_0 \in Q$ *è lo* stato iniziale*,* $F \subseteq Q$ *è l'insieme degli* stati finali*, e* $\delta_N : Q \times \Sigma \mapsto \mathcal{P}(Q)$ *è una funzione (parziale), detta* funzione di transizione*, che ad ogni coppia* $\langle stato, carattere \rangle$ *su cui è definita associa un sottoinsieme di* Q *(eventualmente vuoto).*[4]

Esempio 3.5 In Tabella 3.3 viene presentata la funzione di transizione di un automa a stati finiti non deterministico: come si può osservare, in corrispondenza ad alcune coppie stato-carattere (segnatamente le coppie (q_0, b) e (q_1, a)) la funzione fornisce più stati, corrispondenti a diverse possibili continuazioni di una computazione.

δ_N	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2, q_3\}$	$\{q_3\}$
q_2	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$

Tabella 3.3 Esempio di tabella di transizione di un ASFND.

Un automa a stati finiti non deterministico può essere anch'esso descritto, così come un ASFD, per mezzo di un grafo di transizione: in questo caso, per effetto del non determinismo, esisteranno archi uscenti dal medesimo nodo etichettati con il medesimo carattere.

Esempio 3.6 In Figura 3.5 è riportato il grafo di transizione corrispondente all'automa a stati finiti non deterministico la cui funzione di transizione è descritta nella Tabella 3.3. La presenza del non determinismo fa sì che dal nodo etichettato q_1 escano due archi etichettati con il carattere b, così come dal nodo etichettato q_2 escono due archi etichettati con il carattere a.

Dalla sua stessa definizione deriva che un automa a stati finiti non deterministico definisce quindi, data una stringa in input, un insieme di computazioni.

[4] Si osservi che anche in questo caso si applicano le considerazioni fatte nella sezione precedente a proposito della possibilità di definire funzioni di transizione parziali. Comunque, è del tutto equivalente dire che la funzione di transizione non è definita per un dato stato q e per un dato carattere di input a, o che essa è definita e $\delta_N(q, a) = \emptyset$.

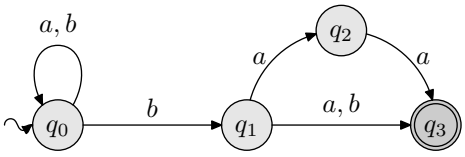


FIGURA 3.5 Grafo di transizione dell'ASFND in Tabella 3.3.

Alternativamente, come visto in Sezione 2.5.2, possiamo considerare che l'automa esegua una sola *computazione non deterministica*, nel corso della quale, per ogni carattere letto, assume non uno solo, ma un insieme di stati attuali e transita, ad ogni nuovo carattere, non da stato a stato ma da un insieme di stati ad un insieme di stati.[5]

Esempio 3.7 L'automa in Figura 3.5 definisce, in corrispondenza alla stringa in input *bba*, le tre computazioni:

- $(q_0, bba) \vdash\!\!- (q_0, ba) \vdash\!\!- (q_0, a) \vdash\!\!- (q_0, \varepsilon)$;
- $(q_0, bba) \vdash\!\!- (q_0, ba) \vdash\!\!- (q_1, a) \vdash\!\!- (q_2, \varepsilon)$;
- $(q_0, bba) \vdash\!\!- (q_0, ba) \vdash\!\!- (q_1, a) \vdash\!\!- (q_3, \varepsilon)$.

Inoltre, il prefisso *bb* della stringa di input dà luogo anche alla computazione:

- $(q_0, bb) \vdash\!\!- (q_1, b) \vdash\!\!- (q_3, \varepsilon)$;

la quale però non presenta continuazioni possibili. L'albero di computazione in Figura 3.6 mostra il complesso di tali computazioni.

Alternativamente, possiamo considerare che l'automa definisca la computazione non deterministica:

- $(\{q_0\}, bba) \vdash\!\!- (\{q_0, q_1\}, ba) \vdash\!\!- (\{q_0, q_1, q_3\}, a) \vdash\!\!- (\{q_0, q_2, q_3\}, \varepsilon)$.

Ricordando quanto detto nella Sezione 2.5.2, diciamo che una stringa viene accettata da un automa a stati finiti non deterministico se almeno una delle computazioni definite per la stringa stessa è di accettazione. In modo sintetico, utilizzando il concetto di computazione non deterministica, possiamo dire che una stringa x è accettata se $(\{q_0\}, x) \overset{*}{\vdash\!\!-} (\mathcal{Q}, \varepsilon)$, dove $\mathcal{Q} \subseteq Q$ e $\mathcal{Q} \cap F \neq \emptyset$.

Esercizio 3.5 Definiamo un ε-ASFND come un ASFND avente funzione di transizione $\delta_N : Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$, vale a dire per il quale sono ammesse ε-transizioni. Si dimostri che il modello degli ε-ASFND è equivalente a quello degli ASFND.
[Suggerimento: Definire ed utilizzare una funzione $c : Q \mapsto \mathcal{P}(Q)$ di ε-chiusura, che associa ad ogni stato l'insieme degli stati raggiungibili con sequenze di ε-transizioni, per derivare da ogni ASFND un ε-ASFND equivalente.]

[5] Anche se in modo improprio, useremo comunque la notazione $\vdash\!\!-$ per indicare la transizione tra insiemi di configurazioni nell'ambito di una computazione non deterministica.

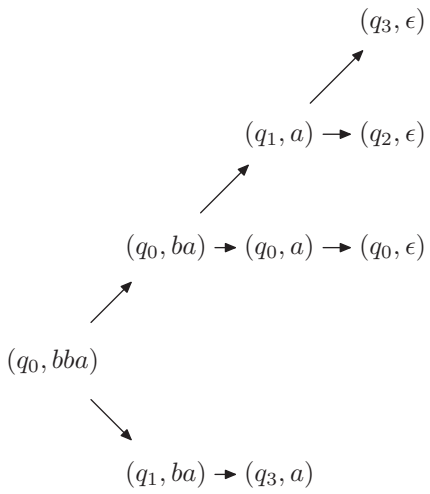


FIGURA 3.6 Albero di computazione relativo all'Esempio 3.7.

Esempio 3.8 Nel caso dell'Esempio 3.7, la stringa *bba* viene accettata in quanto una delle computazioni conduce alla configurazione di accettazione (q_3, ε). L'accettazione di *bba* deriva anche dal fatto che al termine della computazione non deterministica l'automa, finita di leggere la stringa, si trovi nell'insieme di stati $\{q_0, q_2, q_3\}$, e che $q_3 \in F$.

Possiamo allora definire il linguaggio $L(\mathcal{A})$ accettato da un ASFND $\mathcal{A}$ nel modo seguente:

$$L(\mathcal{A}) = \left\{ x \in \Sigma^* \mid (\{q_0\}, x) \stackrel{*}{\vdash\!\!-} (\mathcal{Q}, \varepsilon), \mathcal{Q} \cap F \neq \emptyset \right\}.$$

Esercizio 3.6 Dimostrare che per ogni ASFND esiste un ASFND equivalente avente $|F| = 1$.
[Suggerimento: Utilizzare quanto mostrato nell'Esercizio 3.5.]

Un modo alternativo di definire il linguaggio accettato da un ASFND richiede, analogamente a quanto fatto nel caso degli ASFD, di estendere alle stringhe la definizione di funzione di transizione.

Definizione 3.8 *Dato un ASFND, la* funzione di transizione estesa *è la funzione* $\overline{\delta}_N : Q \times \Sigma^* \mapsto \mathcal{P}(Q)$*, definita nel seguente modo*

$$\overline{\delta}_N(q, \varepsilon) \;=\; \{q\}$$

$$\overline{\delta}_N(q, xa) = \bigcup_{p \in \overline{\delta}_N(q,x)} \delta_N(p, a)$$

dove $a \in \Sigma$, $x \in \Sigma^*$, $p \in Q$.

Dalla definizione precedente, avremo quindi che, dato uno stato q ed una stringa di input x, lo stato q' appartiene all'insieme $\overline{\delta}_N(q, x)$ se e solo se esiste una computazione dell'automa la quale, a partire da q ed in conseguenza della lettura della stringa x, conduce allo stato q'.

Definiamo ora la classe di linguaggi accettati dagli ASFND.

Definizione 3.9 *Il linguaggio accettato da un ASFND* $\mathcal{A}_N$ *è l'insieme*

$$L(\mathcal{A}_N) = \left\{ x \in \Sigma^* | \overline{\delta}_N(q_0, x) \cap F \neq \emptyset \right\}.$$

Successivamente ci porremo il problema di stabilire la relazione esistente fra tale classe e quella dei linguaggi riconosciuti dagli ASFD.

Esempio 3.9 Si consideri il linguaggio, definito su $\Sigma = \{a, b\}$, delle parole terminanti con *bb*, o *ba* o *baa*. L'automa non deterministico $\mathcal{A}_N$ che riconosce tale linguaggio ha $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_3\}$, e la relativa funzione di transizione è rappresentata come tabella di transizione in Tabella 3.3 e come grafo di transizione in Figura 3.5.

Si consideri ora, ad esempio, la stringa *abba*: possiamo allora verificare che, dopo 1 passo di computazione, l'insieme degli stati attuali è $\{q_0\}$, dopo 2 passi $\{q_0, q_1\}$, dopo 3 passi $\{q_0, q_1, q_3\}$, dopo 4 passi $\{q_0, q_2, q_3\}$; la stringa è accettata in quanto uno di questi stati (q_3) è finale.

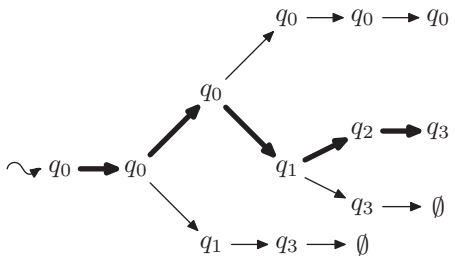


FIGURA 3.7 Albero delle transizioni seguite dall'automa dell'Esempio 3.5 con input *abbaa*.

Esempio 3.10 Consideriamo l'ASFND dell'Esempio 3.9. In Figura 3.7 viene presentato l'albero delle traiettorie seguite dall'automa nello spazio degli stati quando esso riceve in ingresso la stringa *abbaa*.

Come si vede, in questo caso un solo ramo dell'albero, quello iniziante in q_0 e terminante in q_3 (riportato in modo più marcato) conduce ad uno stato finale, ma ciò è sufficiente perché la stringa sia accettata.

Si osservi anche che, nell'Esempio 3.10, una eventuale conoscenza di quale transizione eseguire in corrispondenza ad ogni scelta non deterministica (passi 2, 3 e 4 della computazione) permetterebbe anche ad un automa deterministico di indovinare il cammino accettante: infatti, ad ogni passo per il quale la funzione di transizione fornisce un insieme di più stati, l'automa sarebbe in grado di "scegliere" uno stato sul cammino che conduce ad uno stato finale.

Tale conoscenza può essere rappresentata da una opportuna stringa di "scelta" di 5 caratteri (uno per ogni passo di computazione) sull'alfabeto $\{1, 2\}$, dove il simbolo 1 indica che lo stato successivo è il primo tra quelli (al massimo 2) specificati dalla funzione di transizione, mentre il simbolo 2 indica che lo stato successivo è il secondo. Quindi, il cammino di computazione accettante è specificato dalla stringa 11211: tale stringa può essere interpretata come una "dimostrazione" che *abbaa* appartiene al linguaggio accettato dall'automa. Tali considerazioni saranno riprese nel Capitolo 9 in cui vengono, tra l'altro, considerati i linguaggi accettabili efficientemente (in termini di lunghezza della computazione) da dispositivi di calcolo più potenti e complessi, quali le Macchine di Turing non deterministiche.

Esempio 3.11 Si consideri il linguaggio definito dall'espressione regolare

$$\big((a^+a + b)ba\big)^*\big(b + bb + a^+(b + a + ab)\big).$$

In Figura 3.8 è riportato il grafo di transizione dell'automa $\mathcal{A}_N$ che riconosce tale linguaggio.

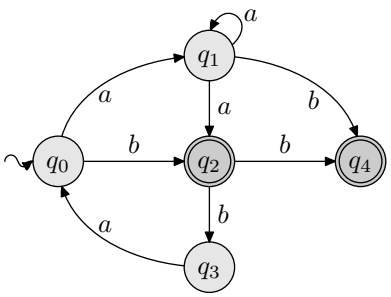


FIGURA 3.8 Esempio di ASFND.

Esercizio 3.7 Costruire l'automa non deterministico che riconosce il linguaggio delle stringhe su $\{a, b\}$ in cui il penultimo carattere è una b.

3.3 Relazioni tra ASFD, ASFND e grammatiche di tipo 3

In base alle definizioni date si potrebbe pensare che gli ASFND siano dispositivi più potenti degli ASFD, nel senso che riconoscono una classe di linguaggi più ampia. In realtà, anche se in altri tipi di macchine astratte il non determinismo accresce effettivamente il potere computazionale, nel caso degli automi a stati finiti non è così: infatti si può dimostrare che gli ASFD e gli ASFND riconoscono la medesima classe di linguaggi.

Teorema 3.1 *Dato un ASFD che riconosce un linguaggio L, esiste corrispondentemente un ASFND che riconosce lo stesso linguaggio L; viceversa, dato un ASFND che riconosce un linguaggio L', esiste un ASFD che riconosce lo stesso linguaggio L'.*

Dimostrazione. La dimostrazione si divide in due parti: la prima è il passaggio dall'automa deterministico a quello non deterministico, la seconda il passaggio inverso.

La prima parte della dimostrazione è banale, perché la simulazione di un ASFD mediante un ASFND si riduce alla costruzione di un automa non deterministico con stessi insiemi Σ, Q e F e in cui $\forall q \in Q, a \in \Sigma$ abbiamo che $\delta(q,a) = q'$ implica $\delta_N(q,a) = \{q'\}$.

La seconda parte della dimostrazione è più complessa. Dato un ASFND il cui insieme di stati è Q, essa si basa sull'idea di costruire un ASFD il cui insieme degli stati è in corrispondenza biunivoca con l'insieme delle parti di Q. Più precisamente, dato l'automa non deterministico

$$\mathcal{A}_N = \langle \Sigma, Q, \delta_N, q_0, F \rangle$$

si costruisce un automa deterministico equivalente

$$\mathcal{A}' = \langle \Sigma', Q', \delta', q_0', F' \rangle$$

procedendo come nel modi seguente.

- Si pone $\Sigma' = \Sigma$.
- Si definisce un insieme di stati Q' tale che $\mid Q' \mid = \mid \mathcal{P}(Q) \mid$ e lo si pone in corrispondenza biunivoca con $\mathcal{P}(Q)$. L'elemento di Q' che corrisponde al sottoinsieme $\{q_{i_1}, \ldots, q_{i_k}\}$ di Q sarà indicato con la notazione $[q_{i_1}, \ldots, q_{i_k}]$, in cui, per convenzione, i nomi degli stati di Q sono ordinati lessicograficamente. Quindi Q' risulta così definito:

  $$Q' = \{[q_{i_1}, \ldots, q_{i_k}] | \{q_{i_1}, \ldots, q_{i_k}\} \in \mathcal{P}(Q)\}.$$

  Si noti che $|Q'| = 2^{|Q|}$.
- Si pone $q_0' = [q_0]$.

- Si definisce F' come l'insieme degli stati di Q' corrispondenti a sottoinsiemi di Q che contengono almeno un elemento di F, cioè:

$$F' = \{[q_{i_1}, \ldots, q_{i_k}] \mid \{q_{i_1}, \ldots, q_{i_k}\} \in \mathcal{P}(Q) \wedge \{q_{i_1}, \ldots, q_{i_k}\} \cap F \neq \emptyset\}.$$

- La funzione δ' è definita nel seguente modo:

$$\forall q_{i_1}, \ldots, q_{i_k} \in Q, \forall a \in \Sigma,\ \delta'([q_{i_1}, \ldots, q_{i_k}], a) = [q_{j_1}, \ldots, q_{j_h}],$$

  se e solo se $\delta_N(q_{i_1}, a) \cup \ldots \cup \delta_N(q_{i_k}, a) = \{q_{j_1}, \ldots, q_{j_h}\}$, con $k > 0$ e $h \geq 0$. Inoltre si assume che $\forall a \in \Sigma\ \delta'([\,], a) = [\,]$.

Si deve ora dimostrare che ad ogni computazione effettuata dall'automa $\mathcal{A}'$ ne corrisponde una equivalente dell'automa $\mathcal{A}_N$ e viceversa. Per farlo bisogna mostrare in sostanza che per ogni $x \in \Sigma^*$

$$\overline{\delta}'([q_0], x) = [q_{j_1}, \ldots, q_{j_h}] \Longleftrightarrow \overline{\delta}_N(q_0, x) = \{q_{j_1}, \ldots, q_{j_h}\}.$$

Ciò può essere provato per induzione sulla lunghezza di x.

Nel passo base, essendo $\mid x \mid = 0$, vale necessariamente $x = \varepsilon$, per cui abbiamo $\overline{\delta}_N(q_0, \varepsilon) = \{q_0\}$ e $\overline{\delta}'([q_0], \varepsilon) = [q_0]$.

Supponiamo ora che l'asserto valga per $\mid x \mid = m$ e proviamo che esso continua a valere per $\mid x \mid = m + 1$. Poniamo $x = x'a$, con $\mid x' \mid = m$. Per $\overline{\delta}_N$ abbiamo:

$$\overline{\delta}_N(q_0, x'a) = \bigcup_{p \in \overline{\delta}_N(q_0, x')} \delta_N(p, a).$$

Supponendo che $\overline{\delta}_N(q_0, x') = \{q_{i_1}, \ldots, q_{i_k}\}$ e che $\delta_N(q_{i_1}, a) \cup \ldots \cup \delta_N(q_{i_k}, a) = \{q_{j_1}, \ldots, q_{j_h}\}$ otteniamo

$$\overline{\delta}_N(q_0, x'a) = \{q_{j_1}, \ldots, q_{j_h}\}.$$

Per $\overline{\delta}'$ vale invece:

$$\overline{\delta}'(q_0, x'a) = \delta'(\overline{\delta}'([q_0], x'), a).$$

Essendo $\mid x' \mid = m$ possiamo sfruttare l'ipotesi induttiva, il che ci consente di scrivere:

$$\delta'(\overline{\delta}'([q_0], x'), a) = \delta'([q_{i_1}, \ldots, q_{i_k}], a),$$

che, per costruzione, vale proprio $[q_{j_1}, \ldots, q_{j_h}]$.

La prova è completata osservando che $\overline{\delta}'([q_0], x) \in F'$ esattamente quando $\delta_N(q_0, x)$ contiene uno stato di Q che è in F. □

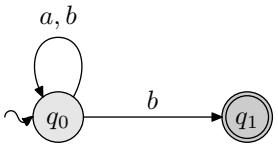


FIGURA 3.9 ASFND che riconosce le stringhe in $\{a, b\}^*$ terminanti con b.

Esempio 3.12 Si consideri l'automa non deterministico $\mathcal{A}_N$ che accetta tutte le stringhe in $\{a, b\}^*$ che terminano con b. Il suo grafo di transizione è riportato in Figura 3.9. L'automa deterministico $\mathcal{A}'$ corrispondente si può costruire seguendo i passi della dimostrazione precedente (vedi Figura 3.10).

- L'alfabeto di $\mathcal{A}'$ coincide con quello di $\mathcal{A}$, cioè $\Sigma' = \{a, b\}$.
- $\mathcal{A}'$ ha $2^{|Q|} = 4$ stati, due dei quali non significativi, in quanto non raggiungibili dallo stato iniziale.
- Lo stato iniziale è $[q_0]$.
- La funzione δ' è determinata considerando i 4 possibili stati di $\mathcal{A}'$ ed applicando a ciascuno di essi il procedimento descritto nel teorema precedente.

$$
\begin{aligned}
\delta'([\,], a) &= [\,] \\
\delta'([\,], b) &= [\,] \\
\delta'([q_0], a) &= [q_0] \\
\delta'([q_0], b) &= [q_0, q_1] \\
\delta'([q_1], a) &= [\,] \\
\delta'([q_1], b) &= [\,] \\
\delta'([q_0, q_1], a) &= [q_0] \\
\delta'([q_0, q_1], b) &= [q_0, q_1];
\end{aligned}
$$

  Come si vede, dei quattro possibili stati solo due (gli stati $[q_0]$ e $[q_0, q_1]$) sono raggiungibili da $[q_0]$, e quindi sono gli unici ad essere rappresentati in Figura 3.10.

- L'insieme F' è l'insieme degli stati raggiungibili che contengono un elemento di F; nel caso in esame $F' = \{[q_0, q_1]\}$.

Come evidenziato nell'esempio precedente, quando si realizza la trasformazione da un automa non deterministico con $\mid Q \mid$ stati ad un automa deterministico, non sempre tutti i $2^{|Q|}$ stati generati sono raggiungibili. In tal caso, possiamo

dire che l'insieme degli stati dell'automa deterministico è costituito dai soli stati raggiungibili. Per tale motivo, il procedimento di costruzione dell'automa non deterministico può essere reso più efficiente costruendo la funzione di transizione a partire dallo stato iniziale $[q_0]$, generando i soli stati raggiungibili. Si noti comunque che, in base al metodo generale di costruzione dell'ASFD

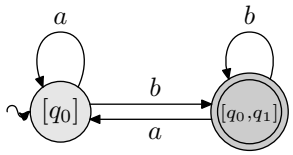


FIGURA 3.10 ASFD che riconosce le stringhe in $\{a, b\}^*$ terminanti con b.

equivalente ad un dato ASFND, si verificano situazioni in cui è necessario introdurre un numero di stati esponenziale nel numero degli stati dell'automa non deterministico. Gli automi non deterministici hanno infatti il pregio di essere a volte molto più "concisi" di quelli deterministici equivalenti. Vediamo innanzi tutto un esempio in cui l'ASFD ha più stati dell'ASFND equivalente.

Esempio 3.13 Si consideri l'automa di Figura 3.11, che riconosce le stringhe date dall'espressione regolare $(a + b)^*b(a + b)$. Procedendo come visto nell'esempio prece-

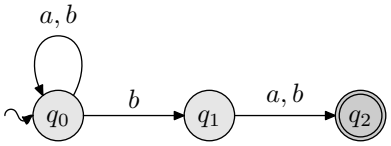


FIGURA 3.11 ASFND che riconosce le stringhe date da $(a + b)^*b(a + b)$.

dente si ottiene un ASFD con quattro stati, due dei quali sono finali, come illustrato in Figura 3.12.

Generalizzando l'esempio al caso delle stringhe $(a + b)^*b(a + b)^k$ si può vedere che mentre l'ASFND che riconosce tali stringhe ha in generale $k + 2$ stati, l'ASFD equivalente ne ha $O(2^k)$.

Esercizio 3.8 Costruire l'ASFND e l'ASFD che riconoscono le stringhe $(a+b)^*b(a+b)^k$, per $k = 2, 3$ e 4 e valutare come cresce la dimensione dei due automi al crescere di k.

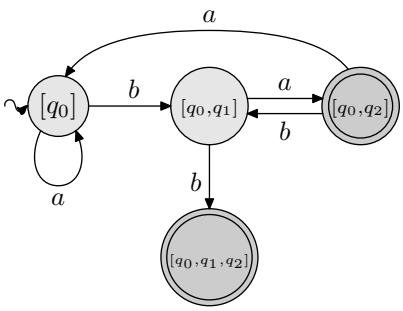


FIGURA 3.12 ASFD che riconosce le stringhe date da $(a+b)^*b(a+b)$.

Una rilevante conseguenza della equivalenza tra ASFD e ASFND è data dal seguente risultato.

Definizione 3.10 *Data una stringa $x = a_1 \cdots a_r$ definiamo stringa* riflessa *di x la stringa $\tilde{x} = a_r \cdots a_1$.*

Teorema 3.2 *Dato un linguaggio L riconosciuto da un ASF esiste un ASF che riconosce il linguaggio $\{\tilde{x} \mid x \in L\}$.*

Esercizio 3.9 Dimostrare il Teorema 3.2

Dopo avere mostrato l'equivalenza tra ASFD e ASFND, possiamo ora considerare la relazione esistente tra automi a stati finiti e grammatiche. Come già anticipato, la classe dei linguaggi accettati da automi a stati finiti coincide con quella dei linguaggi generabili per mezzo di grammatiche di tipo 3.

Teorema 3.3 *Data una grammatica regolare $\mathcal{G} = \langle V_T, V_N, P, S\rangle$, esiste un ASFND $\mathcal{A}_N = \langle \Sigma, Q, \delta_N, q_0, F\rangle$ che riconosce il linguaggio che essa genera. Viceversa, dato un ASFND esiste una grammatica regolare che genera il linguaggio che esso riconosce.*

Dimostrazione. La dimostrazione si divide in due parti: la prima consiste nel dimostrare che per ogni grammatica regolare si può trovare un ASFND equivalente, la seconda nel dimostrare il passaggio inverso.

Consideriamo una grammatica regolare $\mathcal{G} = \langle V_T, V_N, P, S\rangle$; assumiamo, senza perdita di generalità, che l'unica possibile ε-produzione sia $S \longrightarrow \varepsilon$.

Per costruire l'ASFND $\mathcal{A}_N$ a partire da $\mathcal{G}$ si procede come segue.

- Si pone: $\Sigma = V_T$.

- Ad ogni non terminale si associa uno degli stati di Q e si aggiunge inoltre un ulteriore stato q_F. Si pone, cioè:

$$Q = \{q_I \mid I \in V_N\} \cup \{q_F\}.$$

- Lo stato iniziale viene scelto corrispondente all'assioma, per cui si pone: $q_0 = q_S$.

- L'insieme F degli stati finali si sceglie come

$$F = \begin{cases} \{q_0, q_F\} & \text{se } S \longrightarrow \varepsilon \in P \\ \{q_F\} & \text{altrimenti} \end{cases}$$

- La funzione di transizione δ_N si definisce, per $a \in \Sigma$ e $q_B \in Q$, come

$$\delta_N(q_B, a) = \begin{cases} \{q_C \mid B \longrightarrow aC \in P\} \cup \{q_F\} & \text{se } B \longrightarrow a \in P \\ \{q_C \mid B \longrightarrow aC \in P\} & \text{altrimenti.} \end{cases}$$

  Ovviamente l'automa è non deterministico perché ad ogni non terminale possono essere associate più produzioni aventi la parte destra iniziante con lo stesso terminale.

La dimostrazione che l'automa $\mathcal{A}_N$ riconosce lo stesso linguaggio generato dalla grammatica $\mathcal{G}$ può essere data mostrando che $\forall x \in \Sigma^*$ $S \stackrel{*}{\Longrightarrow} x$ se e solo se $\overline{\delta}_N(q_S, x)$ include uno stato finale. Nel caso $x = \varepsilon$ ciò è immediato per costruzione.

Nel caso $x \in \Sigma^+$ proveremo dapprima, per induzione sulla lunghezza di x, che esiste una derivazione $S \stackrel{*}{\Longrightarrow} xZ$ se e solo se $q_Z \in \overline{\delta}_N(q_S, x)$.

Sia $\mid x \mid = 1$. Supponiamo $x = a$, con $a \in \Sigma$: allora abbiamo che $S \longrightarrow aZ \in P$, e quindi $S \Longrightarrow aZ$ se e solo se, per costruzione dell'automa, $q_Z \in \delta_N(q_S, a)$.

Sia $\mid y \mid = n \geq 1$ e $x = ya$, con $a \in \Sigma$. Per l'ipotesi induttiva il risultato si assume valido per y, cioè $S \stackrel{*}{\Longrightarrow} yZ$ se e solo se $q_Z \in \overline{\delta}_N(q_S, y)$. Mostriamo che esso è valido anche per la stringa x. Infatti, osserviamo che una derivazione $S \stackrel{*}{\Longrightarrow} xZ'$ esiste se e solo se esiste $Z \in V_N$ tale che $S \stackrel{*}{\Longrightarrow} yZ \Longrightarrow yaZ' = xZ'$; per induzione ciò è equivalente ad assumere che esista $q_Z \in Q$ tale che

$$q_Z \in \overline{\delta}_N(q_S, y) \text{ e } q_{Z'} \in \delta_N(q_Z, a),$$

e ciò ovviamente avviene se e solo se

$$q_{Z'} \in \bigcup_{p \in \overline{\delta}_N(q_S, y)} \delta_N(p, a),$$

cioè $q_{Z'} \in \overline{\delta}_N(q_S, x)$.

Osserviamo ora che $S \stackrel{*}{\Longrightarrow} x$ se e solo se esistono $Z \in V_N$, $y \in \Sigma^*$ e $Z \longrightarrow a \in P$ tali che $x = ya$ e $S \stackrel{*}{\Longrightarrow} yZ \stackrel{*}{\Longrightarrow} ya = x$. Da quanto detto sopra, ciò è vero se e solo se $q_Z \in \overline{\delta}_N(q_S, y)$ e $q_F \in \delta_N(q_Z, a)$, e quindi se e solo se $q_F \in \overline{\delta}_N(q_S, ya) = \overline{\delta}_N(q_S, x)$.

In base alla definizione di F si può dimostrare che esiste una derivazione $S \Longrightarrow x$ se e solo se $\overline{\delta}_N(q_S, x) \cap F \neq \emptyset$.

Mostriamo ora che, dato un automa a stati finiti (che, senza perdita di generalità, possiamo assumere deterministico), è possibile costruire una grammatica di tipo 3 che genera lo stesso linguaggio da esso riconosciuto. Sia $\langle \Sigma, Q, \delta, q_0, F \rangle$ l'automa dato: supponendo per il momento che $q_0 \notin F$, definiamo la seguente grammatica.

$1'$ $V_T = \Sigma$;

$2'$ $V_N = \{A_i \mid \text{ per ogni } q_i \in Q\}$;

$3'$ $S = A_0$;

$4'$ per ogni regola di transizione $\delta(q_i, a) = q_j$ abbiamo la produzione $A_i \longrightarrow aA_j$, e se $q_j \in F$ anche la produzione $A_i \longrightarrow a$.

Se $q_0 \in F$ — cioè se il linguaggio riconosciuto dall'automa contiene anche la stringa vuota — arricchiamo V_T con un nuovo non terminale A'_0, per ogni produzione $A_0 \longrightarrow aA_i$ aggiungiamo la produzione $A'_0 \longrightarrow aA_i$ e introduciamo infine la produzione $A'_0 \longrightarrow \varepsilon$. In tal caso l'assioma sarà A'_0 anziché A_0.

Dimostriamo ora che una stringa x è derivabile mediante la grammatica se e solo se essa è accettata dall'automa. Consideriamo innanzi tutto il caso in cui la stringa x è di lunghezza nulla. In tal caso necessariamente $q_0 \in F$ e per costruzione l'assioma è A'_0 ed esiste la produzione $A'_0 \longrightarrow \varepsilon$. Nel caso in cui la lunghezza di x è maggiore di 0 possiamo dimostrare che esiste la derivazione $A_i \stackrel{*}{\Longrightarrow} xA_j$ se e solo se $\overline{\delta}(q_i, x) = q_j$ e che esiste anche la derivazione $A_i \stackrel{*}{\Longrightarrow} x$ se e solo se $q_j \in F$.

La dimostrazione procede per induzione.

Se $x = a$ abbiamo $A_i \Longrightarrow aA_j$ se e solo se $\overline{\delta}(q_i, a) = q_j$ (e inoltre abbiamo che $A_i \Longrightarrow a$ se e solo se $q_j \in F$). Assumiamo ora che la proprietà sia valida per ogni stringa y di lunghezza n e mostriamo che essa è valida per ogni stringa x di lunghezza $n+1$. Infatti, per l'ipotesi induttiva abbiamo $A_i \stackrel{*}{\Longrightarrow} yA_k$ se e solo se $\overline{\delta}(q_i, y) = q_k$, inoltre per costruzione abbiamo $A_k \Longrightarrow aA_j$ se e solo se $\overline{\delta}(q_k, a) = q_j$. Pertanto abbiamo $A_i \stackrel{*}{\Longrightarrow} yA_k \Longrightarrow yaA_j$ $(= xA_j)$ se e solo se $\overline{\delta}(q_i, x) = \overline{\delta}(q_i, ya) = \delta(\overline{\delta}(q_i, y), a) = q_j$ e $q_k = \overline{\delta}(q_i, y)$. Analogamente, se

$q_i \in F$, abbiamo $A_i \stackrel{*}{\Longrightarrow} x$ se e solo se $\overline{\delta}(q_i, x) = q_j$. In particolare, abbiamo $A_0 \stackrel{*}{\Longrightarrow} x$ (o $A'_0 \stackrel{*}{\Longrightarrow} x$ se $q_0 \in F$) se e solo se $\overline{\delta}(q_0, x) = q_j$ e $q_j \in F$. □

Esempio 3.14 Si consideri il linguaggio L rappresentato dall'espressione regolare $a(a + ba)^*a$. Tale linguaggio è generato dalla grammatica

$$\mathcal{G} = \langle \{a, b\}, \{S, B\}, P, S \rangle$$

in cui P è dato da

$$\begin{array}{lcl} S & \longrightarrow & aB \\ B & \longrightarrow & aB \mid bS \mid a. \end{array}$$

L'automa $\mathcal{A}_N = \langle \Sigma, Q, \delta_N, q_0, F \rangle$ che riconosce il linguaggio L è definito nel modo seguente (vedi Figura 3.13).

- $\Sigma = V_T$.
- $Q = \{q_S, q_B, q_F\}$.
- $q_0 = q_S$ come stato iniziale.
- $F = \{q_F\}$.
- δ_N è definita come segue

$$\begin{array}{lcl} \delta_N(q_S, a) & = & \{q_B\} \\ \delta_N(q_B, a) & = & \{q_B, q_F\} \\ \delta_N(q_B, b) & = & \{q_S\}. \end{array}$$

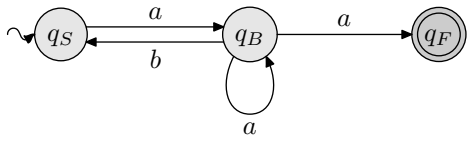


FIGURA 3.13 Automa non deterministico che riconosce il linguaggio L rappresentato dall'espressione regolare $a(a + ba)^*a$.

In base al procedimento visto nella dimostrazione del Teorema 3.1, ed esemplificato nell'Esempio 3.12, da questo automa si ricava l'automa deterministico corrispondente, riportato in Figura 3.14. A partire dall'automa di Figura 3.14 è possibile derivare la grammatica

$$\mathcal{G}' = \langle \{a, b\}, \{A_0, A_1, A_2, A_3\}, P, A_0 \rangle$$

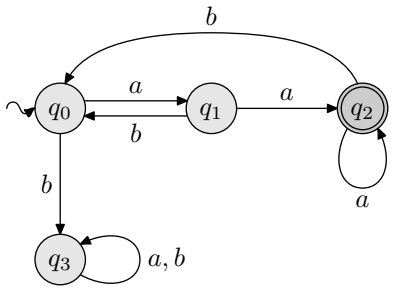


FIGURA 3.14 Automa deterministico che riconosce il linguaggio L rappresentato dall'espressione regolare $a(a + ba)^*a$.

---

con le seguenti produzioni (si noti che le ultime tre produzioni sono "inutili", in quanto corrispondenti alle transizioni relative allo stato di errore q_2).

$$
\begin{array}{lcl}
A_0 & \longrightarrow & aA_1 \\
A_1 & \longrightarrow & bA_0 \\
A_1 & \longrightarrow & aA_3 \\
A_1 & \longrightarrow & a \\
A_3 & \longrightarrow & aA_3 \\
A_3 & \longrightarrow & a \\
A_3 & \longrightarrow & bA_0
\end{array}
$$

$$
\begin{array}{lcl}
A_2 & \longrightarrow & aA_2 \\
A_2 & \longrightarrow & bA_2 \\
A_0 & \longrightarrow & bA_2
\end{array}
$$

Esercizio 3.10 Si consideri la grammatica regolare avente le seguenti produzioni:

$$
\begin{array}{lcl}
S & \longrightarrow & 0A \mid 1B \mid 0S \\
A & \longrightarrow & aB \mid bA \mid a \\
B & \longrightarrow & bA \mid aB \mid b.
\end{array}
$$

Si generino l'automa non deterministico e l'automa deterministico che riconoscono il linguaggio generato da tale grammatica. A partire dall'automa deterministico, generare poi la grammatica corrispondente.

Si osservi che la procedura, descritta nella seconda parte della dimostrazione del Teorema 3.3, utilizzata per costruire una grammatica regolare equivalente ad un dato automa a stati finiti deterministico può essere appli-

cata anche qualora tale automa sia non deterministico, con la sola modifica che per ogni transizione $\delta_N(q_i, a) = \{q_{j_1}, \ldots, q_{j_h}\}$ abbiamo le produzioni $A_i \longrightarrow aA_{j_1} \mid aA_{j_2} \mid \ldots \mid aA_{j_h}$ e, se almeno uno fra gli A_{j_l} $(l = 1, \ldots, h)$ appartiene ad F, abbiamo anche la produzione $A_i \longrightarrow a$. L'assunzione del determinismo ha tuttavia consentito di semplificare la dimostrazione.

Esercizio 3.11 Dimostrare che, per ogni grammatica lineare sinistra, esiste una grammatica regolare equivalente.
[Suggerimento: Utilizzare quanto mostrato in Esercizio 3.5.]

3.4 Il "pumping lemma" per i linguaggi regolari

Un risultato fondamentale sulla struttura dei linguaggi regolari è dato dal seguente teorema, noto nella letteratura specializzata come "pumping lemma".

Questo risultato essenzialmente ci dice che ogni stringa sufficientemente lunga appartenenente ad un linguaggio regolare ha una struttura che presenta delle regolarità, nel senso, in particolare, che contiene una sottostringa che può essere ripetuta quanto si vuole, ottenendo sempre stringhe del linguaggio.

Teorema 3.4 (Pumping lemma per i linguaggi regolari) *Per ogni linguaggio regolare L esiste una costante n tale che se $z \in L$ e $\mid z \mid \geq n$ allora possiamo scrivere $z = uvw$, con $\mid uv \mid \leq n$, $\mid v \mid \geq 1$ e ottenere che $uv^iw \in L$, per ogni $i \geq 0$.*

Dimostrazione. Dato un linguaggio regolare L esiste un ASFD che lo riconosce. Sia $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ l'automa che riconosce L avente il minimo numero possibile di stati $n = \mid Q \mid$.

Sia z una stringa appartenente ad L, con $\mid z \mid \geq n$. In tal caso, deve necessariamente valere $\overline{\delta}(q_0, z) \in F$: supponiamo senza perdita di generalità che $q_{i_0}, q_{i_1}, \ldots, q_{i_{|z|}}$ sia la sequenza di stati attraversata da $\mathcal{A}$ durante il riconoscimento di z, con $q_{i_0} = q_0$ e $q_{i_{|z|}} \in F$. Dal momento che $\mid z \mid \geq n$ deve esistere, per il *pigeonhole principle*, almeno uno stato in cui l'automa si porta almeno due volte durante la lettura di z: sia j il minimo valore tale che q_{i_j} viene attraversato almeno due volte, sia u il più breve prefisso (eventualmente nullo) di z tale che $\overline{\delta}(q_0, u) = q_{i_j}$ e sia $z = ux$. Ovviamente, $\mid u \mid < n$ e $\overline{\delta}(q_{i_j}, x) = q_{i|z|}$. Sia inoltre v il più breve prefisso (non nullo) di x tale che $\overline{\delta}(q_{i_j}, v) = q_{i_j}$ e sia $x = vw$; vale ovviamente $\mid uv \mid \leq n$ e $\overline{\delta}(q_{i_j}, w) = q_{i_{|z|}}$ (vedi Figura 3.15).

Per ogni $i \geq 0$ abbiamo allora

$$\begin{aligned} \overline{\delta}(q_0, uv^iw) &= \overline{\delta}\Big(\overline{\delta}(q_0, u), v^iw\Big) \\ &= \overline{\delta}\left(q_{i_j}, v^iw\right) \end{aligned}$$

$$
\begin{aligned}
&= \overline{\delta}\Big(\overline{\delta}(q_{i_j}, v), v^{i-1}w\Big) \\
&= \overline{\delta}\left(q_{i_j}, v^{i-1}w\right) \\
&= \ldots \\
&= \overline{\delta}(q_{i_j}, w) \\
&= q_{i_{|z|}}
\end{aligned}
$$

il che mostra che ogni stringa del tipo uv^iw appartiene ad L. □

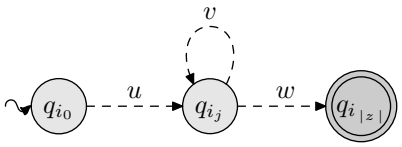


FIGURA 3.15 Comportamento dell'ASFD che accetta la stringa z nel pumping lemma.

Il pumping lemma evidenzia uno dei principali limiti degli automi finiti: essi non possiedono la capacità di contare. Infatti essi possono memorizzare mediante i propri stati solo un numero finito di diverse situazioni, mentre non possono memorizzare numeri arbitrariamente grandi o quantità arbitrariamente crescenti di informazione.

Si osservi che il pumping lemma fornisce soltanto una condizione necessaria perché un linguaggio sia regolare. Esso non può essere quindi utilizzato per mostrare la regolarità di un linguaggio, ma viene generalmente usato per dimostrare la non regolarità del linguaggio, come si può vedere nel seguente teorema.

Teorema 3.5 *Il linguaggio* $L = \{a^n b^n \mid n \geq 0\}$ *non è regolare.*

Dimostrazione. Si assuma, per assurdo, che L sia regolare: in tal caso il pumping lemma varrà per tutte le strighe in L di lunghezza maggiore o uguale di un opportuno n. Consideriamo ora una stringa $z = a^m b^m \in L$, $m > n$: in base al pumping lemma, dovrebbero esistere stringhe u, v, w tali che $z = uvw$, $| uv | \leq n$, $| v | \geq 1$ e per ogni $i \geq 0$ $uv^iw \in L$. Poiché $m > n$ e poiché deve essere $| uv | \leq n$, si deve avere $v = a^h$ per un intero positivo h. In tal caso, L dovrebbe però contenere anche stringhe del tipo $a^{m-h}a^{hi}b^m$, $i = 0, 1, \ldots$. Ciò mostra che l'ipotesi che il linguaggio L sia regolare è falsa. □

Esercizio 3.12 Si consideri il linguaggio $L = \{w\tilde{w} \mid w \in \{a, b\}^*\}$, ove si è indicata con $\tilde{w}$ la stringa ottenuta invertendo i caratteri presenti in w. Dimostrare, utilizzando il pumping lemma, che tale linguaggio non è regolare.

3.5 Proprietà di chiusura dei linguaggi regolari

In questa sezione mostreremo come la classe dei linguaggi regolari sia chiusa rispetto ad un insieme significativo di operazioni, insieme che include le operazioni di unione, complementazione, intersezione, concatenazione e iterazione.

Le dimostrazioni di tali proprietà presentano tutte uno stesso schema, in cui, a partire dagli ASFD che riconoscono i linguaggi regolari dati, viene derivato un automa (deterministico o non deterministico) che riconosce il linguaggio risultante.

Teorema 3.6 *Dati due linguaggi regolari L_1 e L_2, la loro unione $L_1 \cup L_2$ è un linguaggio regolare.*

Dimostrazione. Siano dati due qualunque automi deterministici $\mathcal{A}_1 = \langle \Sigma_1, Q_1, \delta_{N_1}, q_{0_1}, F_1 \rangle$ e $\mathcal{A}_2 = \langle \Sigma_2, Q_2, \delta_{N_2}, q_{0_2}, F_2 \rangle$, che accettano i linguaggi $L_1 = L(\mathcal{A}_1)$ e $L_2 = L(\mathcal{A}_2)$, rispettivamente. Mostriamo ora come sia possibile, a partire da $\mathcal{A}_1$ e $\mathcal{A}_2$, costruire un automa $\mathcal{A} = \{\Sigma, Q, \delta_N, q_0, F\}$ che riconosce il linguaggio $L = L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.

La costruzione procede nel modo seguente:

- $\Sigma = \Sigma_1 \cup \Sigma_2$.
- $Q = Q_1 \cup Q_2 \cup \{q_0\}$.
- $F = F_1 \cup F_2$, oppure $F = F_1 \cup F_2 \cup \{q_0\}$ se uno dei due automi $\mathcal{A}_1$, $\mathcal{A}_2$ riconosce anche la stringa vuota.
- La funzione di transizione δ_N viene definita come segue:

$$\begin{aligned} \delta_N(q, a) &= \delta_{N_1}(q, a) \text{ se } q \in Q_1, a \in \Sigma_1 \\ \delta_N(q, a) &= \delta_{N_2}(q, a) \text{ se } q \in Q_2, a \in \Sigma_2 \\ \delta_N(q_0, a) &= \delta_{N_1}(q_{0_1}, a) \cup \delta_{N_2}(q_{0_2}, a), a \in \Sigma. \end{aligned}$$

Si può verificare immediatamente che l'automa così definito accetta tutte e sole le stringhe in $L(\mathcal{A}_1)$ o in $L(\mathcal{A}_2)$. □

Si noti che anche se gli automi di partenza $\mathcal{A}_1$ e $\mathcal{A}_2$ sono deterministici, l'automa $\mathcal{A}$ costruito secondo la procedura testé descritta potrà risultare non deterministico.

Esempio 3.15 Si consideri l'automa

$$\mathcal{A}_1 = \langle \{a, b\}, \{q_0, q_1\}, \delta_1, q_0, \{q_1\} \rangle,$$

il cui grafo di transizione è rappresentato in Figura 3.16; si consideri inoltre l'automa $\mathcal{A}_2$:

$$\mathcal{A}_2 = \langle \{a, b\}, \{q_2, q_3\}, \delta_2, q_2, \{q_2\} \rangle,$$

il cui grafo di transizione è rappresentato in Figura 3.17.

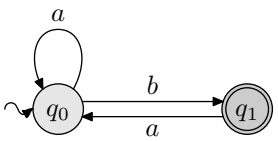


FIGURA 3.16 Automa $\mathcal{A}_1$ dell'Esempio 3.15.

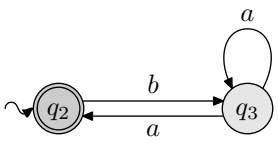


FIGURA 3.17 Automa $\mathcal{A}_2$ dell'Esempio 3.15.

L'automa $\mathcal{A} = \langle \Sigma, Q, \delta_N, q_0', F\} \rangle$ che accetta l'unione dei linguaggi accettati da $\mathcal{A}_1$ e da $\mathcal{A}_2$, si costruisce come segue.

- $\Sigma = \{a, b\}$.
- $Q = \{q_0', q_0, q_1, q_2, q_3\}$, dove q_0' è il un nuovo stato iniziale.
- $F = \{q_1, q_2\} \cup \{q_0'\}$ (perché il secondo automa riconosce anche la stringa vuota).
- La funzione di transizione δ_N è data da

$$\begin{array}{lcl}
\delta_N(q_0', a) & = & \big\{\delta_1(q_0, a)\big\} \\
\delta_N(q_0', b) & = & \big\{\delta_1(q_0, b), \delta_2(q_2, b)\big\} \\
\delta_N(q_0, a) & = & \big\{\delta_1(q_0, a)\big\} \\
\delta_N(q_0, b) & = & \big\{\delta_1(q_0, b)\big\} \\
\delta_N(q_1, a) & = & \big\{\delta_1(q_1, a)\big\} \\
\delta_N(q_2, b) & = & \big\{\delta_2(q_2, b)\big\} \\
\delta_N(q_3, a) & = & \big\{\delta_2(q_3, a)\big\} \\
\delta_N(q_3, b) & = & \big\{\delta_2(q_3, b)\big\}.
\end{array}$$

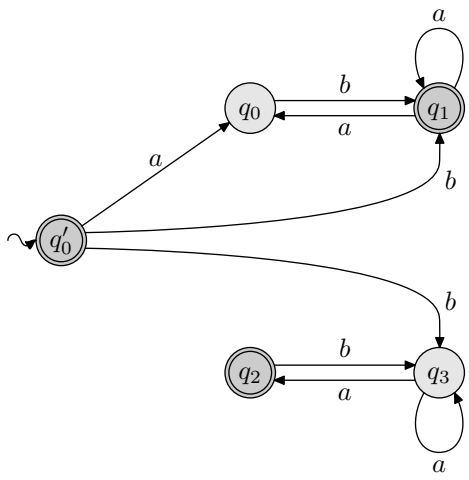


FIGURA 3.18 Automa che riconosce l'unione tra $L(\mathcal{A}_1)$ e $L(\mathcal{A}_2)$.

Il grafo di transizione dell'automa $\mathcal{A}$ è rappresentato in Figura 3.18.

Teorema 3.7 *Dato un linguaggio regolare L, il suo complemento $\overline{L}$ è un linguaggio regolare.*

Dimostrazione. Sia

$$\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$$

un automa deterministico che riconosce il linguaggio $L = L(\mathcal{A})$: si può allora costruire l'automa

$$\overline{\mathcal{A}} = \langle \Sigma, Q, \delta, q_0, \{Q - F\} \rangle;$$

che riconosce il linguaggio $\overline{L(\mathcal{A})}$. Infatti, ogni stringa che porta l'automa $\mathcal{A}$ in uno stato finale porta l'automa $\overline{\mathcal{A}}$ in uno stato non finale e, viceversa, ogni stringa che porta $\overline{\mathcal{A}}$ in uno stato finale porta $\mathcal{A}$ in uno stato non finale, per cui

$$L(\overline{\mathcal{A}}) = \Sigma^* - L(\mathcal{A}) = \overline{L(\mathcal{A})}.$$

□

Si noti che, per applicare la costruzione mostrata nel teorema precedente è necessario assumere che la funzione di transizione dell'automa dato sia totale.

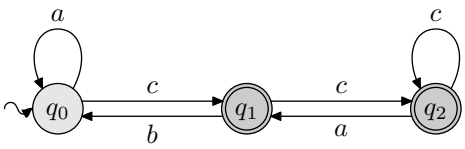


FIGURA 3.19 Esempio di automa.

Esempio 3.16 Si consideri il linguaggio $L(\mathcal{A})$, dove $\mathcal{A}$ è l'automa

$$\mathcal{A} = \langle \{a, b, c\}, \{q_0, q_1, q_2\}, \delta, q_0, \{q_1, q_2\} \rangle$$

il cui grafo di transizione è riportato in Figura 3.19. Per costruire l'automa $\overline{\mathcal{A}}$ che riconosce il linguaggio $\overline{L(\mathcal{A})}$, in questo caso si può procedere come segue.

Prima di tutto, osservando che l'automa dato ha una funzione di transizione parziale δ, è necessario trasformarlo in un automa con funzione di transizione $\overline{\delta}$ totale, aggiungendo il nuovo stato q_3 e le transizioni $\overline{\delta}(q_0, b) = q_3$, $\overline{\delta}(q_1, a) = q_3$, $\overline{\delta}(q_2, b) = q_3$. Per tutte le altre transizioni, $\overline{\delta} = \delta$.

Quindi, si procede come previsto dalla dimostrazione del teorema e si ottiene così l'automa seguente.

- Si introduce il nuovo stato finale d.
- Si pone:

$$\begin{aligned}
\overline{\delta}(q_0, a) &= \delta(q_0, a) \\
\overline{\delta}(q_0, c) &= \delta(q_0, c) \\
\overline{\delta}(q_1, b) &= \delta(q_1, b) \\
\overline{\delta}(q_1, c) &= \delta(q_1, c) \\
\overline{\delta}(q_2, a) &= \delta(q_2, a) \\
\overline{\delta}(q_2, c) &= \delta(q_2, c).
\end{aligned}$$

- Si pone:

$$\begin{aligned}
\overline{\delta}(q_0, b) &= q_3 \\
\overline{\delta}(q_1, a) &= q_3 \\
\overline{\delta}(q_2, b) &= q_3.
\end{aligned}$$

- Si pone

$$\begin{aligned}
\overline{\delta}(d, a) &= q_3 \\
\overline{\delta}(d, b) &= q_3
\end{aligned}$$

$$\overline{\delta}(d, c) \quad = \quad q_3.$$

Si ottiene così l'automa

$$\overline{\mathcal{A}} = \langle \{a, b, c\}, \{q_0, q_1, q_2, q_3\}, \overline{\delta}, q_0, \{q_0, q_3\} \rangle$$

il cui grafo di transizione è riportato in Figura 3.20.

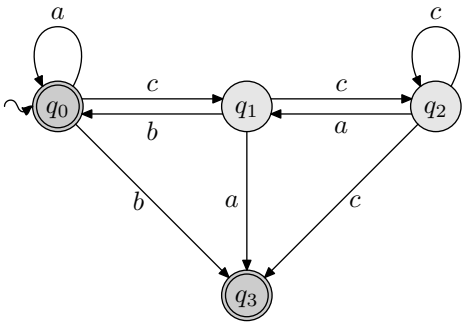


FIGURA 3.20 Automa che accetta il complemento del linguaggio accettato dall'automa di Figura 3.19.

In conseguenza dei due risultati precedenti, possiamo mostrare immediatamente che i linguaggi regolari sono chiusi anche rispetto all'intersezione.

Teorema 3.8 *Dati due linguaggi regolari L_1 e L_2, la loro intersezione $L = L_1 \cap L_2$ è un linguaggio regolare.*

Dimostrazione. È sufficiente osservare che, per la legge di De Morgan,

$$L = L_1 \cap L_2 \equiv \overline{\overline{L_1} \cup \overline{L_2}}.$$

□

Consideriamo ora la chiusura dell'insieme dei linguaggi regolari rispetto alla concatenazione.

Teorema 3.9 *Dati due linguaggi regolari L_1 e L_2, la loro concatenazione $L = L_1 \circ L_2$ è un linguaggio regolare.*

Dimostrazione. Si considerino gli automi deterministici

$$\begin{aligned}\mathcal{A}_1 &= \langle \Sigma_1, Q_1, \delta_1, q_{01}, F_1 \rangle \\ \mathcal{A}_2 &= \langle \Sigma_2, Q_2, \delta_2, q_{02}, F_2 \rangle,\end{aligned}$$

che riconoscono, rispettivamente, i linguaggi $L_1 = L(\mathcal{A}_1)$ ed $L_2 = L(\mathcal{A}_2)$. Sia $\mathcal{A} = \langle \Sigma, Q, \delta_N, q_0, F \rangle$ un automa non deterministico tale che:

- $\Sigma = \Sigma_1 \cup \Sigma_2$;
- $Q = Q_1 \cup Q_2$;
- $F = \begin{cases} F_2 & \text{se } \varepsilon \notin L(\mathcal{A}_2), \\ F_1 \cup F_2 & \text{altrimenti;} \end{cases}$
- $q_0 = q_{01}$;
- la δ_N è definita come segue:

$$\begin{aligned}\delta_N(q,a) &= \delta_1(q,a), \forall q \in Q_1 - F_1, a \in \Sigma_1 \\ \delta_N(q,a) &= \delta_1(q,a) \cup \delta_2(q_{02},a), \forall q \in F_1, a \in \Sigma \\ \delta_N(q,a) &= \delta_2(q,a), \forall q \in Q_2, a \in \Sigma_2.\end{aligned}$$

Per costruzione, l'automa $\mathcal{A}$ accetta tutte e sole le stringhe di $L = L_1 \circ L_2$. □

Esempio 3.17 Si consideri l'automa $\mathcal{A}_1$ definito da

$$\mathcal{A}_1 = \langle \{a,b\}, \{q_0, q_1\}, \delta_1, q_0, \{q_1\} \rangle,$$

ed il cui grafo di transizione è quello di Figura 3.16; si consideri inoltre l'automa $\mathcal{A}_2$ definito da

$$\mathcal{A}_2 = \langle \{a,b\}, \{q_2, q_3\}, \delta_2, q_2, \{q_2\} \rangle,$$

ed il cui grafo di transizione è quello di Figura 3.17. Per costruire l'automa $\mathcal{A}$ che effettua la concatenazione $L(\mathcal{A}_2) \circ L(\mathcal{A}_1)$ si procede come segue.

- Si pone $\Sigma = \{a, b\}$.
- Si pone $Q = \{q_0, q_1, q_2, q_3\}$.
- Si sceglie q_2 come stato iniziale.
- Si pone $F = \{q_1\}$ (perché $\mathcal{A}_1$ non accetta la stringa vuota).
- La funzione di transizione si ricava come segue. Per gli stati appartenenti a $Q_2 - F_2$ si ha:

$$\delta_N(q_3, a) = \delta_2(q_3, a).$$

Per gli stati appartenenti ad F_2 si ha:

$$\begin{aligned}\delta_N(q_2, a) &= \emptyset \cup \{\delta_1(q_0, a)\} \\ \delta_N(q_2, b) &= \{\delta_2(q_2, b)\} \cup \{\delta_1(q_0, b)\}.\end{aligned}$$

Per gli stati appartenenti a Q_1 si ha:

$$\begin{aligned}\delta_N(q_0, a) &= \delta_1(q_0, a) \\ \delta_N(q_0, b) &= \delta_1(q_0, b) \\ \delta_N(q_1, a) &= \delta_1(q_1, a).\end{aligned}$$

Il grafo di transizione dell'automa $\mathcal{A}$ è riportato in Figura 3.21.

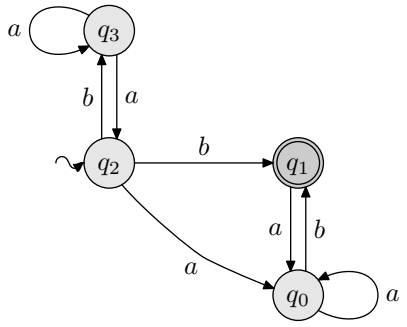


FIGURA 3.21 Automa che accetta la concatenazione dei linguaggi accettati dagli automi di Figura 3.16 e Figura 3.17.

Infine, mostriamo come la classe dei linguaggi regolari sia chiusa anche rispetto all'iterazione.

Teorema 3.10 *Dato un linguaggio regolare L, anche L^* è un linguaggio regolare.*

Dimostrazione. Per dimostrare questa proprietà, si consideri l'automa deterministico

$$\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$$

che riconosce $L = L(\mathcal{A})$. A partire da questo, deriviamo un automa $\mathcal{A}' = \langle \Sigma, Q \cup \{q_0'\}, \delta', q_0', F \cup \{q_0'\} \rangle$, che riconosce $L^* = (L(\mathcal{A}))^*$, ponendo

$$\begin{aligned}\delta'(q, a) &= \delta(q, a), \ \forall q \in Q - F \\ \delta'(q, a) &= \delta(q, a) \cup \delta(q_0, a), \ \forall q \in F \\ \delta'(q_0', a) &= \delta(q_0, a).\end{aligned}$$

Per costruzione l'automa $\mathcal{A}'$ è in grado di accettare tutte e sole le stringhe di L^*. □

Esercizio 3.13 Mostrare che in effetti l'automa $\mathcal{A}'$ definito nella dimostrazione del Teorema 3.10 accetta tutte e sole stringhe di L^*.

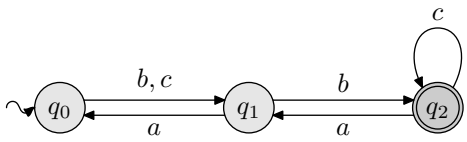


FIGURA 3.22 Esempio di automa.

Esempio 3.18 Si consideri l'automa $\mathcal{A}$ di Figura 3.22.

Per costruire l'automa che riconosce l'iterazione di $L(\mathcal{A})$ si procede come segue.

- Si pone $\Sigma' = \Sigma$.
- Si pone $Q' = Q \cup q_0'$, dove q_0' è il nuovo stato iniziale.
- Si pone $F' = F \cup q_0'$.
- La funzione di transizione è definita al seguente modo. Per gli stati appartenenti a $Q - F$ si ha:

$$\begin{aligned} \delta'(q_0, b) &= \delta(q_0, b) \\ \delta'(q_1, a) &= \delta(q_1, a) \\ \delta'(q_1, b) &= \delta(q_1, b). \end{aligned}$$

Per gli stati appartenenti ad F si ha:

$$\begin{aligned} \delta'(q_2, a) &= \delta(q_2, a) \\ \delta'(q_2, b) &= \delta(q_0, b) \\ \delta'(q_2, c) &= \{\delta(q_2, c)\} \cup \{\delta(q_0, c)\}. \end{aligned}$$

Inoltre si ha

$$\begin{aligned} \delta'(q_0', b) &= \delta(q_0, b) \\ \delta'(q_0', c) &= \delta(q_0, c). \end{aligned}$$

L'automa $\mathcal{A}'$ che si ottiene è riportato in Figura 3.23.

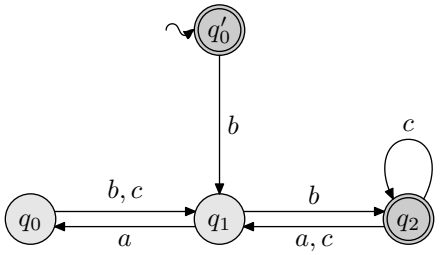


FIGURA 3.23 Iterazione dell'automa di Figura 3.22.

Esercizio 3.14 Dati i linguaggi L_1 e L_2 generati dalle seguenti grammatiche

$$\begin{array}{lcl} S & \longrightarrow & aA \mid bS \mid a \\ A & \longrightarrow & aS \mid bA \end{array} \quad \text{e} \quad \begin{array}{lcl} S & \longrightarrow & bB \mid aS \mid \varepsilon \\ B & \longrightarrow & bS \mid aB \end{array}$$

determinare gli ASFND che accettano $L_1 \cup L_2$, $L_1 \circ L_2$ e L_2^*.

L'insieme dei linguaggi regolari presenta la proprietà di chiusura anche rispetto ad altre operazioni, oltre quelle considerate in questa sezione. Ad esempio, abbiamo già visto che se il linguaggio L è regolare anche il linguaggio delle stringhe riflesse di L è regolare (Teorema 3.2). Il seguente esercizio introduce una ulteriore operazione rispetto a cui i linguaggi regolari sono chiusi.

Esercizio 3.15 Dimostrare che i linguaggi regolari sono chiusi rispetto alla seguente operazione: sia $\Sigma = \{a_1, \ldots, a_n\}$ l'alfabeto e siano $L_1, \ldots, L_n$ n linguaggi regolari. Dato il linguaggio L l'operazione costruisce il linguaggio L' nel seguente modo: per ogni stringa $a_{i_1}, \ldots, a_{i_n} \in L$, L' contiene il linguaggio $L_{i_1} \circ \cdots \circ L_{i_n}$.

3.6 Espressioni regolari e grammatiche regolari

Nelle sezioni precedenti si è visto che le grammatiche di tipo 3 e gli automi a stati finiti definiscono la stessa classe di linguaggi e cioè i linguaggi regolari. Inoltre, si è anche accennato al fatto che le espressioni regolari rappresentano linguaggi regolari. In questa sottosezione la caratterizzazione dei linguaggi regolari verrà completata mostrando formalmente le proprietà che legano i linguaggi regolari e le espressioni regolari.

Teorema 3.11 *Tutti i linguaggi definiti da espressioni regolari sono regolari.*

Dimostrazione. Basta dimostrare che ogni espressione regolare definisce un linguaggio regolare. Ciò appare evidente dalla definizione delle espressioni

regolari, che rappresentano linguaggi a partire dai linguaggi regolari finiti Λ, $\{a\}$, $\{b\}$, ..., mediante le operazioni di unione, concatenazione e iterazione ripetute un numero finito di volte (vedi Tabella 1.3). Poiché è stato dimostrato che per i linguaggi regolari vale la proprietà di chiusura per tutte le operazioni citate (vedi Teoremi 3.6, 3.9 e 3.10), si può concludere che le espressioni regolari definiscono solamente linguaggi regolari. □

Si osservi che applicando in maniera opportuna le operazioni di composizione di ASF descritte nelle dimostrazioni dei Teoremi 3.6, 3.9 e 3.10, è possibile anche, data una espressione regolare r, costruire un ASFND $\mathcal{A}$ tale che $L(\mathcal{A}) = \mathcal{L}(r)$, tale cioè da riconoscere il linguaggio definito da r.

Mostriamo ora l'implicazione opposta, che cioè le espressioni regolari sono sufficientemente potenti da descrivere tutti i linguaggi regolari. Faremo ciò in due modi alternativi, mostrando prima come derivare, a partire da una grammatica $\mathcal{G}$ di tipo 3, una espressione regolare che descrive il linguaggio generato da $\mathcal{G}$. Quindi, mostreremo come derivare, a partire da un ASFD $\mathcal{A}$, una espressione regolare che descrive il linguaggio riconosciuto da $\mathcal{A}$. Entrambi questi risultati implicano che tutti i linguaggi regolari sono definibili mediante espressioni regolari.

Teorema 3.12 *Data una grammatica $\mathcal{G}$ di tipo 3, esiste una espressione regolare r tale che $L(\mathcal{G}) = \mathcal{L}(r)$, che descrive cioè il linguaggio generato da $\mathcal{G}$.*

Dimostrazione. Consideriamo una grammatica $\mathcal{G}$ di tipo 3 ed il linguaggio L da essa generato, che per semplicità assumiamo non contenga la stringa vuota ε. Se così non fosse, applichiamo le considerazioni seguenti al linguaggio $L - \{\varepsilon\}$, anch'esso regolare: una volta derivata un'espressione regolare r che lo definisce, l'espressione regolare che definisce L sarà chiaramente $r + \varepsilon$.

Trascurando alcuni dettagli formali e limitandoci alla sola intuizione, alla grammatica $\mathcal{G}$ possiamo far corrispondere un sistema di equazioni su espressioni regolari, nel seguente modo. Innanzi tutto estendiamo con variabili $A, \ldots, Z$ il linguaggio delle espressioni regolari, associando una variabile ad ogni non terminale in $\mathcal{G}$. Tali variabili potranno assumere valori nell'insieme delle espressioni regolari. Pertanto, se il valore dell'espressione regolare A è, ad esempio, $(00 + 11)^*$, l'espressione estesa $A(0 + 1)$ indica tutte le stringhe del tipo $(00 + 11)^*(0 + 1)$.

Raggruppiamo ora tutte le produzioni che presentano a sinistra lo stesso non terminale. Per ogni produzione del tipo

$$A \longrightarrow a_1B_1 \mid a_2B_2 \mid \ldots \mid a_nB_n \mid b_1 \mid \ldots \mid b_m$$

creiamo un'equazione del tipo

$$A = a_1B_1 + a_2B_2 + \ldots + a_nB_n + b_1 + \ldots + b_m.$$

Come si vede, un'equazione può essere impostata come l'attribuzione ad una variabile del valore di un'espressione regolare "estesa".

Risolvere un sistema di equazioni su espressioni regolari estese significa dunque individuare i valori, cioè le espressioni regolari normali, prive delle variabili che definiscono a loro volta espressioni regolari, che, una volta sostituiti alle variabili, soddisfano il sistema di equazioni. È facile osservare che ad una grammatica regolare corrisponde un sistema di *equazioni lineari destre*, in cui ogni monomio contiene una variabile a destra di simboli terminali.

Ad esempio, la grammatica

$$\begin{array}{lcl} A & \longrightarrow & aA \mid bB \\ B & \longrightarrow & bB \mid c \end{array}$$

corrisponde al sistema di equazioni

$$\begin{cases} A &= aA + bB \\ B &= bB + c. \end{cases}$$

Per risolvere un sistema di tale genere utilizzeremo, oltre alle trasformazioni algebriche applicabili sulle operazioni di unione e concatenazione (distributività, fattorizzazione, ecc.), le seguenti due regole.

- *Sostituzione* di una variabile con un'espressione regolare estesa. Con riferimento all'esempio precedente abbiamo

$$\begin{cases} A &= aA + b(bB + c) = aA + bbB + bc \\ B &= bB + c. \end{cases}$$

- *Eliminazione della ricursione*. L'equazione $B = bB + c$ si risolve in $B = b^*c$. Infatti, sostituendo a destra e sinistra abbiamo

$$b^*c = b(b^*c) + c = b^+c + c = (b^+ + \varepsilon)c = b^*c.$$

  Più in generale abbiamo che un'equazione del tipo

$$A = \alpha_1 A + \alpha_2 A + \ldots + \alpha_n A + \beta_1 + \beta_2 + \ldots + \beta_m$$

  si risolve in

$$A = (\alpha_1 + \alpha_2 + \ldots + \alpha_n)^*(\beta_1 + \beta_2 + \ldots + \beta_m),$$

  dove $\alpha_1, \ldots, \alpha_n$, β_1, $\ldots$, β_m sono espressioni regolari estese.

Applicando le due regole suddette possiamo risolvere ogni sistema di equazioni (lineari destre) su espressioni regolari estese corrispondente ad una grammatica, e quindi individuare le espressioni regolari che corrispondono al linguaggio generato dai rispettivi non terminali della grammatica e, in particolare, al linguaggio generato a partire dall'assioma. □

Esempio 3.19 Data la grammatica regolare

$$\begin{array}{lcl} A_0' & \longrightarrow & \varepsilon \mid aA_1 \mid a \\ A_0 & \longrightarrow & aA_1 \mid a \\ A_1 & \longrightarrow & bA_3 \mid bA_2 \\ A_2 & \longrightarrow & aA_2 \mid bA_0 \mid b \\ A_3 & \longrightarrow & bA_3 \mid aA_2. \end{array}$$

si ottiene il seguente sistema lineare

$$\left\{ \begin{array}{lcl} A_0' & = & \varepsilon + aA_1 + a \\ A_0 & = & aA_1 + a \\ A_1 & = & bA_3 + bA_2 \\ A_2 & = & aA_2 + bA_0 + b \\ A_3 & = & bA_3 + aA_2 \end{array} \right.$$

che può essere risolto tramite i seguenti passaggi.

$$\left\{ \begin{array}{lcl} A_0' & = & \varepsilon + aA_1 + a \\ A_0 & = & aA_1 + a \\ A_1 & = & bA_3 + bA_2 \\ A_2 & = & aA_2 + bA_0 + b \\ A_3 & = & b^*aA_2 \end{array} \right.$$

per eliminazione della ricursione su A_3.

$$\left\{ \begin{array}{lcl} A_0' & = & \varepsilon + aA_1 + a \\ A_0 & = & aA_1 + a \\ A_1 & = & bA_3 + bA_2 \\ A_2 & = & a^*(bA_0 + b) \\ A_3 & = & b^*aA_2 \end{array} \right.$$

per eliminazione della ricursione su A_2.

$$\left\{ \begin{array}{lcl} A_0' & = & \varepsilon + aA_1 + a \\ A_0 & = & aA_1 + a \\ A_1 & = & bA_3 + bA_2 \\ A_2 & = & a^*(bA_0 + b) \\ A_3 & = & b^*aa^*(bA_0 + b) \end{array} \right.$$

per sostituzione di A_2 nell'equazione relativa ad A_3.

$$\begin{cases} A'_0 &= \varepsilon + aA_1 + a \\ A_0 &= aA_1 + a \\ A_1 &= b(b^*aa^*(bA_0 + b)) + b(a^*(bA_0 + b)) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per sostituzione di A_2 e A_3 nell'equazione relativa ad A_1.

$$\begin{cases} A'_0 &= \varepsilon + aA_1 + a \\ A_0 &= aA_1 + a \\ A_1 &= b(b^*aa^* + a^*)(bA_0 + b) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per fattorizzazione nell'equazione relativa ad A_1.

$$\begin{cases} A'_0 &= \varepsilon + aA_1 + a \\ A_0 &= a(b(b^*aa^* + a^*)(bA_0 + b)) + a \\ A_1 &= b(b^*aa^* + a^*)(bA_0 + b) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per sostituzione di A_1 nell'equazione relativa ad A_0.

$$\begin{cases} A'_0 &= \varepsilon + aA_1 + a \\ A_0 &= ab(b^*aa^* + a^*)bA_0 + ab(b^*aa^* + a^*)b + a \\ A_1 &= b(b^*aa^* + a^*)(bA_0 + b) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per fattorizzazione nell'equazione relativa ad A_0.

$$\begin{cases} A'_0 &= \varepsilon + aA_1 + a \\ A_0 &= (ab(b^*aa^* + a^*)b)^*(ab(b^*aa^* + a^*)b + a) \\ A_1 &= b(b^*aa^* + a^*)(bA_0 + b) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per eliminazione della ricursione su A_0.

$$\begin{cases} A_0' &= a(b(b^*aa^* + a^*)(bA_0 + b)) + a + \varepsilon \\ A_0 &= (ab(b^*aa^* + a^*)b)^*(ab(b^*aa^* + a^*)b + a) \\ A_1 &= b(b^*aa^* + a^*)(bA_0 + b) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per sostituzione di A_1 nell'equazione relativa ad A_0'.

$$\begin{cases} A_0' &= ab(b^*aa^* + a^*)(b((ab(b^*aa^* + a^*)b)^*(ab(b^*aa^* + a^*)b + a)) \\ & \quad +b) + a + \varepsilon \\ A_0 &= (ab(b^*aa^* + a^*)b)^*(ab(b^*aa^* + a^*)b + a) \\ A_1 &= b(b^*aa^* + a^*)(bA_0 + b) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per sostituzione di A_0 nell'equazione relativa ad A_0'.

$$\begin{cases} A_0' &= ((ab(b^*a^* + \varepsilon)a^*b)^* + \varepsilon)(ab(b^*a + \varepsilon)a^*b + a) + \varepsilon \\ A_0 &= (ab(b^*aa^* + a^*)b)^*(ab(b^*aa^* + a^*)b + a) \\ A_1 &= b(b^*aa^* + a^*)(bA_0 + b) \\ A_2 &= a^*(bA_0 + b) \\ A_3 &= b^*aa^*(bA_0 + b) \end{cases}$$

per fattorizzazione nell'equazione relativa ad A_0'.

Nel caso di ε-produzioni presenti nella grammatica regolare conviene operare delle sostituzioni per eliminarle preliminarmente, mantenendo al più una produzione $S \longrightarrow \varepsilon$, dove S è l'assioma.

Esempio 3.20 Data la grammatica regolare avente le produzioni

$$\begin{aligned} S &\longrightarrow aS \mid bA \mid \varepsilon \\ A &\longrightarrow aA \mid bS \mid \varepsilon \end{aligned}$$

eliminando la produzione $A \longrightarrow \varepsilon$ la si può trasformare nella grammatica regolare equivalente

$$\begin{aligned} S &\longrightarrow aS \mid bA \mid \varepsilon \mid b \\ A &\longrightarrow aA \mid bS \mid a. \end{aligned}$$

Per ricavare l'espressione regolare corrispondente abbiamo:

$A = a^*(bS + a)$ eliminando la ricursione su A

e, per sostituzione,

$S = (a + ba^*b)S + ba^*a + b + \varepsilon.$

Eliminando anche la ricursione rispetto ad S si ottiene

$S = (a + ba^*b)^*(ba^*a + b + \varepsilon).$

Esercizio 3.16 Si consideri la seguente grammatica:

$$\begin{array}{lcl} A & \longrightarrow & aB \mid bC \mid a \\ B & \longrightarrow & aA \mid bD \mid b \\ C & \longrightarrow & ab \mid aD \mid a \\ D & \longrightarrow & aC \mid bB \mid b \end{array}$$

che genera le stringhe contenenti un numero dispari di a o un numero dispari di b. Si costruisca l'espressione regolare corrispondente.

Vediamo ora come sia possibile costruire una espressione regolare corrispondente al linguaggio accettato da un automa a stati finiti deterministico.

Teorema 3.13 *Dato un ASFD $\mathcal{A}$, esiste una espressione regolare r tale che $L(\mathcal{A}) = \mathcal{L}(r)$, che descrive cioè il linguaggio riconosciuto da $\mathcal{A}$.*

Dimostrazione. Sia $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ un ASFD e sia L il linguaggio da esso riconosciuto. Assumiamo, per semplicità e senza perdere generalità che $F = \{q_F\}$.

Sia $n = \mid Q \mid$ e sia $\langle q_0, \ldots, q_{n-1} \rangle$ un qualunque ordinamento degli stati tale che $q_{n-1} = q_F$. Definiamo ora come R^k_{ij}, $0 \leq i, j \leq n-1$, $k \geq \max(i, j)$, l'insieme delle stringhe tali da portare $\mathcal{A}$ da q_i a q_j senza transitare per nessuno stato q_h con $h \geq k$.

Abbiamo cioè che $x = a_1, \ldots, a_m \in R^k_{ij}$ se e solo se:

1. $\overline{\delta}(q_i, x) = q_j$;
2. Se $\overline{\delta}(q_i, a_1 \ldots a_l) = q_{i_l}$ allora $i_l < k$, per $1 \leq l \leq m-1$.

Osserviamo che, per $k = 1$, si ha:

$$R^1_{ij} = \begin{cases} \cup\{a\} & \text{tali che } \delta(q_i, a) = q_j, \text{ se ne esiste almeno uno;} \\ \emptyset & \text{altrimenti.} \end{cases}$$

Per $k > 1$ possiamo osservare che se $x \in R^{k+1}_{ij}$ è una stringa che conduce da q_i a q_j senza transitare per nessuno stato q_h con $h \geq k+1$, possono verificarsi due casi:

1. x conduce da q_i a q_j senza transitare per q_k, dal che deriva che $x \in R^k_{ij}$.
2. x conduce da q_i a q_j transitando per q_k. In tal caso la sequenza degli stati attraversati può essere divisa in varie sottosequenze:
   (a) una prima sequenza, da q_i a q_k senza transitare per nessuno stato q_h con $h > k$, la corrispondente sottostringa di x apparterrà quindi a $x \in R^k_{ik}$;

(b) $r \geq 0$ sequenze, ognuna delle quali inizia e termina in q_k senza transitare per nessuno stato q_h con $h \geq k$, la corrispondente sottostringa di x apparterrà quindi a $x \in R^k_{kk}$;

(c) una sequenza finale, da q_k a q_j senza transitare per nessuno stato q_h con $h \geq k$, la corrispondente sottostringa di x apparterrà quindi a $x \in R^k_{kj}$.

In conseguenza di ciò, deriva la relazione $R^{k+1}_{ij} = R^k_{ij} \cup R^k_{ik} \circ (R^k_{kk})^* \circ R^k_{kj}$.

Dalle osservazioni precedenti deriva che è possibile costruire tutti gli insiemi R^k_{ij} a partire da $k = 1$ e derivando poi, man mano i successivi. Osserviamo anche che $L = R^n_{0(n-1)}$.

Notiamo ora che ogni insieme di stringhe R^k_{ij} può essere descritto per mezzo di una opportuna espressione regolare r^k_{ij}, infatti abbiamo che, per $k = 1$,

$$r^1_{ij} = \begin{cases} a_{i_1} + \ldots + a_{i_l} & \text{dove } \delta(q_i, a_{i_k}) = q_j, k = 1, \ldots, l; \\ \varepsilon & \text{se } l = 0. \end{cases}$$

Mentre, per $k > 1$, abbiamo che, dalla relazione tra R^{k+1}_{ij}, R^k_{ik}, R^k_{kk} e R^k_{kj}, deriva che $r^{k+1}_{ij} = r^k_{ij} + r^k_{ik}(r^k_{kk})^* r^k_{kj}$.

Quindi, il linguaggio L sarà descritto dall'espressione regolare $r^n_{0(n-1)}$. □

Esempio 3.21 Consideriamo ad esempio l'ASFD di Figura 3.14. Applicando la costruzione del Teorema 3.13 con l'ordinamento $q_1 = q_0, q_2 = q_1, q_3 = q_3, q_4 = q_2$ tra gli stati, abbiamo che:

$r^0_{00} = \emptyset; r^0_{01} = a; r^0_{02} = b; r^0_{03} = \emptyset;$

$r^0_{10} = b; r^0_{11} = \emptyset; r^0_{12} = \emptyset; r^0_{13} = a;$

$r^0_{20} = \emptyset; r^0_{21} = \emptyset; r^0_{22} = a + b; r^0_{23} = \emptyset;$

$r^0_{30} = b; r^0_{31} = \emptyset; r^0_{32} = \emptyset; r^0_{33} = a;$

$r^1_{00} = \emptyset; r^1_{01} = a; r^1_{02} = b; r^1_{03} = \emptyset;$

$r^1_{10} = b; r^1_{11} = ba; r^1_{12} = bb; r^1_{13} = a;$

$r^1_{20} = \emptyset; r^1_{21} = \emptyset; r^1_{22} = a + b; r^1_{23} = \emptyset;$

$r^1_{30} = b; r^1_{31} = ba; r^1_{32} = bb; r^1_{33} = a;$

$r^2_{00} = a(ba)^*b; r^2_{01} = a + a(ba)^*ba; r^2_{02} = b + a(ba)^*bb; r^2_{03} = a(ba)^*a;$

$r^2_{10} = b + ba(ba)^*b; r^2_{11} = ba + ba(ba)^*ba; r^2_{12} = bb + ba(ba)^*bb; r^2_{13} = a + ba(ba)^*a;$

$r_{20}^2 = \emptyset; r_{21}^2 = \emptyset; r_{22}^2 = a + b; r_{23}^2 = \emptyset;$

$r_{30}^2 = b + ba(ba)^*b; r_{31}^2 = ba + ba(ba)^*ba; r_{32}^2 = bb + ba(ba)^*bb; r_{33}^2 = a + ba(ba)^*a;$

$\vdots$

Esercizio 3.17 Completare la costruzione delle espressioni regolari r_{ij}^k dell'Esempio 3.21 e mostrare che in effetti l'automa riconosce il linguaggio $a(a + ba)^*a$.

Sia dal Teorema 3.12 che dal Teorema 3.13 è quindi immediato derivare il seguente corollario.

Corollario 3.14 *La classe dei linguaggi regolari coincide con la classe dei linguaggi rappresentati da espressioni regolari.*

In conclusione, sulla base dei risultati visti precedentemente e delle proprietà di chiusura studiate nella Sezione 3.5, è possibile dimostrare il seguente teorema.

Teorema 3.15 *La classe dei linguaggi regolari su di un alfabeto* $\Sigma = \{a_1, \ldots, a_n\}$ *costituisce la più piccola classe contenente il linguaggio vuoto* Λ *e tutti i linguaggi* $\{a_1\}, \ldots, \{a_n\}$*, che sia chiusa rispetto ad unione, concatenazione e iterazione.*

Dimostrazione. La dimostrazione viene lasciata come esercizio. □

Esercizio 3.18 Dimostrare il Teorema 3.15.

3.7 Predicati decidibili sui linguaggi regolari

Le proprietà dei linguaggi regolari viste nelle sezioni precedenti possono essere sfruttate per dimostrare la verità o falsità di predicati relativi a specifici linguaggi regolari. I primi e più significativi di tali predicati sono quelli che fanno riferimento alla cardinalità di un linguaggio regolare.

Teorema 3.16 *È possibile decidere se il linguaggio accettato da un dato automa a stati finiti è vuoto, finito oppure infinito.*

Dimostrazione. Mostriamo innanzi tutto che se un automa con n stati accetta qualche stringa, allora accetta una stringa di lunghezza inferiore ad n. Infatti, detta z la stringa più breve accettata dall'automa, se fosse $\mid z \mid \geq n$ allora, in base al pumping lemma (Teorema 3.4), z potrebbe essere scritta come uvw e anche la stringa uw, più breve di z, sarebbe accettata dall'automa.

Dato quindi un automa $\mathcal{A}$ con n stati, per decidere se $L(\mathcal{A}) = \Lambda$ basta verificare se esso accetta almeno una delle $\sum_{i=0}^{n-1} \mid \Sigma \mid^i$ stringhe di lunghezza inferiore ad n.

In modo analogo, mostriamo che $L(\mathcal{A})$ è infinito se e solo se $\mathcal{A}$ accetta una stringa z, con $n \leq \mid z \mid < 2n$. Infatti se $\mathcal{A}$ accetta una z lunga almeno n allora, in virtù del pumping lemma, accetta infinite stringhe. D'altra parte, se $L(\mathcal{A})$ è infinito, allora esiste $z \in L(\mathcal{A})$ tale che $\mid z \mid \geq n$. Se $\mid z \mid < 2n$ l'asserto è provato; se invece $\mid z \mid \geq 2n$ allora, per il pumping lemma, z può essere scritta come uvw, con $1 \leq \mid v \mid \leq n$ e $uw \in L(\mathcal{A})$ e da ciò segue che esiste una stringa $x = uv \in L(\mathcal{A})$ tale che $n \leq \mid x \mid < 2n$.

Per decidere dunque se $L(\mathcal{A})$ è infinito basta verificare se $\mathcal{A}$ accetta una delle $\sum_{i=n}^{2n-1} \mid \Sigma \mid^i$ stringhe di lunghezza compresa fra n e $2n-1$. □

In definitiva per verificare se il linguaggio accettato da un automa sia finito, vuoto o infinito, si osserva il comportamento dell'automa su tutte le stringhe di lunghezza non superiore a $2n$:

- se nessuna viene accettata allora il linguaggio è vuoto;
- se tutte quelle accettate hanno lunghezza inferiore ad n, allora il linguaggio è finito;
- se tra le stringhe accettate ve ne sono alcune la cui lunghezza sia superiore o uguale ad n allora il linguaggio è infinito.

Si osservi che questo metodo, seppur utile per mostrare la decidibilità dei problemi considerati, risulta fortemente inefficiente, in quanto richiede di esaminare una quantità di stringhe pari a $\sum_{i=0}^{2n-1} \mid \Sigma \mid^i = O(2^n)$, vale a dire esponenziale rispetto ad n.

Metodi più efficienti per determinare se $L(\mathcal{A})$ è vuoto, finito o infinito considerano la struttura della funzione di transizione di $\mathcal{A}$ e, in particolare, il relativo grafo di transizione. Non è difficile rendersi conto che valgono le proprietà seguenti:

a) Se non esiste un cammino nel grafo dallo stato iniziale q_0 ad un qualche stato finale $q \in F$, allora $\mathcal{L}(\mathcal{A}) = \emptyset$ (e viceversa).

b) Se esiste un cammino nel grafo dallo stato iniziale q_0 ad un qualche stato finale $q \in F$ che attraversa almeno due volte uno stesso stato, e quindi include un ciclo, allora $\mathcal{L}(\mathcal{A})$ è infinito (e viceversa).

Esercizio 3.19 Dimostrare formalmente le proprietà a) e b).

Quindi, è possibile verificare se $L(\mathcal{A})$ è vuoto o infinito semplicemente verificando se, nel grafo di transizione di $\mathcal{A}$, esistono cammini tra particolari coppie di nodi, nel primo caso, o cicli che sono inclusi in tali cammini, nel

secondo: tali proprietà, su un grafo orientato, sono determinabili in tempo lineare (nel numero di nodi e di archi) applicando opportunamente i classici metodi di visita di grafi.

Esercizio 3.20 Scrivere un algoritmo che, dato un automa, determini, per mezzo di una visita del relativo grafo di transizione, se il linguaggio accettato è vuoto, infinito o finito.

Il Teorema 3.16 consente di dimostrare altre proprietà di decidibilità sui linguaggi regolari.

Teorema 3.17 *Il problema dell'equivalenza di due linguaggi regolari è decidibile.*

Dimostrazione. Per dimostrare l'equivalenza di due linguaggi, basta dimostrare che la loro intersezione coincide con la loro unione, cioè che la loro differenza simmetrica (l'unione dell'intersezione del primo con il complemento del secondo e dell'intersezione del secondo con il complemento del primo) è il linguaggio vuoto. Facendo riferimento agli automi $\mathcal{A}_1$ e $\mathcal{A}_2$ che riconoscono L_1 ed L_2, basta dimostrare che

$$\begin{aligned} L(\mathcal{A}_1) \triangle L(\mathcal{A}_1) &= \left(L(\mathcal{A}_1) \cap \overline{L(\mathcal{A}_2)}\right) \cup \left(\overline{L(\mathcal{A}_1)} \cap L(\mathcal{A}_2)\right) \\ &= \left(\overline{\overline{L(\mathcal{A}_1)} \cup L(\mathcal{A}_2)}\right) \cup \left(\overline{L(\mathcal{A}_1) \cup \overline{L(\mathcal{A}_2)}}\right) \\ &= \emptyset. \end{aligned}$$

dove il simbolo Δ denota l'operazione di differenza simmetrica tra insiemi.

Tale proprietà è verificabile considerando il corrispondente automa, che è possibile costruire a partire da $\mathcal{A}_1$ e $\mathcal{A}_2$ applicando le tecniche, illustrate in Sezione 3.5, per derivare automi che riconoscano il linguaggio unione ed il linguaggio complemento. □

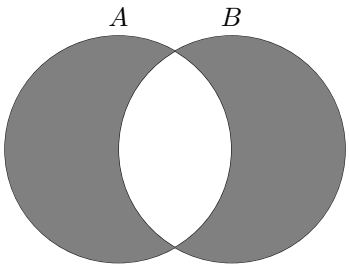


FIGURA 3.24 Differenza simmetrica tra insiemi (ombreggiata).

La decidibilità dell'equivalenza di due linguaggi è caratteristica dei soli linguaggi regolari, nell'ambito della gerarchia di Chomsky. Infatti, tale proprietà è indecidibile già per i linguaggi di tipo 2. Nel caso dei linguaggi regolari essa è molto importante anche perchè consente di realizzare un banale algoritmo per minimizzare il numero degli stati di un automa. Uno stesso linguaggio, infatti, può venire accettato da automi diversi, per cui si può desiderare di costruire l'automa che riconosca lo stesso linguaggio di un automa dato, ma che abbia il minimo numero di stati necessario all'accettazione. Dato un automa ad n stati che riconosce un certo linguaggio L, è possibile, in linea di principio, generare *tutti* gli automi con $m < n$ stati e controllare se qualcuno è equivalente all'automa dato.

In pratica, per costruire l'ASF minimo che riconosce un dato linguaggio non si procede in tal modo ma si procede, in modo più efficiente, partizionando l'insieme Q degli stati in classi di equivalenza, come illustrato nella sezione che segue.

3.8 Il teorema di Myhill-Nerode

Dato un linguaggio L definiamo la relazione binaria R_L su Σ^* come segue:

$$xR_Ly \iff (\forall z \in \Sigma^* \;\; xz \in L \iff yz \in L).$$

La R_L è palesemente una relazione di congruenza (vedi Definizione 1.36) e il teorema che segue stabilisce una condizione necessaria e sufficiente sull'indice di tale relazione affinché L sia regolare.

Teorema 3.18 (Myhill-Nerode) *Un linguaggio $L \subseteq \Sigma^*$ è regolare se e solo se la relazione R_L ha indice finito.*

Dimostrazione. Supponendo che L sia regolare, sia $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ un ASFD che lo riconosce. Consideriamo la relazione binaria $R_{\mathcal{A}}$ su Σ^* definita come

$$xR_{\mathcal{A}}y \iff \overline{\delta}(q_0, x) = \overline{\delta}(q_0, y),$$

in cui due stringhe sono in relazione se e solo se, partendo dallo stato iniziale, esse portano l'automa nello stesso stato. Come è possibile immediatamente verificare, questa relazione gode delle seguenti proprietà.

i) $R_{\mathcal{A}}$ è una relazione di equivalenza;

ii) $xR_{\mathcal{A}}y \implies \forall z \in \Sigma^* \; xzR_{\mathcal{A}}yz$;

iii) $R_{\mathcal{A}}$ ha indice finito, pari al più al numero di stati dell'automa $\mathcal{A}$, poiché ad ogni stato raggiungibile da q_0 si può associare una differente classe di equivalenza;

iv) $R_{\mathcal{A}}$ induce un partizionamento di Σ^* in un numero di classi non inferiore al numero di classi dovute ad R_L, perché se $xR_{\mathcal{A}}y$ allora, per la definizione di $R_{\mathcal{A}}$ e per la proprietà ii),

$$\overline{\delta}(q_0, xz) = \overline{\delta}(q_0, yz) \in Q, \ \forall z \in \Sigma^*,$$

per cui in particolare $xz \in L \Leftrightarrow yz \in L$ e quindi xR_Ly.

Dato che, per la proprietà iv) la $R_{\mathcal{A}}$ costituisce al più un raffinamento della R_L, cioè

$$\text{ind}(R_{\mathcal{A}}) \geq \text{ind}(R_L)$$

e la $R_{\mathcal{A}}$ ha indice finito, allora anche la R_L ha indice finito.

Supponiamo ora che sia data la relazione R_L ed essa abbia indice finito. Possiamo allora definire un automa a stati finiti deterministico $\mathcal{A}' = \langle \Sigma, Q', \delta', q_0', F' \rangle$ nel seguente modo: Q' è costruito in maniera tale da associare a ogni classe $[x]$ di Σ^* uno stato $q_{[x]}$, $q_0' = q_{[\varepsilon]}$, $F' = \{q_{[x]} | x \in L\}$, $\delta'(q_{[x]}, a) = q_{[xa]}$ $\forall a \in \Sigma$. È possibile mostrare facilmente, per induzione sulla lunghezza delle stringhe, che $\mathcal{A}'$ riconosce L, che pertanto è un linguaggio regolare. □

Il teorema di Myhill-Nerode consente di ribadire la proprietà, già incontrata precedentemente, secondo cui il potere computazionale di un automa a stati finiti è limitato proprio alla classificazione di stringhe in un numero finito di classi: proprio da ciò discende l'impossibilità di usare un automa a stati finiti per riconoscere linguaggi del tipo a^nb^n per i quali si rende necessaria, in un certo senso, la capacità di contare.

Esercizio 3.21 Dimostrare che per il linguaggio $L = \{a^nb^n \mid n \geq 1\}$ la relazione R_L non ha indice finito.

Il teorema di Myhill-Nerode ha anche un'altra importante applicazione. Esso può essere utilizzato per minimizzare un dato ASFD $\mathcal{A}$, vale a dire per costruire l'automa equivalente avente il minimo numero di stati: l'automa $\mathcal{A}'$ introdotto nella dimostrazione del Teorema 3.18 gode infatti di tale proprietà, valendo le seguenti disuguaglianze:

$$|Q| \ \geq \ \text{ind}(R_{\mathcal{A}}) \ \geq \ \text{ind}(R_L) = |Q'|,$$

il che mostra che $\mathcal{A}$ non può avere un numero di stati inferiore a quello degli stati di $\mathcal{A}'$. Tale automa $\mathcal{A}'$ gode inoltre della ulteriore proprietà di essere unico (a meno di una ridenominazione degli stati), proprietà che tuttavia non proveremo in questa sede.

3.8.1 Minimizzazione di automi a stati finiti

Dalla dimostrazione del Teorema 3.18 scaturisce un semplice algoritmo di minimizzazione di un automa a stati finiti deterministico $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$. Denotiamo con $\equiv$ la relazione su Q^2 tale che

$$q_i \equiv q_j \iff (\forall x \in \Sigma^* \ \overline{\delta}(q_i, x) \in F \iff \overline{\delta}(q_j, x) \in F).$$

Si tratta, come è ovvio, di una relazione di equivalenza. Gli stati q_i, q_j tali che $q_i \equiv q_j$ sono detti *indistinguibili*; nel caso invece esista una stringa $x \in \Sigma^*$ per cui $\overline{\delta}(q_i, x) \in F$ e $\overline{\delta}(q_j, x) \in Q - F$ (o viceversa) diremo che q_i e q_j sono *distinguibili* tramite x.

Per minimizzare un ASFD è in pratica sufficiente individuare tutte le coppie di stati indistinguibili sfruttando un semplice algoritmo di marcatura delle coppie distinguibili. Ciò fatto, l'automa minimo è ottenuto "unificando" gli stati equivalenti, eliminando quelli non raggiungibili e modificando opportunamente la funzione di transizione.

Le considerazioni che seguono vengono svolte assumendo che tutti gli stati di $\mathcal{A}$ siano raggiungibili dallo stato iniziale, altrimenti sarà necessario effettuare un passo preliminare di eliminazione degli stati irraggiungibili.

Esercizio 3.22 Costruire un algoritmo che, dato un ASFD $\mathcal{A}$, individui tutti gli stati irraggiungibili dallo stato iniziale. Si noti che il miglior algoritmo per questo problema richiede un numero di passi proporzionale a $|Q| \cdot |\Sigma|$.

Per marcare le coppie di stati distinguibili conviene utilizzare una tabella contenente una casella per ciascuna coppia (non ordinata) di stati di Q. Le caselle vengono usate per marcare le coppie di stati distinguibili e per elencare, in una lista associata, tutte le coppie che dovranno essere marcate qualora la coppia a cui è associata la casella venga marcata.

La procedura inizia con la marcatura delle coppie distinguibili tramite la stringa ε (tutte e sole le coppie costituite da uno stato finale e da uno non finale). Quindi, per ogni coppia (p, q) non ancora marcata, si considerano, per ogni $a \in \Sigma$, tutte le coppie (r, s), con $r = \delta(p,)$ e $s = \delta(q, a)$.

Se nessuna delle coppie (r, s) è già stata riconosciuta come coppia di stati distinguibili allora si inserisce (p, q) nella lista associata ad ognuna di esse, eccetto che nel caso in cui tale coppia risulti essere nuovamente (p, q). Altrimenti p e q vengono riconosciuti distinguibili e la corrispondente casella viene marcata; qualora questa contenga una lista di coppie si procede (ricorsivamente) con la marcatura delle relative caselle.

In maniera più formale possiamo considerare il seguente algoritmo di marcatura. Non è difficile dimostrare che, per ogni coppia di stati p, q di Q, essi sono distinguibili se e solo se, al termine dell'esecuzione dell'Algoritmo 3.1, la casella relativa alla coppia (p, q) risulta marcata: la dimostrazione è lasciata al lettore come esercizio (vedi Esercizio 3.23).

input automa a stati finiti $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$;
output coppie di stati distinguibili di Q;
begin
  for $p \in F$ and $q \in Q - F$ **do**
    marca (p, q) e (q, p);
  for each coppia non marcata di stati distinti **do**
    if $\exists a \in \Sigma :\ (\delta(p, a), \delta(q, a))$ distinguibili **then**
      begin
        marca (p, q);
        marca ricorsivamente tutte le coppie non ancora marcate sulla lista
          di (p, q) e su quelle delle coppie marcate a questo passo
      end
    else
      for $a \in \Sigma$ **do**
        if $\delta(p, a) \neq \delta(q, a)$**and** $(p, q) \neq (\delta(p, a), \delta(q, a))$ **then**
            aggiungi (p, q) alla lista di $(\delta(p, a), \delta(q, a))$
end.

Algoritmo 3.1: Marca Stati Distinguibili

Esercizio 3.23 Dato un automa a stati finiti deterministico $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, dimostrare che due stati p e q appartenenti a Q sono distinguibili se e solo se l'Algoritmo 3.1 marca la casella relativa alla coppia (p, q).
[Suggerimento: Effettuare una induzione sulla lunghezza della minima stringa che consente di distinguere i due stati distinguibili.]

Una volta identificate le coppie di stati indistinguibili, ricordando che la relazione di indistinguibilità è una relazione di equivalenza, l'automa equivalente con il minimo numero di stati è dato evidentemente da $\mathcal{A}' = \langle \Sigma, Q', \delta', q_0', F' \rangle$, in cui:

- Q' è costruito selezionando uno ed un solo stato di Q (rappresentante) per ogni insieme di stati indistinguibili;
- F' è costituito da tutti i rappresentanti appartenenti ad F;
- δ' è ottenuta da δ mediante restrizione al dominio $Q' \times \Sigma$ ed inoltre, per ogni $\delta(q_i, a) = q_j$, con $q_i \in Q'$ e $q_j \in Q$, poniamo $\delta'(q_i, a) = q_k$, dove $q_k \in Q'$ è il rappresentante dell'insieme di stati indistingubili che include q_j (chiaramente, se $q_j \in Q'$ allora è esso stesso un rappresentante e dunque $q_k = q_j$).

Esempio 3.22 Si consideri l'automa deterministico dato in Figura. 3.25 La procedura di minimizzazione inizia con la costruzione di una tabella contenente una cella per ciascuna coppia (non ordinata) di stati. Le celle vengono usate per marcare le

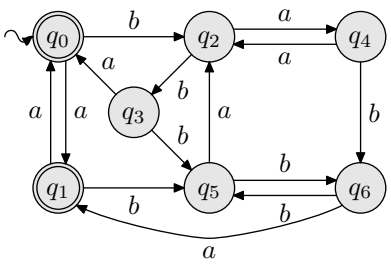


FIGURA 3.25 Automa a stati finiti da minimizzare.

coppie di stati (q_i, q_j) tali che esiste una stringa x per cui $\overline{\delta}(q_i, x) \in F$ e $\overline{\delta}(q_j, x) \notin F$, o viceversa ($q_i$ e q_j si dicono in tal caso *distinguibili*). A tal fine si iniziano a marcare le coppie (q_i, q_j) tali che $q_i \in F$ e $q_j \notin F$ o $q_i \notin F$ e $q_j \in F$. In Figura 3.26 queste coppie sono le 10 coppie per cui $q_i \in \{q_0, q_1\}$ e $q_j \in \{q_2, q_3, q_4, q_5, q_6\}$: le celle corrispondenti vengono quindi marcate tramite una X, come mostrato in Figura 3.26. L'applicazione

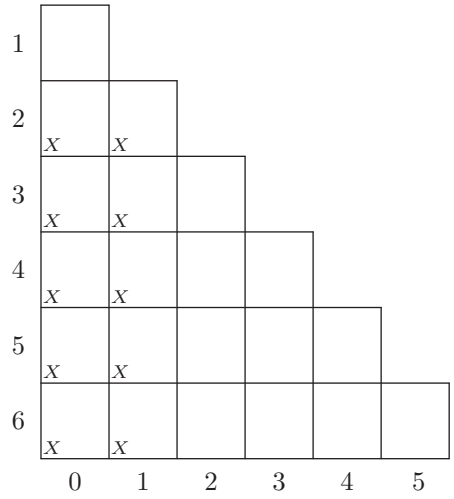


FIGURA 3.26 Marcatura delle coppie di stati immediatamente distinguibili.

dell'Algoritmo 3.1 porta inizialmente a distinguere gli stati q_0 e q_1, che sono finali, da tutti gli altri: da ciò risulta la situazione in Figura 3.26.

Osserviamo ora il comportamento dell'algoritmo assumendo che esso consideri le coppie di stati scandendo le celle da sinistra verso destra e dall'alto verso il basso.

La prima coppia (q_0, q_1) risulta distinguibile se lo è la coppia (q_2, q_5), in quanto

$\delta(q_0, b) = q_2$ e $\delta(q_1, b) = q_5$: quindi, l'elemento $(0, 1)$ viene inserito nella lista associata alla cella $(2, 5)$.

La coppia successiva, (q_2, q_3), risulta distinguibile in quanto $\delta(q_2, a) = q_4$, $\delta(q_3, a) = q_0$, e la coppia (q_0, q_4) è distinguibile, come si può verificare osservando che la cella $(0, 4)$ è marcata: ne deriva che l'algoritmo marca anche la cella $(2, 3)$.

Proseguendo, l'algoritmo determina che:

- la coppia (q_2, q_4) è distinguibile se lo è la coppia (q_3, q_6): $(2, 4)$ è inserito nella lista della cella $(3, 6)$;
- la coppia (q_3, q_4) è distinguibile: la cella viene marcata;
- la coppia (q_2, q_5) è distinguibile se lo sono le coppie (q_2, q_4) e (q_3, q_6): $(2, 5)$ è inserito nelle liste delle celle $(2, 4)$ e $(3, 6)$;
- la coppia (q_3, q_5) è distinguibile: la cella viene marcata;
- la coppia (q_4, q_5) non è distinguibile;
- la coppia (q_2, q_6) è distinguibile: la cella non viene marcata;
- la coppia (q_3, q_6) è distinguibile se lo è la coppia (q_0, q_1): $(3, 6)$ è inserito nella lista della cella $(0, 1)$;
- la coppia (q_4, q_6) è distinguibile: la cella viene marcata;
- la coppia (q_5, q_6) è distinguibile: la cella viene marcata;

Al termine della sua esecuzione, l'algoritmo fornisce la tabella finale in Figura 3.27, la quale ci dice che le coppie di stati (q_0, q_1), (q_2, q_4), (q_2, q_5), (q_4, q_5), (q_3, q_6) non sono distinguibili: ne deriva che l'insieme degli stati dell'automa è partizionato nelle 3 classi di equivalenza $\{q_0, q_1\}$, $\{q_2, q_4, q_5\}$ e $\{q_3, q_6\}$. L'automa equivalente col minimo numero di stati risultante dal procedimento è quindi mostrato in Figura 3.28, in cui lo stato q_0 corrisponde alla classe d'equivalenza $\{q_0, q_1\}$, lo stato q_1 corrisponde alla classe $\{q_3, q_6\}$ e lo stato q_3, infine, corrisponde alla classe $\{q_2, q_4, q_5\}$.

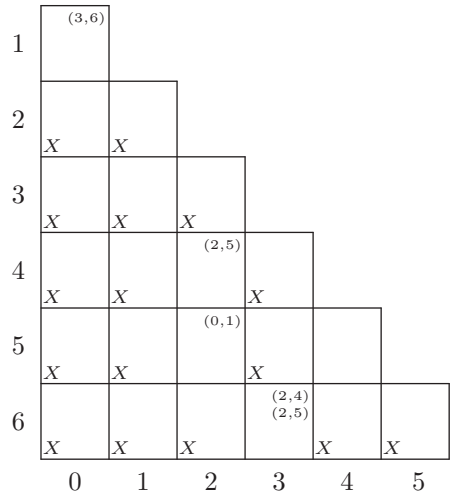


FIGURA 3.27 Tabella di tutte le coppie di stati distinguibili.

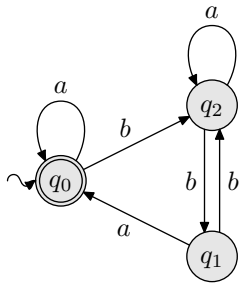


FIGURA 3.28 Automa minimo equivalente all'automa di Figura 3.25.

Capitolo 4

Linguaggi non contestuali

I linguaggi non contestuali (o context free, o di tipo 2) sono particolarmente interessanti per vari motivi. Innanzi tutto, da un punto di vista storico, è interessante notare che le grammatiche context free sono state introdotte dal linguista Noam Chomsky come un primo strumento per definire le strutture sintattiche del linguaggio naturale.

Un classico esempio di struttura di questo tipo è quella che caratterizza molte frasi del linguaggio naturale, come la sequenza di un sostantivo e di un predicato, in cui un predicato può essere un verbo o un verbo seguito da un complemento oggetto; questa struttura grammaticale si può rappresentare, mediante le notazioni viste, per mezzo della grammatica

$$\begin{aligned} F &\longrightarrow SP \\ P &\longrightarrow V \mid VC. \end{aligned}$$

Anche se le grammatiche di Chomsky si sono rivelate inadeguate allo studio del linguaggio naturale, l'uso di modelli del tipo delle grammatiche non contestuali rimane uno strumento concettuale utile per descrivere strutture linguistiche sufficientemente semplici.

D'altra parte, l'applicazione delle grammatiche di Chomsky ai linguaggi di programmazione ha dato risultati molto più validi, consentendo alle grammatiche non contestuali di assumere un ruolo fondamentale in quell'ambito. Uno dei principali strumenti per la descrizione formale dei linguaggi di programmazione, la Forma di Backus (BNF), è nient'altro che un formalismo per denotare una grammatica di tipo 2 (vedi Sezione 2.4). Anche i diagrammi sintattici, nonostante per alcuni aspetti appaiano simili ad automi a stati, sono in effetti riconducibili a strutture grammaticali di tipo non contestuale (vedi Sezione 2.4).

L'uso così estensivo di tale classe di linguaggi è essenzialmente dovuto al fatto che, come vedremo, i linguaggi non contestuali hanno alcune proprietà

di riconoscibilità particolarmente semplici. In particolare, i linguaggi generalmente usati nella programmazione dei calcolatori sono sostanzialmente un sottoinsieme proprio dei linguaggi non contestuali, e sono accettabili in tempo lineare con dispositivi piuttosto elementari, che utilizzano una memoria gestita come una pila. Questa condizione è utile perché, nel caso dei linguaggi di programmazione, l'analisi sintattica (o, con termine inglese, il *parsing*) di un programma e la sua traduzione in un linguaggio più elementare richiedono che, oltre a decidere se una stringa appartiene al linguaggio, venga anche, in caso positivo, determinato come la stringa stessa sia derivata nella corrispondente grammatica. Infatti, la conoscenza della struttura della derivazione di un programma in un linguaggio ad alto livello è necessaria per effettuare correttamente la sua traduzione in linguaggio macchina.

Tra le caratteristiche più interessanti dei linguaggi di tipo 2 c'è la possibilità di associare le derivazioni a strutture ad albero, note come *alberi di derivazione*, o *alberi sintattici*.

Esempio 4.1 Data la grammatica $\mathcal{G}$ avente le produzioni

$$\begin{aligned} S &\longrightarrow aSbA \mid ab \\ A &\longrightarrow cAd \mid cd \end{aligned}$$

la stringa $aaabbcdbcd \in L(\mathcal{G})$ può essere così derivata:

$$S \Longrightarrow aSbA \Longrightarrow aaSbAbA \Longrightarrow aaabbAbA \Longrightarrow aaabbcdbA \Longrightarrow aaabbcdbcd.$$

Questo processo di derivazione può essere utilmente rappresentato dall'albero dato in Figura 4.1 Si noti che usando tale rappresentazione non si determina univocamente

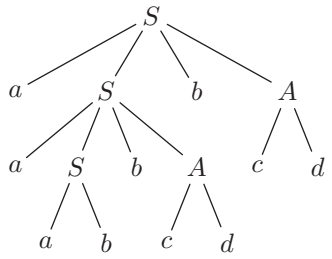


FIGURA 4.1 Esempio di albero di derivazione di una stringa.

---

un ordine con cui le produzioni debbono essere applicate. Al tempo stesso, l'albero di derivazione fornisce la descrizione più sintetica della struttura sintattica della stringa.

4.1 Forme ridotte e forme normali

Come si è detto precedentemente, le grammatiche di tipo 2 ammettono produzioni del tipo $A \longrightarrow \beta$, ove β è un'arbitraria stringa di terminali e non terminali. Al fine di studiare alcune proprietà dei linguaggi generati da queste grammatiche, è utile considerare grammatiche "ristrette", comprendenti soltanto produzioni con struttura particolare. È importante, quindi, dimostrare preliminarmente che i linguaggi non contestuali possono essere generati mediante tali tipi di grammatiche.

Prima di poter mostrare quali forme ristrette possono assumere le grammatiche per i linguaggi non contestuali, è necessario introdurre le grammatiche non contestuali in forma ridotta.

Definizione 4.1 *Una grammatica $\mathcal{G}$ è in forma ridotta se*

1. *non contiene ε-produzioni (se non, eventualmente, in corrispondenza all'assioma, ed in tal caso l'assioma non compare mai al lato destro di una produzione),*

2. *non contiene* produzioni unitarie*, cioè produzioni del tipo*

$$A \longrightarrow B, \text{ con } A, B \in V_N,$$

3. *non contiene* simboli inutili*, cioè simboli che non compaiono in nessuna derivazione di una stringa di soli terminali.*

La trasformazione di una grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ di tipo 2 in una grammatica equivalente in forma ridotta può essere effettuata nel modo seguente.

1. A partire da $\mathcal{G}$, si costruisce una grammatica $\mathcal{G}_1$ di tipo 2 senza ε-produzioni che genera $L(\mathcal{G}) - \{\varepsilon\}$.

2. A partire da $\mathcal{G}_1$ si costruisce una grammatica $\mathcal{G}_2$ di tipo 2 senza ε-produzioni e senza produzioni unitarie che genera $L(\mathcal{G}_1)$.

3. A partire da $\mathcal{G}_2$ si costruisce una grammatica $\mathcal{G}_3$ di tipo 2 senza ε-produzioni, senza produzioni unitarie e senza simboli inutili che genera $L(\mathcal{G}_2)$.

4. La grammatica $\mathcal{G}_4$, di tipo 2, equivalente a $\mathcal{G}$ coincide con $\mathcal{G}_3$ se $\varepsilon \notin L(\mathcal{G})$; altrimenti, $\mathcal{G}_4$ è ottenuta da $\mathcal{G}_3$ introducendo un nuovo assioma ed un opportuno insieme di produzioni su tale simbolo.

Abbiamo già visto nella Sezione 2.2 che, per effettuare il passo 1, si possono applicare gli Algoritmi 2.1 e 2.2 presentati nella dimostrazione del Teorema 2.2.

Per quanto riguarda il passo 2, consideriamo il seguente teorema.

Teorema 4.1 *Per ogni grammatica $\mathcal{G}$ di tipo 2 senza ε-produzioni, esiste sempre una grammatica $\mathcal{G}'$ di tipo 2 senza ε-produzioni, priva di produzioni unitarie ed equivalente a $\mathcal{G}$.*

Dimostrazione. Sia, per ogni $A \in V_N$, $U(A)$, il sottoinsieme di $V_N - \{A\}$ comprendente tutti i non terminali derivabili da A applicando una sequenza di produzioni unitarie, ovvero $U(A) = \{B \in V_N - \{A\} \mid A \stackrel{*}{\Longrightarrow} B\}$. Consideriamo l'Algoritmo 4.1 che, data la grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$, opera nella maniera seguente. L'algoritmo costruisce l'insieme P' delle produzioni di $\mathcal{G}'$ inserendo dapprima in P' tutte le produzioni non unitarie in P. Quindi, per ogni non terminale A e per ogni $B \in U(A)$, inserisce in P' una produzione $A \longrightarrow \beta$ se e solo se esiste in P una produzione non unitaria $B \longrightarrow \beta$.

input Grammatica non contestuale $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ priva di ε-produzioni;
output Grammatica non contestuale $\mathcal{G}' = \langle V_T, V_N, P', S\rangle$ priva di ε-produzioni e di produzioni unitarie equivalente a $\mathcal{G}$;
begin
  $P' := \{A \longrightarrow \alpha \in P \mid \alpha \notin V_N\}$;
  for each $A \in V_N$ **do**
    $P' := P' \cup \{A \longrightarrow \beta \mid (B \longrightarrow \beta \in P) \wedge (B \in U(A)) \wedge (\beta \notin V_N)\}$
end.

Algoritmo 4.1: Elimina Produzioni Unitarie

Per costruzione P' non contiene produzioni unitarie ed è inoltre facile provare che ogni stringa derivabile in $\mathcal{G}$ è anche derivabile in $\mathcal{G}'$ e viceversa. □

Esercizio 4.1 Costruire un algoritmo che, data una grammatica $\mathcal{G}$ di tipo 2 senza ε-produzioni e dato un non terminale A della grammatica, determini l'insieme $U(A)$.

Per quanto riguarda il passo 3, la possibilità di eliminare i simboli inutili è assicurata dal seguente teorema.

Teorema 4.2 *Per ogni grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ di tipo 2 senza ε-produzioni e senza produzioni unitarie, esiste sempre una grammatica $\mathcal{G}'$ di tipo 2 senza ε-produzioni, priva di produzioni unitarie e di simboli inutili ed equivalente a $\mathcal{G}$.*

Dimostrazione. Osserviamo che, affinché un simbolo $A \in V_N$ non sia inutile, è necessario che nella grammatica $\mathcal{G}$ si abbia che:

- A sia un simbolo *fecondo*, vale a dire che da esso siano generabili stringhe di terminali, cioè $\exists w \in V_T^+$ tale che $A \stackrel{*}{\Longrightarrow} w$;

- A sia generabile dall'assioma in produzioni che non contengano simboli non fecondi, cioè $S \stackrel{*}{\Longrightarrow} \alpha A \beta$ con $\alpha, \beta \in (V_T \cup V_N)^*$ e, per ogni $B \in V_N$ in α o β, valga la proprietà precedente.

Equivalentemente, un simbolo $A \in V_N$ non è inutile se esiste una derivazione $S \stackrel{*}{\Longrightarrow} \alpha A \beta \stackrel{*}{\Longrightarrow} w \in V_T^+$.

Per eliminare i simboli non fecondi, basta osservare che un non terminale A è fecondo se e solo se vale una delle due condizioni seguenti:

1. esiste $w \in V_T^+$ tale che $A \longrightarrow w \in P$;
2. esiste $\alpha \in (V_N \cup V_T)^*$ tale che $A \longrightarrow \alpha \in P$ e tutti i simboli non terminali in α sono fecondi.

L'Algoritmo 4.2 applica questa proprietà per eliminare dalla grammatica tutti i non terminali non fecondi, nonché tutte le produzioni che li generano, ottenendo così una grammatica ridotta $\widehat{\mathcal{G}} = \left\langle V_T, \widehat{V}_N, \widehat{P}, S \right\rangle$ priva di ε-produzioni, di produzioni unitarie ed equivalente a $\mathcal{G}$, contenente tutti non terminali da cui è possibile derivare delle stringhe di soli terminali. È facile convincersi che, in realtà, l'Algoritmo 4.2 elimina i simboli non fecondi anche se $\mathcal{G}$ contiene produzioni unitarie. È necessario a questo punto verificare che i simboli

input Grammatica non contestuale $\mathcal{G} = \langle V_T, V_N, P, S \rangle$, priva di ε-produzioni e di produzioni unitarie;
output Grammatica non contestuale $\widehat{\mathcal{G}} = \left\langle V_T, \widehat{V}_N, \widehat{P}, S \right\rangle$ priva di ε-produzioni, di produzioni unitarie e di simboli non fecondi, equivalente a $\mathcal{G}$;
begin
  $Q := V_T$;
  $R := V_N$;
  $\widehat{V}_N := \emptyset$;
  while $\exists A \in R$ per cui esiste $A \longrightarrow \alpha \in P$, con $\alpha \in Q^*$ **do**
    begin
      $\widehat{V}_N := \widehat{V}_N \cup \{A\}$;
      $Q := Q \cup \{A\}$;
      $R := R - \{A\}$
    end;
  $\widehat{P} := \{A \longrightarrow \alpha \mid A \longrightarrow \alpha \in P, A \in \widehat{V}_N, \alpha \in Q^*\}$
end.

Algoritmo 4.2: Algoritmo per l'eliminazione dei non terminali non fecondi

rimasti siano generabili a partire dall'assioma. Ciò può essere effettuato semplicemente, in modo iterativo, osservando che A è generabile a partire da S se vale una delle due condizioni seguenti:

1. esistono α, $\beta \in (\widehat{V}_N \cup V_T)^*$ tali che $S \longrightarrow \alpha A \beta \in \widehat{P}$;
2. esistono α, $\beta \in (\widehat{V}_N \cup V_T)^*$ e $B \in \widehat{V}_N$, generabile a partire da S, tali che $B \longrightarrow \alpha A \beta \in \widehat{P}$.

L'Algoritmo 4.3 applica questa proprietà per eliminare dalla grammatica tutti i non terminali non generabili dall'assioma. Dal fatto che l'applicazione dei

input Grammatica non contestuale $\widehat{\mathcal{G}} = \left\langle V_T, \widehat{V}_N, \widehat{P}, S \right\rangle$ priva di ε-produzioni, di produzioni unitarie e di simboli non fecondi;
output Grammatica non contestuale $\mathcal{G}' = \langle V_T, V'_N, P', S \rangle$ priva di ε-produzioni, di produzioni unitarie, di simboli non fecondi, di simboli non generabili da S, equivalente a $\widehat{\mathcal{G}}$;
begin
  $Q := \widehat{V}_N - \{S\}$;
  for each $A \in \widehat{V}_N - Q$ **do**
    for each $\alpha \in (V_T \cup \widehat{V}_N)^*$ tale che $A \longrightarrow \alpha \in \widehat{P}$ **do**
      for each $B \in Q$ che appare in α **do**
        $Q := Q - \{B\}$;
  $V'_N := \widehat{V}_N - Q$;
  $P' := \{A \longrightarrow \alpha \mid A \longrightarrow \alpha \in \widehat{P}, A \in V'_N, \alpha \in (V'_N \cup V_T)^*\}$
end.

Algoritmo 4.3: Algoritmo per l'eliminazione dei non terminali non generabili

due algoritmi elimina tutti e soli i simboli inutili deriva immediatamente che nell'ambito di $\mathcal{G}'$ vengono generate tutte le stringhe di terminali generabili in $\mathcal{G}$. □

Infine, per quanto riguarda il passo 4, ricordiamo (vedi Sezione 2.2) che, in generale, una grammatica $\mathcal{G} = \langle V_T, V_N, P, S \rangle$ può essere estesa in una grammatica $\mathcal{G}' = \langle V_T, V'_N, P', S' \rangle$ che generi anche la stringa vuota ε nel modo seguente:

1. $V'_N = V_N \cup \{T\}$, dove $T \notin V_N$;
2. $P' = P \cup \{T \longrightarrow \varepsilon\} \cup \{T \longrightarrow \alpha \mid S \longrightarrow \alpha \in P\}$;
3. $S' = T$.

Esercizio 4.2 Dimostrare che la grammatica $\mathcal{G}'$ genera il linguaggio $L(\mathcal{G}) \cup \{\varepsilon\}$.

A conclusione delle considerazioni precedenti, e dei Teoremi 2.2, 4.1 e 4.2, possiamo enunciare il seguente risultato.

Teorema 4.3 *Per ogni grammatica $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ di tipo 2 esiste una grammatica $\mathcal{G}'$ di tipo 2 in forma ridotta, equivalente a $\mathcal{G}$.*

Si osservi che, al fine di eliminare i simboli inutili (non fecondi e non generabili da S) è necessario applicare i due algoritmi nell'ordine dato: eliminare prima i simboli non generabili e poi quelli non fecondi può far sì che non tutti i simboli inutili vengano rimossi dalla grammatica. Infatti, si consideri la seguente grammatica

$$\begin{array}{lcl} S & \longrightarrow & AB \mid a \\ A & \longrightarrow & a. \end{array}$$

Se procedessimo prima all'eliminazione dei simboli non derivabili dall'assioma e poi all'eliminazione di quelli non fecondi, otterremmo le seguenti grammatiche:

$$\begin{array}{lcl} S & \longrightarrow & AB \mid a \\ A & \longrightarrow & a \end{array} \quad \text{e successivamente} \quad \begin{array}{lcl} S & \longrightarrow & a \\ A & \longrightarrow & a. \end{array}$$

che non è in forma ridotta. Se invece si procede come indicato nel teorema si ottengono le due grammatiche

$$\begin{array}{lcl} S & \longrightarrow & a \\ A & \longrightarrow & a \end{array} \quad \text{e successivamente} \quad S \longrightarrow a.$$

Esempio 4.2 Sia data la grammatica

$$\begin{array}{lcll} 1 & S & \longrightarrow & aUVb \mid TZ \\ 2 & Z & \longrightarrow & aZ \\ 3 & U & \longrightarrow & bU \mid b \\ 4 & V & \longrightarrow & W \\ 5 & V & \longrightarrow & aY \\ 6 & Y & \longrightarrow & bY \mid b \\ 7 & W & \longrightarrow & cWd \mid cd \\ 8 & T & \longrightarrow & tT \mid tz. \end{array}$$

L'eliminazione delle produzioni unitarie porta ad escludere la produzione 4 e ad aggiungere una terza produzione alla 1. L'eliminazione di simboli non fecondi porta ad escludere la produzione 2 e la seconda produzione della 1. L'eliminazione dei simboli non raggiungibili porta infine ad escludere la produzione 8.

Si ottiene quindi la grammatica

$$\begin{array}{lcl} S & \longrightarrow & aUVb \mid aUWb \\ U & \longrightarrow & bU \mid b \\ V & \longrightarrow & aY \\ Y & \longrightarrow & bY \mid b \\ W & \longrightarrow & cWd \mid cd. \end{array}$$

Esercizio 4.3 Data la seguente grammatica:

$$\begin{aligned} S &\longrightarrow H \mid Z \\ H &\longrightarrow A \mid \varepsilon \\ Z &\longrightarrow bZb \\ A &\longrightarrow bbABa \mid a \\ B &\longrightarrow cB \mid BZY \mid \varepsilon \\ Y &\longrightarrow Yb \mid b. \end{aligned}$$

Trasformarla in una grammatica equivalente in forma ridotta.

Mostriamo ora come per qualunque grammatica non contestuale ne esista un'altra equivalente con una struttura delle produzioni particolarmente semplice. Considereremo due possibili strutture delle produzioni, che danno luogo a due *forme normali* per le grammatiche context free e che sono entrambe semplici generalizzazioni delle produzioni di una grammatica regolare: la prima consiste in produzioni che generano o un solo terminale o una coppia di non terminali, la seconda in produzioni che generano un terminale seguito da una sequenza (eventualmente vuota) di non terminali.

Definizione 4.2 *Una grammatica di tipo 2 si dice in* Forma Normale di Chomsky *(CNF) se tutte le sue produzioni sono del tipo* $A \longrightarrow BC$ *o del tipo* $A \longrightarrow a$*, con* $A, B, C \in V_N$ *ed* $a \in V_T$*.*

Teorema 4.4 *Data una grammatica* $\mathcal{G}$ *non contestuale tale che* $\varepsilon \notin L(G)$*, esiste una grammatica equivalente in CNF.*

Dimostrazione. Costruiamo innanzi tutto una grammatica $\mathcal{G}'$ in forma ridotta equivalente a $\mathcal{G}$, come indicato nel Teorema 4.3: si ricordi in particolare che in $\mathcal{G}'$ non abbiamo produzioni unitarie.

Mostriamo ora come derivare da $\mathcal{G}'$ una grammatica equivalente $\mathcal{G}''$ in CNF. Sia $A \longrightarrow \zeta_{i_1} \dots \zeta_{i_n}$ una produzione di $\mathcal{G}'$ non in CNF. Si possono verificare due casi:

1. $n \geq 3$ e $\zeta_{i_j} \in V_N$, $j = 1, \dots, n$. In tal caso, introduciamo $n-2$ nuovi simboli non terminali Z_1, $\dots$, Z_{n-2} e sostituiamo la produzione $A \longrightarrow \zeta_{i_1} \dots \zeta_{i_n}$ con le produzioni

$$\begin{aligned} A &\longrightarrow \zeta_{i_1} Z_1 \\ Z_1 &\longrightarrow \zeta_{i_2} Z_2 \\ &\dots \\ Z_{n-2} &\longrightarrow \zeta_{i_{n-1}} \zeta_{i_n}. \end{aligned}$$

2. $n \geq 2$ e $\zeta_{i_j} \in V_T$ per qualche $j \in \{1, \dots, n\}$. In tal caso per ciascun $\zeta_{i_j} \in V_T$ introduciamo un nuovo non terminale $\overline{Z}_{i_j}$, sostituiamo $\overline{Z}_{i_j}$ a ζ_{i_j}

nella produzione considerata e aggiungiamo la produzione $\overline{Z}_{i_j} \longrightarrow \zeta_{i_j}$. Così facendo o abbiamo messo in CNF la produzione considerata (se $n = 2$) o ci siamo ricondotti al caso precedente (se $n \geq 3$).

□

Esercizio 4.4 Mostrare che le grammatiche $\mathcal{G}'$ e $\mathcal{G}''$ nella dimostrazione del Teorema 4.4 sono equivalenti.

Come si può vedere, la minore complessità nella struttura delle produzioni, caratteristica della Forma Normale di Chomsky, viene pagata in generale, in termini quantitativi, dalla necessità di utilizzare una maggiore quantità di simboli non terminali e di produzioni per descrivere lo stesso linguaggio.

Esempio 4.3 Si consideri la grammatica di tipo 2 che genera il linguaggio $\{a^n b^n \mid n \geq 1\}$ con le produzioni

$$\begin{aligned} S &\longrightarrow aSb \\ S &\longrightarrow ab \end{aligned}$$

Applicando il metodo visto nella dimostrazione del Teorema 4.4 e osservando che la grammatica è già in forma ridotta, otteniamo una nuova grammatica avente i non terminali $S, Z_1, \overline{Z}_1, \overline{Z}_2, \overline{Z}_3$ e $\overline{Z}_4$, e l'insieme di produzioni

$$\begin{aligned} S &\longrightarrow \overline{Z}_1 Z_1 \\ Z_1 &\longrightarrow S\overline{Z}_2 \\ S &\longrightarrow \overline{Z}_3 \overline{Z}_4 \\ \overline{Z}_1 &\longrightarrow a \\ \overline{Z}_2 &\longrightarrow b \\ \overline{Z}_3 &\longrightarrow a \\ \overline{Z}_4 &\longrightarrow b \end{aligned}$$

Esercizio 4.5 Data una grammatica $\mathcal{G} = \langle V_T, V_N, P, S \rangle$ con al più k simboli nella parte destra di ogni produzione, determinare la massima cardinalità possibile degli insiemi V'_N e P' della corrispondente grammatica in CNF.

Il secondo tipo di forma normale che consideriamo risulta particolarmente interessante in quanto, oltre a permettere di caratterizzare i linguaggi context free in termini di accettabilità per mezzo di un opportuno modello di calcolo (gli *Automi a pila non deterministici*, che saranno introdotti nella Sezione 4.5) fornisce una struttura delle produzioni che è alla base di varie tecniche di analisi sintattica dei linguaggi non contestuali.

Definizione 4.3 *Una grammatica di tipo 2 si dice in* Forma Normale di Greibach *(GNF) se tutte le sue produzioni sono del tipo* $A \longrightarrow a\beta$, *con* $A \in V_N$, $a \in V_T, \beta \in V_N^*$.

Si noti come anche una grammatica in GNF abbia produzioni con una struttura che è una generalizzazione della struttura delle produzioni di una grammatica regolare (nel qual caso $|\beta| \leq 1$ per tutte le produzioni).

Al fine di mostrare che ogni grammatica di tipo 2 può essere trasformata in una grammatica equivalente in GNF, osserviamo innanzi tutto le seguenti proprietà.

Definizione 4.4 *Sia data una grammatica non contestuale* $\mathcal{G} = \langle V_T, V_N, P, S \rangle$ *e sia* $A \in V_N$. *Definiamo come* A-produzioni *tutte le produzioni in* P *del tipo* $A \longrightarrow \alpha$, *con* $\alpha \in (V_N \cup V_T)^*$.

Lemma 4.5 (Sostituzione) *Sia* $\mathcal{G}$ *una grammatica di tipo 2 le cui produzioni includono*

$$\begin{aligned} A &\longrightarrow \alpha_1 B \alpha_2 \\ B &\longrightarrow \beta_1 \mid \ldots \mid \beta_n, \end{aligned}$$

($\alpha_1, \alpha_2 \in V^*$*) e in cui non compaiono altre* B*-produzioni oltre a quelle indicate. La grammatica* $\mathcal{G}'$ *in cui la produzione* $A \longrightarrow \alpha_1 B \alpha_2$ *è stata sostituita dalla produzione*

$$A \longrightarrow \alpha_1\beta_1\alpha_2 \mid \ldots \mid \alpha_1\beta_n\alpha_2$$

è equivalente alla grammatica $\mathcal{G}$.

Dimostrazione. Ogni derivazione in $\mathcal{G}'$ comprendente il passo

$$\sigma_1 A \sigma_2 \underset{\mathcal{G}'}{\Longrightarrow} \sigma_1\alpha_1\beta_i\alpha_2\sigma_2$$

può essere sostituita da una derivazione in $\mathcal{G}$ contenente i due passi

$$\sigma_1 A \sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \sigma_1\alpha_1 B \alpha_2\sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \sigma_1\alpha_1\beta_i\alpha_2\sigma_2.$$

Al contrario, si osservi che ogni derivazione in $\mathcal{G}$ che applica la produzione $A \longrightarrow \alpha_1 B \alpha_2$ deve anche applicare, successivamente, una delle produzioni $B \longrightarrow \beta_1 \mid \ldots \mid \beta_n$, e quindi dovrà essere del tipo:

$$\begin{aligned} S \underset{\mathcal{G}}{\Longrightarrow} & \cdots \underset{\mathcal{G}}{\Longrightarrow} \sigma_1 A \sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \cdots \underset{\mathcal{G}}{\Longrightarrow} \sigma_1\alpha_1 B \alpha_2\sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \cdots \\ & \cdots \underset{\mathcal{G}}{\Longrightarrow} \sigma_1\alpha_1\beta_i\alpha_2\sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \cdots \underset{\mathcal{G}}{\Longrightarrow} w \in V_T^*. \end{aligned}$$

Chiaramente, riordinando la sequenza di applicazioni delle produzioni nella derivazione, possiamo osservare che la stessa stringa w può essere derivata nel modo seguente:

$$S \underset{\mathcal{G}}{\Longrightarrow} \cdots \underset{\mathcal{G}}{\Longrightarrow} \sigma_1 A \sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \sigma_1\alpha_1 B \alpha_2\sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \sigma_1\alpha_1\beta_i\alpha_2\sigma_2 \underset{\mathcal{G}}{\Longrightarrow} \cdots \underset{\mathcal{G}}{\Longrightarrow} w \in V_T^*.$$

A tale derivazione corrisponderà, in $\mathcal{G}'$, la derivazione

$$S \underset{\mathcal{G}'}{\Longrightarrow} \cdots \underset{\mathcal{G}'}{\Longrightarrow} \sigma_1 A \sigma_2 \underset{\mathcal{G}'}{\Longrightarrow} \sigma_1 \alpha_1 \beta_i \alpha_2 \sigma_2 \cdots \underset{\mathcal{G}'}{\Longrightarrow} w \in V_T^*.$$

□

Definizione 4.5 *Data una grammatica $\mathcal{G} = \langle V_T, V_N, P, S \rangle$ di tipo 2 e dato un non terminale $A \in V_N$, diciamo che in $\mathcal{G}$ esiste una* ricursione sinistra *rispetto al non terminale A se esiste una produzione $A \longrightarrow A\alpha$, con $\alpha \in (V_N \cup V_T)^*$.*

Lemma 4.6 (Eliminazione della ricursione sinistra) *Sia $\mathcal{G}$ una grammatica con ricursione sinistra sul non terminale A e sia*

$$A \longrightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n,$$

l'insieme dell A-produzioni in $\mathcal{G}$, dove nessuna delle stringhe β_i inizia per A. La grammatica $\mathcal{G}'$ in cui le A-produzioni in $\mathcal{G}$ sono state sostituite dalle produzioni:

$$\begin{array}{rcl} A & \longrightarrow & \beta_1 A' \mid \ldots \mid \beta_n A' \mid \beta_1 \ldots \mid \beta_n \\ A' & \longrightarrow & \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \alpha_1 \ldots \mid \alpha_m \end{array}$$

è equivalente a $\mathcal{G}$ e non presenta ricursione sinistra rispetto al non terminale A.

Dimostrazione. Basta osservare che tutte le stringhe derivabili in $\mathcal{G}$ a partire da A sono del tipo del tipo $(\beta_1 + \ldots + \beta_n)(\alpha_1 + \ldots + \alpha_m)^*$: è immediato verificare che lo stesso è vero in $\mathcal{G}'$. □

A questo punto, possiamo mostrare come per ogni grammatica non contestuale ne esista una equivalente in GNF.

Teorema 4.7 *Ogni linguaggio non contestuale L tale che $\varepsilon \notin L$ può essere generato da una grammatica di tipo 2 in GNF.*

Dimostrazione. La dimostrazione si basa sulla derivazione, a partire da una grammatica $\mathcal{G}$ in CNF che genera L (ottenuta in base al Teorema 4.4), di una grammatica equivalente $\mathcal{G}'$ in GNF. Tale derivazione è effettuata mediante una ripetuta esecuzione dei due passi fondamentali, sostituzione ed eliminazione della ricursione sinistra, di cui si è vista la struttura e dimostrata la correttezza nei due lemmi precedenti. La derivazione di $\mathcal{G}'$ avviene applicando l'Algoritmo 4.4. Non è difficile verificare che la grammatica $\mathcal{G}'$ ottenuta è in GNF ed è equivalente a $\mathcal{G}$. □

input Grammatica non contestuale $\mathcal{G}$ in CNF;
output Grammatica non contestuale $\mathcal{G}'$ in GNF;
begin
  Sia $A_1, \ldots, A_n$ un ordinamento arbitrario tra i non terminali di $\mathcal{G}$;
  for $k := 2$ **to** n **do**
  begin
    for $j := 1$ **to** $k-1$ **do**
      Applica il Lemma 4.5 ad ogni produzione del tipo $A_k \longrightarrow A_j\alpha$;
    Applica il Lemma 4.6 ad ogni produzione del tipo $A_k \longrightarrow A_k\alpha$
  end;
// *Siano* $B_1 \ldots, B_l$ *i non terminali aggiunti*;
// *A questo punto le produzioni sono tutte di uno tra i tipi:*
// (a) $A_k \longrightarrow A_j\gamma$, *con* $j > k, \gamma \in (V_N \cup \{B_1, \ldots, B_l\})^*$;
// (b) $A_k \longrightarrow a\gamma$, *con* $a \in V_T, \gamma \in (V_N \cup \{B_1, \ldots, B_l\})^*$;
// (c) $B_k \longrightarrow \gamma$, *con* $\gamma \in V_N \cdot (V_N \cup \{B_1, \ldots, B_l\})^*$.
// *Inoltre, se* $k = n$ *le* A_k*-produzioni sono tutte del tipo* (b),
// *e se* $k = m - h$ *sono del tipo* (b) *o del tipo* (a), *con* $k < j \leq n$.
  for $h := n-1$ **down to** 1 **do**
    for $j := n$ **down to** h **do**
      Applica il Lemma 4.5 ad ogni produzione del tipo $A_h \longrightarrow A_j\gamma$;
// *A questo punto le produzioni sono tutte del tipo* (b) *o* (c).
  for $i := 1$ **to** l **do**
    for $j := 1$ **to** m **do**
      Applica il Lemma 4.5 ad ogni produzione del tipo $B_i \longrightarrow A_j\gamma$;
end.

Algoritmo 4.4: Algoritmo per la riduzione in Forma Normale di Greibach

Esercizio 4.6 Mostrare che le grammatiche $\mathcal{G}$ e $\mathcal{G}'$ nella dimostrazione del Teorema 4.7 sono equivalenti.
[Suggerimento: Mostrare che ogni applicazione dei Lemmi 4.5 e 4.6 non altera il linguaggio generato.]

Esempio 4.4 Data una grammatica avente le produzioni

$$\begin{aligned} S &\longrightarrow AB \mid b \\ A &\longrightarrow b \mid BS \\ B &\longrightarrow a \mid BA \mid AS, \end{aligned}$$

il procedimento illustrato nella dimostrazione del Teorema 4.7 costruisce una grammatica equivalente in GNF nel modo seguente (si noti che la grammatica iniziale è già in CNF).

1. Consideriamo, in modo arbitrario, il seguente ordinamento fra i non terminali: S, A, B.

2. Sostituiamo alla produzione $B \longrightarrow AS$ la coppia di produzioni $B \longrightarrow bS \mid BSS$, ottenendo le B-produzioni:

   $$\begin{array}{lll} B & \longrightarrow & a \mid bS \\ B & \longrightarrow & BA \mid BSS. \end{array}$$

   A questo punto non vi sono più produzioni del tipo $A_k \longrightarrow A_j\gamma$ con $j < k$, e quindi non sono applicabili altre sostituzioni.

3. Interveniamo quindi per eliminare la ricursione sinistra nelle B-produzioni. In base al Lemma 4.6 le produzioni suddette sono equivalenti alle

   $$\begin{array}{lll} B & \longrightarrow & a \mid bS \mid aB' \mid bSB' \\ B' & \longrightarrow & A \mid SS \mid AB' \mid SSB'. \end{array}$$

4. Effettuiamo ora le sostituzioni nelle produzioni del tipo $A_k \longrightarrow A_j\gamma$ con $k < j$. Alla produzione $A \longrightarrow BS$ sostituiamo le produzioni

   $$A \longrightarrow aS \mid bSS \mid aB'S \mid bSB'S$$

   ed alla produzione $S \longrightarrow AB$ sostituiamo le produzioni

   $$S \longrightarrow aSB \mid bSSB \mid aB'SB \mid bSB'SB \mid bB.$$

5. Alle produzioni $B' \longrightarrow A \mid SS \mid AB' \mid SSB'$ sostituiamo ora le produzioni

   $$\begin{array}{lll} B' & \longrightarrow & aS \mid bSS \mid aB'S \mid bSB'S \mid b \mid \\ & & aSBS \mid bSSBS \mid aB'SBS \mid bSB'SBS \mid bSB \mid bS \mid \\ & & aSB' \mid bSSB' \mid aB'SB' \mid bSB'SB' \mid bB' \mid \\ & & aSBSB' \mid bSSBSB' \mid aB'SBSB' \mid bSB'SBSB' \mid bBSB' \mid bSB'. \end{array}$$

Si è giunti, in conclusione, alla seguente grammatica in GNF:

$$\begin{array}{lll} S & \longrightarrow & aSB \mid bSSB \mid aB'SB \mid bSB'SB \mid bB \mid b \\ B & \longrightarrow & a \mid bS \mid aB \mid bSB' \\ B' & \longrightarrow & aS \mid bSS \mid aB'S \mid bSB'S \mid b \\ & & asBS \mid bSSBS \mid aB'SBS \mid bSB'SBS \mid bSB \mid bS \mid \\ & & aSB' \mid bSSB' \mid aB'SB' \mid bSB'SB' \mid bB' \mid \\ & & aSBSB' \mid bSSBSB' \mid aB'SBSB' \mid bSB'SBSB' \mid bBSB' \mid bSB'. \end{array}$$

Si noti che sono state omesse le A produzioni, in quanto tale non terminale non è più derivabile a partire da S.

Esercizio 4.7 Sia data la seguente grammatica:

$$\begin{array}{lll} S & \longrightarrow & AbA \mid b \\ A & \longrightarrow & SaS \mid a. \end{array}$$

Derivare una grammatica in GNF equivalente ad essa.

Esercizio 4.8 Utilizzando il procedimento sopra illustrato, mostrare che una grammatica lineare sinistra genera un linguaggio regolare.

4.2 Il "pumping lemma" per i linguaggi context free

Così come per i linguaggi regolari, è possibile mostrare come anche i linguaggi non contestuali presentino delle "regolarità", nel senso che l'appartenza di una stringa sufficientemente lunga ad un linguaggio CF implica l'appartenenza allo stesso linguaggio di un insieme (infinito) di altre stringhe strutturalmente simili ad essa.

Teorema 4.8 (Pumping lemma per i linguaggi CF) *Sia $L \subseteq V_T^*$ un linguaggio non contestuale. Esiste allora una costante n tale che se $z \in L$ e $\mid z \mid \geq n$ allora esistono 5 stringhe $u, v, w, x, y \in V_T^*$ tali che*

$i)$ $uvwxy = z$
$ii)$ $\mid vx \mid \geq 1$
$iii)$ $\mid vwx \mid \leq n$
$iv)$ $\forall i \geq 0\ uv^iwx^iy \in L$.

Dimostrazione. Consideriamo una grammatica $\mathcal{G} = \langle V_T, V_N, P, S \rangle$ in CNF che genera $L = L(\mathcal{G})$ e sia $k = \mid V_N \mid$ il numero di simboli non terminali in $\mathcal{G}$. Si osservi che qualunque albero sintattico $A(x)$ relativo ad una stringa $x \in V_T^*$ derivata in $\mathcal{G}$ sarà tale da avere tutti i nodi interni (corrispondenti a simboli non terminali) di grado 2, eccetto quelli aventi foglie dell'albero come figli, che hanno grado 1.

È allora facile verificare che, se $h(x)$ è l'altezza di $A(x)$ (numero massimo di archi in un cammino dalla radice ad una foglia), abbiamo $\mid x \mid \leq 2^{h(x)}$: in conseguenza di ciò, se $\mid x \mid > 2^{\mid V_N \mid}$ allora $h(x) > \mid V_N \mid$ e, quindi, deve esistere un cammino dalla radice ad una foglia che attraversa almeno $\mid V_N \mid +1$ nodi interni. Per il pigeonhole principle, (almeno) due di questi nodi saranno associati ad uno stesso non terminale, ad esempio A. Indichiamo con r il nodo più vicino alla radice associato al simbolo A, e con s il nodo associato ad A più vicino alla foglia (vedi Figura 4.2).

Come si può osservare, dalle due occorrenze di A in r ed s derivano stringhe diverse (indicate come vwx e w, rispettivamente), di cui, in particolare, l'una occorre come sottostringa nell'altra. Si osservi anche che, avendo almeno un nodo sul cammino da r ad s grado 2, ne segue che $\mid vx \mid \geq 1$. Considerando che gli alberi in Figura 4.3 possono essere sostituiti l'uno all'altro all'interno di un qualunque albero sintattico, abbiamo che anche la stringa uwy è generata dalla grammatica (sostituendo nell'albero di Figura 4.2 all'albero di Figura 4.3(b)

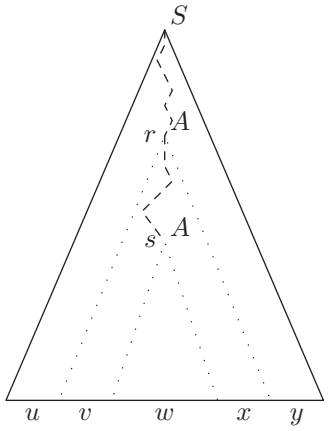


FIGURA 4.2 Non terminale ripetuto due volte in un ramo.

quello di Figura 4.3(a)), così come anche la stringa $uvvwxxy$ (sostituendo nell'albero di Figura 4.2 all'albero di Figura 4.3(a) quello di Figura 4.3(b)): si noti che, iterando questo tipo di sostituzione, si può mostrare che qualunque stringa uv^iwx^iy appartiene al linguaggio. □

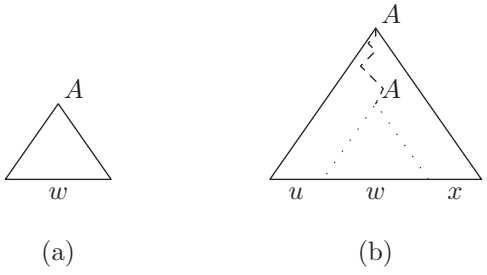


FIGURA 4.3 (a) sottoalbero che genera w.
(b) sottoalbero che genera vwx.

Come nel caso dei linguaggi regolari, il pumping lemma è una condizione necessaria ma non sufficiente affinché un linguaggio appartenga alla classe, e quindi, in questo caso, sia non contestuale.[1] Ciò significa che possono esistere

[1]Esiste in effetti un risultato, detto Lemma di Ogden, che in questa sede non viene riportato e che costituisce una condizione necessaria e sufficiente affinché un linguaggio sia

linguaggi non di tipo 2 che tuttavia soddisfano le condizioni poste dal lemma.

In effetti, il pumping lemma viene normalmente applicato in senso "negativo", per dimostrare che un determinato linguaggio non è di tipo 2.

Esempio 4.5 Il linguaggio $\{a^n b^n c^n \mid n \geq 1\}$ non è di tipo 2.

Per mostrare che il linguaggio non è di tipo 2, verifichiamo che esso non soddisfa il pumping lemma. Mostriamo quindi che per ogni valore di n esiste una strina z, con $\mid z \mid = n$, che non è decomponibile nelle 5 stringhe u, v, w, x, y aventi le proprietà richieste dal lemma.

Sia $z = a^n b^n c^n = uvwxy$. Allora se v ed x contengono almeno due diversi simboli, la stringa uv^2wx^2y conterrà simboli mescolati. Se viceversa v ed x contengono un solo simbolo, uv^2wx^2y non ha più la proprietà che le stringhe del linguaggio contengono un ugual numero di a, di b e di c.

Esercizio 4.9 Usando il Pumping Lemma dimostrare che i seguenti linguaggi non sono context free:

$\{a^n b^m c^p \mid 1 \leq n \leq m \leq p\}$
$\{a^{2^n} \mid n \geq 1\}$
$\{a^i b^j c^i d^j \mid i, j \geq 1\}$
$\{ww \mid w \in \{0,1\}^+\}$

4.3 Chiusura di operazioni su linguaggi context free

Il fatto che il linguaggio $\{a^n b^n c^n \mid n \geq 1\}$ non sia context free, visto nell'Esempio 4.5, ha alcune conseguenze molto interessanti che riguardano le proprietà di chiusura dei linguaggi non contestuali.

Corollario 4.9 *I linguaggi non contestuali non sono chiusi rispetto all'intersezione.*

Dimostrazione. Sia $\{a^n b^n c^m \mid n, m \geq 1\}$ che $\{a^m b^n c^n \mid n, m \geq 1\}$ sono non contestuali, ma la loro intersezione, che coincide appunto con il linguaggio $\{a^n b^n c^n \mid n \geq 1\}$ non lo è. □

Quest'ultimo risultato ci permette di osservare subito un'importante differenza tra i linguaggi regolari ed i non contestuali. Non essendo questi ultimi chiusi rispetto all'intersezione, la tecnica con cui, nella dimostrazione del Teorema 3.17, abbiamo mostrato che si può decidere l'equivalenza di due linguaggi

non contestuale.

regolari non è estendibile ai linguaggi non contestuali. In effetti, come vedremo successivamente, l'equivalenza di linguaggi non contestuali, così come altre proprietà dei linguaggi stessi, come ad esempio l'ambiguità, non è decidibile.

Mostriamo qui di seguito alcune operazioni rispetto alle quali la classe dei linguaggi non contestuali possiede invece la proprietà di chiusura.

Teorema 4.10 *I linguaggi non contestuali sono chiusi rispetto all'unione.*

Dimostrazione. Dati due linguaggi context free $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$, siano $\mathcal{G}_1 = \langle \Sigma_1, V_{N1}, P_1, S_1 \rangle$ e $\mathcal{G}_2 = \langle \Sigma_2, V_{N2}, P_2, S_2 \rangle$ due grammatiche di tipo 2 tali che $L_1 = L(\mathcal{G}_1)$ e $L_2 = L(\mathcal{G}_2)$.

Mostriamo ora che il linguaggio $L = L_1 \cup L_2$ potrà allora essere generato dalla grammatica di tipo 2 $\mathcal{G} = \langle \Sigma_1 \cup \Sigma_2, V_{N1} \cup V_{N2} \cup \{S\}, P, S \rangle$, dove $P = P_1 \cup P_2 \cup \{S \longrightarrow S_1 \mid S_2\}$.

Chiaramente, ogni stringa in $w \in L_1$ derivabile in $\mathcal{G}_1$ mediante una derivazione $S_1 \stackrel{*}{\Longrightarrow} w$ sarà derivabile in $\mathcal{G}$ mediante la derivazione $S \Longrightarrow S_1 \stackrel{*}{\Longrightarrow} w$ ottenuta dalla precedente anteponendo l'applicazione della produzione $S \longrightarrow S_1$, e lo stesso avverrà per qualunque stringa $v \in L_2$: da ciò segue che $L_1 \cup L_2 \subseteq L(\mathcal{G})$.

Al contrario, qualunque stringa $x \in \mathcal{G}$ sarà derivata mediante una derivazione del tipo $S \Longrightarrow S_1 \stackrel{*}{\Longrightarrow} x$ o del tipo $S \Longrightarrow S_2 \stackrel{*}{\Longrightarrow} x$, in quanto le sole produzioni con l'assioma S nella parte sinistra sono $S \longrightarrow S_1$ e $S \longrightarrow S_2$: nel primo caso la stringa è derivabile in $\mathcal{G}_1$ e nel secondo è derivabile in $\mathcal{G}_2$, e quindi si ha che $L(\mathcal{G}) \subseteq L_1 \cup L_2$, dal che consegue il teorema. □

Teorema 4.11 *I linguaggi non contestuali sono chiusi rispetto alla concatenazione.*

Dimostrazione. Dati due linguaggi context free $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$, siano $\mathcal{G}_1 = \langle \Sigma_1, V_{N1}, P_1, S_1 \rangle$ e $\mathcal{G}_2 = \langle \Sigma_2, V_{N2}, P_2, S_2 \rangle$ due grammatiche di tipo 2 tali che $L_1 = L(\mathcal{G}_1)$ e $L_2 = L(\mathcal{G}_2)$.

Mostriamo che il linguaggio $L = L_1 \circ L_2$ è generato dalla grammatica di tipo 2 definita come $\mathcal{G} = \langle \Sigma_1 \cup \Sigma_2, V_{N1} \cup V_{N2} \cup \{S\}, P, S \rangle$, dove $P = P_1 \cup P_2 \cup \{S \longrightarrow S_1 S_2\}$.

Si osservi che ogni stringa in $w \in L_1 \circ L_2$ ha la struttura $w = w_1 w_2$, dove $w_1 \in L_1$ è derivabile in $\mathcal{G}_1$ mediante una derivazione $S_1 \stackrel{*}{\Longrightarrow} w_1$ e $w_2 \in L_2$ è derivabile in $\mathcal{G}_2$ mediante una derivazione $S_2 \stackrel{*}{\Longrightarrow} w_2$. La stringa w sarà allora derivabile in $\mathcal{G}$ mediante la derivazione $S \Longrightarrow S_1 S_2 \stackrel{*}{\Longrightarrow} w_1 S_2 \stackrel{*}{\Longrightarrow} w_1 w_2$ ottenuta anteponendo l'applicazione della produzione $S \longrightarrow S_1$, alla derivazione di w_1 da S_1 e, quindi, alla derivazione di w_2 da S_2: da ciò segue che $L_1 \circ L_2 \subseteq L(\mathcal{G})$.

Al contrario, qualunque stringa $x \in \mathcal{G}$ sarà derivata mediante una derivazione del tipo $S \Longrightarrow S_1S_2 \stackrel{*}{\Longrightarrow} x$, in quanto questa è la sola produzione con l'assioma S nella parte sinistra: da ciò consegue che x sarà necessariamente composta dalla concatenazione di una prima stringa derivata da S_1, e quindi in L_1, con una seconda stringa derivata da S_2, e quindi in L_2. In conseguenza di ciò, $x \in L_1 \circ L_2$, $L(\mathcal{G}) \subseteq L_1 \circ L_2$, e il teorema deriva. □

Teorema 4.12 *I linguaggi non contestuali sono chiusi rispetto all'iterazione.*

Dimostrazione. La dimostrazione è lasciata come esercizio. □

Esercizio 4.10 Dimostrare il Teorema 4.12.

Esercizio 4.11 Dimostrare che la classe dei linguaggi non contestuali non è chiusa rispetto alla complementazione.

4.4 Predicati decidibili sui linguaggi context free

Una prima applicazione delle considerazioni effettuate nella dimostrazione del pumping lemma permette di evidenziare come sia possibile determinare, mediante opportuni algoritmi, se una grammatica non contestuale genera un linguaggio infinito, finito non vuoto, o vuoto.

Teorema 4.13 *Data una grammatica $\mathcal{G}$ di tipo 2 è decidibile stabilire se $L(G) = \emptyset$.*

Dimostrazione. Si assuma, senza perdita di generalità (vedi Teorema 4.4), che $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ sia una grammatica di tipo 2 in Forma Normale di Chomsky e si ponga $n = |\, V_N \,|$.

Ricordiamo che, per il pumping lemma per i linguaggi non contestuali (Teorema 4.8), se esiste una stringa $z = uvwxy \in L(\mathcal{G})$ con $|\, z \,| > 2^n$, allora esiste una stringa $z' = uwy \in L(\mathcal{G})$ con $|\, z' \,| \leq 2^n$.

In conseguenza di ciò, se $L(\mathcal{G}) \neq \emptyset$ allora esiste qualche stringa di lunghezza al più 2^n che fa parte del linguaggio. Dato che in una grammatica in CNF ogni applicazione di una produzione o incrementa di uno la lunghezza della forma di frase (se la produzione è del tipo $A \longrightarrow BC$) o sostituisce un terminale a un non terminale (se è del tipo $A \longrightarrow a$), ne consegue che una stringa di lunghezza k è generata da una derivazione di lunghezza $2k - 1$, per cui, per verificare se una qualche stringa di lunghezza al più 2^n è generabile dalla grammatica, è sufficiente considerare tutte le derivazioni di lunghezza al più $2^{n+1} - 1$, il cui numero è limitato superiormente dal valore $|\, P \,| \cdot 2^{|V_N|+1}$. □

Il metodo anzidetto è molto inefficiente ed è stato qui citato perché richiama quello utilizzato per dimostrare la decidibilità dell'analoga proprietà per i linguaggi regolari. Un metodo più efficiente per verificare se un non terminale genera stringhe terminali consiste nell'applicare l'Algoritmo 4.2 che elimina i non terminali non fecondi, verificando poi che l'assioma S non sia stato eliminato.

Teorema 4.14 *Data una grammatica $\mathcal{G}$ di tipo 2 è decidibile stabilire se $L(\mathcal{G})$ è infinito.*

Dimostrazione. Analogamente a quanto fatto nella dimostrazione del Teorema 4.13, assumiamo, senza perdita di generalità (vedi Teorema 4.4), che $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ sia una grammatica di tipo 2 in Forma Normale di Chomsky e si ponga $n = \mid V_N \mid$.

Ancora per il pumping lemma per i linguaggi non contestuali (Teorema 4.8), osserviamo che, se esiste una stringa $z = uvwxy \in L(\mathcal{G})$ con $2^n < \mid z \mid \leq 2^{2n}$, allora esistono infinite stringhe $z_i = uv^iwx^iy \in L(\mathcal{G})$, con $i \geq 0$.

Analogamente a quanto fatto nel Teorema 4.13, è sufficiente considerare tutte le derivazioni di lunghezza al più $2^{2(n+1)} - 1$, il cui numero è limitato superiormente dal valore $\mid P \mid \left(2^{(2|V_N|+1)}\right)$, e verificare se qualcuna di esse dà origine ad una stringa di terminali. □

Anche questo metodo risulta fortemente inefficiente. Un metodo più efficiente, di costo polinomiale rispetto alla dimensione della grammatica (in forma ridotta e priva di ε-produzioni), è basato sulla visita del grafo $G = (N, A)$ costruito come segue. L'insieme dei nodi N è in corrispondenza biunivoca con quello dei non terminali della grammatica; esiste inoltre un arco dal nodo n_i al nodo n_j se e solo se dal non terminale associato a n_i è possibile derivare una stringa contenente il non terminale associato al nodo n_j.

È possibile determinare se la grammatica genera un linguaggio infinito semplicemente verificando se il grafo introdotto è ciclico: tale proprietà è verificabile in tempo lineare (nel numero di nodi e di archi) applicando opportunamente i classici metodi di visita di grafi.

4.5 Automi a pila e linguaggi context free

Analogamente a quanto visto nel caso dei linguaggi regolari, per i quali sono stati introdotti gli automi a stati finiti, anche nel caso dei linguaggi non contestuali possiamo introdurre una macchina astratta che consente l'accettazione di tutti e soli i linguaggi di tale tipo. Questa macchina è chiamata *automa a pila*.

Definizione 4.6 *Un* automa a pila *(o automa push-down) è definito come una settupla* $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, F, \delta \rangle$ *dove* Σ *è l'*alfabeto di input*,* Γ *è l'*alfabeto dei simboli della pila*,* $Z_0 \in \Gamma$ *è il* simbolo iniziale di pila*,* Q *è un insieme finito e non vuoto di* stati*,* $q_0 \in Q$*] è lo* stato iniziale*,* $F \subseteq Q$ *è l'insieme degli* stati finali*,* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow Q \times \Gamma^*$ *è la* funzione (parziale) di transizione*.*

Osserviamo che l'automa sopra introdotto non è deterministico, in quanto, come già visto in Sezione 2.5.2, la presenza di ε-transizioni può comportare che esistano continuazioni diverse di una stessa computazione anche in presenza di uno stesso carattere letto.

Per ottenere un comportamento deterministico dobbiamo fare l'ulteriore ipotesi che se, per una coppia $q \in Q$, $Z \in \Gamma$, è definita $\delta(q, \varepsilon, Z)$ allora la funzione di transizione $\delta(q, a, Z)$ non deve essere definita per nessun $a \in \Sigma$.[2]

Dalla definizione della funzione di transizione deriva che ad ogni passo l'automa, a partire dallo stato attuale, dal carattere letto sul nastro di input e dal carattere affiorante sulla pila, sostituisce il simbolo affiorante nella pila con una stringa di caratteri[3] e si porta in un nuovo stato. La rappresentazione grafica degli automi a pila prevede quindi che venga anche riportata la pila ed i simboli in essa contenuti, come ad esempio in Figura 4.4.

Esempio 4.6 Se un automa a pila si trova in uno stato q_1, legge il carattere $a \in \Sigma$ dal nastro, ed il simbolo $A \in \Gamma$ affiora sulla pila, e se la funzione di transizione è tale che $\delta(q_1, a, A) = (q_3, BA)$, ne deriva che l'automa va nello stato q_3 sostituendo al simbolo A sulla pila la stringa BA. Tale transizione è riportata in Figura 4.5.

Esempio 4.7 Se un automa a pila si trova in uno stato q_1, legge il carattere $a \in \Sigma$ dal nastro, ed il simbolo $A \in \Gamma$ affiora sulla pila, e se la funzione di transizione è tale che $\delta(q_1, a, A) = (q_3, \varepsilon)$, ne deriva che l'automa va nello stato q_3 cancellando il simbolo A affiorante sulla pila. Tale transizione è riportata in Figura 4.6.

Per rappresentare la funzione di transizione di un automa a pila utilizzeremo una semplice estensione della tabella di transizione usata per gli automi a stati finiti, in cui le righe corrispondono agli stati dell'automa ed ogni colonna corrisponde ad una coppia composta dal carattere letto in ingresso e dal carattere affiorante sulla pila.

Come si può osservare il comportamento futuro di un automa a pila è determinato esclusivamente dallo stato attuale, dalla stringa da leggere e dal contenuto della pila. Da ciò, ricordando quanto detto nella Sezione 2.5, deriva la seguente definizione.

[2] Una caratterizzazione precisa degli automi a pila deterministici, sarà fornita successivamente, nella Sezione 4.6.

[3] La convenzione è che il primo carattere della stringa diventi il simbolo di pila affiorante. Si noti che se la stringa che viene inserita nella pila è la stringa vuota, ciò equivale a dire che il simbolo precedente affiorante nella pila è stato cancellato.

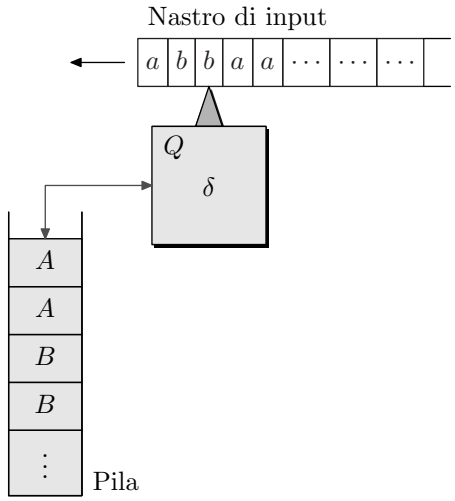


FIGURA 4.4 Struttura di un automa a pila.

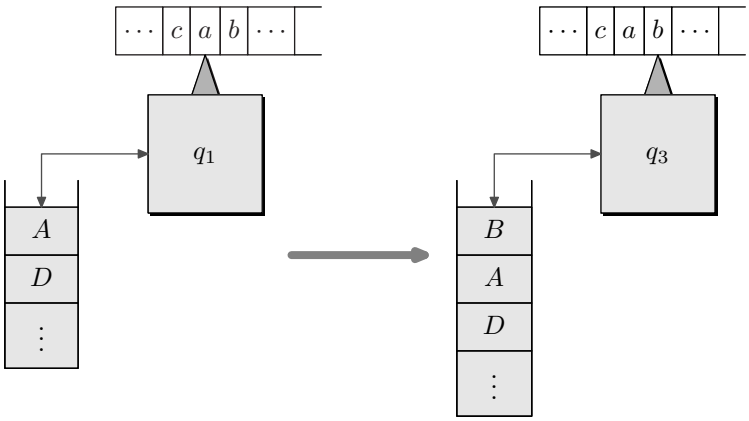


FIGURA 4.5 Esempio di transizione di un automa a pila per $\delta(q_1, a, A) = (q_3, BA)$.

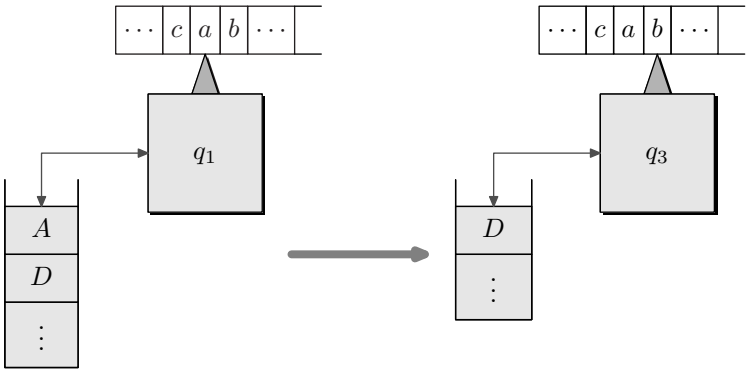


FIGURA 4.6 Esempio di transizione di un automa a pila per $\delta(q_1, a, A) = (q_3, \varepsilon)$.

Definizione 4.7 *Dato un automa a pila* $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, F, \delta \rangle$*, una* configurazione *di* $\mathcal{M}$ *è data dalla tripla* $\langle q, x, \gamma \rangle$*, dove* $q \in Q$*,* $x \in \Sigma^*$ *e* $\gamma \in \Gamma^*$*.*

La funzione di transizione δ permette di definire la relazione di transizione $\vdash$, che associa ad una configurazione la configurazione successiva, nel modo seguente.

Definizione 4.8 *Sia* $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, F, \delta \rangle$ *un automa a pila e due configurazioni* (q, x, γ) *e* (q', x', γ') *di* $\mathcal{M}$*, avremo che* $(q, x, \gamma) \underset{\mathcal{M}}{\vdash} (q', x', \gamma')$ *se e solo se valgono le tre condizioni:*

1. *esiste* $a \in \Sigma$ *tale che* $x = ax'$*;*
2. *esistono* $Z \in \Gamma$ *e* $\eta, \zeta \in \Gamma^*$ *tali che* $\gamma = Z\eta$ *e* $\gamma' = \zeta\eta$*;*
3. $\delta(q, a, Z) = (q', \zeta)$*.*

Come enunciato nella Sezione 2.5, una computazione è allora definita come una sequenza $c_0, \ldots, c_k$ di configurazioni di $\mathcal{M}$ tale che $c_i \underset{\mathcal{M}}{\vdash} c_{i+1}$.

Possiamo definire due diverse modalità di accettazione di stringhe da parte di un automa a pila.

Definizione 4.9 (Accettazione per pila vuota) *Sia* $\mathcal{M}$ *un automa a pila. Una configurazione* (q, x, γ) *di* $\mathcal{M}$ *è di accettazione se* $x = \gamma = \varepsilon$*. Secondo tale definizione, una stringa* x *è quindi* accettata *da* $\mathcal{M}$ *se e solo se al termine della scansione della stringa la pila è vuota. Indichiamo con* $N(\mathcal{M})$ *il linguaggio accettato per pila vuota dall'automa* $\mathcal{M}$*.*

Definizione 4.10 (Accettazione per stato finale) *Sia $\mathcal{M}$ un automa a pila. Una configurazione (q, x, γ) di $\mathcal{M}$ è di accettazione se $x = \varepsilon$ e $q \in F$. Secondo tale definizione, una stringa x è quindi* accettata *da $\mathcal{M}$ se e solo se al termine della scansione della stringa l'automa si trova in uno stato finale. Indichiamo con $L(\mathcal{M})$ il linguaggio accettato per stato finale dall'automa $\mathcal{M}$.*

Come vedremo, i due concetti sopra esposti consentono, almeno nel caso di automi a pila non deterministici che ora andiamo ad introdurre, di definire la stessa classe di linguaggi.

Forniamo ora la definizione più usuale di automa a pila non deterministico, che continueremo ad utilizzare nel seguito.

Definizione 4.11 *Un* automa a pila non deterministico *è definito come una settupla $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, F, \delta \rangle$ dove Σ è l'*alfabeto di input*, Γ è l'*alfabeto dei simboli della pila*, $Z_0 \in \Gamma$ è il* simbolo iniziale di pila*, Q è un insieme finito e non vuoto di* stati*, $q_0 \in Q$] è lo* stato iniziale*, $F \subseteq Q$ è l'insieme degli* stati finali*, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \mapsto \mathcal{P}(Q \times \Gamma^*)$ è la* funzione (parziale) di transizione*.*

Coerentemente alla definizione generale di automa non deterministico, un automa a pila non deterministico associa ad una configurazione un insieme di configurazioni successive, e quindi un insieme di continuazioni diverse per la computazione in corso.

A differenza del caso degli automi a stati finiti, per i quali il non determinismo non introduce maggiore potere computazionale (in quanto sia gli automi a stati deterministici che quelli non deterministici riconoscono la stessa classe di linguaggi, i linguaggi regolari), nel caso degli automi a pila la presenza del non determinismo comporta una aumento del potere computazionale: infatti, mentre gli automi a pila non deterministici riconoscono la classe dei linguaggi non contestuali, gli automi a pila deterministici riconoscono un sottoinsieme proprio di tali linguaggi, la classe dei linguaggi non contestuali deterministici.

Esempio 4.8 Consideriamo il caso dell'accettazione del linguaggio non contestuale $\{w\tilde{w} \mid w \in \{a, b\}^+\}$, dove ricordiamo che $\tilde{w}$ indica la stringa riflessa di w.

Nell'accettazione per pila vuota, si ha che la pila è vuota quando finisce la stringa x se e solo se $x \in L$. La funzione di transizione riportata in Tabella 4.1 descrive un automa a pila non deterministico che riconosce il linguaggio $\{w\tilde{w} \mid w \in \{a, b\}^+\}$ per pila vuota. Lo stesso linguaggio è accettato per stato finale dall'automa a pila non deterministico la cui funzione di transizione è descritta in Tabella 4.2 e $q_2 \in F$.

Esercizio 4.12 Realizzare gli automi a pila che riconoscono per pila vuota e per stato finale il linguaggio $\{a^n b^m \mid 1 \leq n \leq m\}$.

L'equivalenza, nel caso di automi a pila non deterministici, tra i due modelli di accettazione, per pila vuota e per stato finale, è mostrata dai due seguenti teoremi.

	A		B		Z_0	
	a	b	a	b	a	b
q_0	(q_0, BA) (q_1, ε)	(q_0, BA)	(q_0, AB)	(q_0, BB) (q_1, ε)	(q_0, A)	(q_0, B)
q_1	(q_1, ε)			(q_1, ε)		

Tabella 4.1 Tabella di transizione dell'automa a pila che accetta per pila vuota il linguaggio $\{w\tilde{w} \mid w \in \{a, b\}^+\}$.

	A		B		Z_0		
	a	b	a	b	a	b	ε
q_0	(q_0, AA) (q_1, ε)	(q_0, BA)	(q_0, AB)	(q_0, BB) (q_1, ε)	(q_0, AZ_0)	(q_0, BZ_0)	
q_1	(q_1, ε)			(q_1, ε)			(q_2, ε)

Tabella 4.2 Tabella di transizione dell'automa a pila che accetta per stato finale il linguaggio $\{w\tilde{w} \mid w \in \{a, b\}^+\}$.

Teorema 4.15 *Dato un automa a pila non deterministico $\mathcal{M}$ che accetta un linguaggio per stato finale, esiste un automa a pila non deterministico $\mathcal{M}'$ che accetta lo stesso linguaggio per pila vuota, vale a dire tale che $L(\mathcal{M}) = N(\mathcal{M}')$.*

Dimostrazione. Sia dato $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, F, \delta \rangle$: mostriamo come costruire l'automa $\mathcal{M}' = \langle \Sigma', \Gamma', Z_0', Q', q_0', \emptyset, \delta' \rangle$. Intuitivamente $\mathcal{M}'$ opererà similmente a $\mathcal{M}$, secondo lo schema seguente:

1. All'inizio $\mathcal{M}'$ ha nella pila un simbolo speciale X non appartenente a Γ ed inserisce al di sopra di X il simbolo Z_0;

2. Quindi, $\mathcal{M}'$ esegue gli stessi passi di $\mathcal{M}$. Si noti che nel corso di tale fase la pila di $\mathcal{M}'$ non sarà mai vuota;

3. Se, alla fine della stringa di input, $\mathcal{M}$ raggiunge uno stato finale, $\mathcal{M}'$ provvede ad eliminare tutti i simboli presenti in pila, incluso X.

In base a quanto anticipato, possiamo porre $\Sigma' = \Sigma$, $\Gamma' = \Gamma \cup \{X\}$ (X non in Γ), $Z_0' = X$, $Q' = Q \cup \{q_0', q_f\}$ (q_0' e q_f non in Q) e, per quanto riguarda la funzione di transizione δ':

1. $\forall q \in Q$, $\forall a \in \Sigma$, $\forall Z \in \Gamma$, $\delta'(q, a, Z) = \delta(q, a, Z)$;

2. $\forall q \in Q - F$, $\forall Z \in \Gamma$ $\delta'(q, \varepsilon, Z) = \delta(q, \varepsilon, Z)$;

3. $\forall q \in F$, $\forall Z \in \Gamma$ $\delta'(q, \varepsilon, Z) = \delta(q, \varepsilon, Z) \cup \{(q_f, \varepsilon)\}$;

4. $\delta'(q_0', \varepsilon, X) = \{(q_0, Z_0 X)\}$;

5. $\forall Z \in \Gamma'$, $\delta'(q_f, \varepsilon, Z) = \{(q_f, \varepsilon)\}$.

Per effetto delle prime tre condizioni $\mathcal{M}'$ simula perfettamente $\mathcal{M}$ se questo non si trova in una stato finale. Altrimenti $\mathcal{M}'$ può eseguire tutte le transizioni definite per $\mathcal{M}$ con l'ulteriore possibilità di effettuare una ε-transizione verso il suo stato q_f lasciando la pila immutata. La quarta condizione fa sì che il simbolo X rimanga in fondo alla pila mentre l'automa $\mathcal{M}'$ effettua la simulazione di $\mathcal{M}$, mentre la quinta condizione assicura lo svuotamento della pila di $\mathcal{M}'$ nel caso in cui $\mathcal{M}$ raggiunga uno stato finale.

È facile verificare che il linguaggio accettato per stato finale da $\mathcal{M}$ coincide con quello accettato per pila vuota da $\mathcal{M}'$. □

Teorema 4.16 *Dato un automa a pila non deterministico $\mathcal{M}$ che accetta un linguaggio per pila vuota, esiste un automa a pila non deterministico $\mathcal{M}'$ che accetta lo stesso linguaggio per stato finale, vale a dire tale che $N(\mathcal{M}) = L(\mathcal{M}')$.*

Dimostrazione. La tecnica di dimostrazione è analoga a quella utilizzata nel teorema precedente. Sia dato $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, \emptyset, \delta \rangle$: mostriamo come costruire l'automa $\mathcal{M}' = \langle \Sigma', \Gamma', Z_0', Q', q_0', \{q_f\}, \delta' \rangle$. Intuitivamente $\mathcal{M}'$ opererà similmente a $\mathcal{M}$, secondo lo schema seguente:

1. All'inizio $\mathcal{M}'$ ha nella pila un simbolo speciale X non appartenente a Γ ed inserisce Z_0 al di sopra di X;
2. Quindi, $\mathcal{M}'$ esegue gli stessi passi di $\mathcal{M}$. Si noti che nel corso di tale fase la pila di $\mathcal{M}'$ non sarà mai vuota;
3. Se, alla fine della stringa di input, $\mathcal{M}$ raggiunge la condizione di pila vuota, $\mathcal{M}'$ entra nello stato finale q_f.

In base a quanto anticipato, possiamo porre $\Sigma' = \Sigma$, $\Gamma' = \Gamma \cup \{X\}$ (X non in Γ), $Z_0' = X$, $Q' = Q \cup \{q_0', q_f\}$ (q_0' e q_f non in Q) e, per quanto riguarda la funzione di transizione δ':

1. $\forall q \in Q$, $\forall a \in \Sigma \cup \{\varepsilon\}$, $\forall Z \in \Gamma$, $\delta'(q, a, Z) = \delta(q, a, Z)$;
2. $\forall q \in Q$, $\delta'(q, \varepsilon, X) = \{(q_f, X)\}$;
3. $\delta'(q_0', \varepsilon, X) = \{(q_0, Z_0 X)\}$.

Per effetto della prima condizione $\mathcal{M}'$ simula perfettamente $\mathcal{M}$ se la pila di questo non è vuota. Altrimenti, per effetto della seconda condizione, $\mathcal{M}'$ effettua una ε-transizione verso il suo stato finale q_f lasciando la pila immutata. La terza condizione fa sì che il simbolo X rimanga in fondo alla pila mentre l'automa $\mathcal{M}'$ effettua la simulazione di $\mathcal{M}$.

È facile verificare che il linguaggio accettato per pila vuota da $\mathcal{M}$ coincide con quello accettato per stato finale da $\mathcal{M}'$. □

L'introduzione del modello di accettazione per pila vuota, che risulta meno intuitivo rispetto a quello di accettazione per stato finale, deriva dal fatto che esso è particolarmente utile nella dimostrazione di teoremi di caratterizzazione del potere computazionale degli automi a pila, come si vedrà nel seguito della sezione.

Introduciamo ora due teoremi che mostrano l'equivalenza tra i linguaggi accettati da automi a pila e linguaggi generati da grammatiche non contestuali. Come su vedrà, in entrambi i casi faremo riferimento alla condizione di accettazione mediante pila vuota.

Teorema 4.17 *Se un linguaggio è generato da una grammatica $\mathcal{G}$ non contestuale, esiste un automa a pila $\mathcal{M}$ tale che $L(\mathcal{G}) = \mathcal{N}(\mathcal{M})$.*

Dimostrazione. La dimostrazione è costruttiva: a partire da $\mathcal{G}$ deriviamo $\mathcal{M}$ e mostriamo poi l'equivalenza. Consideriamo dapprima il caso in cui

$\varepsilon \notin L(\mathcal{G})$. A partire da $\mathcal{G}$, costruiamo una grammatica $\mathcal{G}' = \langle \Sigma, V'_N, P', S' \rangle$ equivalente a $\mathcal{G}$, ma in Forma Normale di Greibach. Poniamo $\Gamma = V'_N$, $Q = \{q_0\}$, $Z_0 = S'$. Per ogni produzione $A \longrightarrow a\gamma$, $\gamma \in (V'_N)^*$ introduciamo la regola $\delta(q_0, a, A) = (q_0, \gamma)$.

Proviamo ora il seguente asserto generale:

$$(q, x, S') \overset{*}{\vdash\!\!-\!\!-} (q, \varepsilon, \alpha) \tag{4.1}$$

se e solo se $S' \overset{*}{\Longrightarrow} x\alpha$, dove $q \in Q$, $x \in (\Sigma')^*$ e $\alpha \in (V'_N)^*$.

Mostriamo dapprima, per induzione su i, numero di passi della computazione, che

$$\text{se } (q, x, S') \overset{i}{\vdash\!\!-\!\!-} (q, \varepsilon, \alpha) \quad \text{allora} \quad S' \overset{*}{\Longrightarrow} x\alpha.$$

Passo base $(i = 1)$

Se $(q, x, S') \vdash\!\!-\!\!- (q, \varepsilon, \alpha)$ allora, per come è stato costruito l'automa a pila, deve esistere in P' una produzione $S' \longrightarrow x\alpha$, con $x \in \Sigma$.

Passo induttivo $(i > 1)$

In tal caso, posto $x = ya$, con $a \in \Sigma$, possiamo riscrivere le derivazioni nel seguente modo:

$$(q, ya, S') \overset{i-1}{\vdash\!\!-\!\!-} (q, a, \beta) \vdash\!\!-\!\!- (q, \varepsilon, \alpha),$$

da cui

$$(q, y, S') \overset{i-1}{\vdash\!\!-\!\!-} (q, \varepsilon, \beta).$$

Per ipotesi induttiva vale $S' \overset{*}{\Longrightarrow} y\beta$. D'altra parte la transizione

$$(q, a, \beta) \vdash\!\!-\!\!- (q, \varepsilon, \alpha)$$

implica che $\beta = A\gamma$, $\alpha = \eta\gamma$, per un certo $\gamma \in (V'_N)^*$, $\eta \in (V'_N)^*$ e $A \in V'_N$ e che in P' vi è la produzione $A \longrightarrow a\eta$. Possiamo perciò concludere che $S' \overset{*}{\Longrightarrow} y\beta \Longrightarrow ya\eta\gamma = x\alpha$.

Mostriamo ora, per induzione sul numero di passi della derivazione, che vale anche la seguente implicazione:

$$\text{se } S' \overset{i}{\Longrightarrow} x\alpha \quad \text{allora} \quad (q, x, S') \overset{*}{\vdash\!\!-\!\!-} (q, \varepsilon, \alpha).$$

Passo base ($i = 1$)

Se $S' \Longrightarrow x\alpha$ (con $x \in \Sigma$) deve esistere in P' la produzione $S' \longrightarrow x\alpha$: allora, per costruzione, esiste una transizione dell'automa tale che $(q, x, S') \vdash\!\!\!- (q, \varepsilon, \alpha)$.

Passo induttivo ($i > 1$)

Posto $x = ya$, con $a \in \Sigma$, supponiamo di avere la derivazione sinistra

$$S' \overset{i-1}{\Longrightarrow} yA\gamma \Longrightarrow ya\eta\gamma, \tag{4.2}$$

con $\alpha = \eta\gamma$ ed $\eta, \gamma \in (V'_N)^*$. Per ipotesi induttiva vale allora

$$(q, y, S') \overset{*}{\vdash\!\!\!-} (q, \varepsilon, A\gamma);$$

ne segue che vale anche $(q, ya, S') \overset{*}{\vdash\!\!\!-} (q, a, A\gamma)$. Poiché $A \longrightarrow a\eta$ è una produzione in P' (vedi la (4.2)) segue, per costruzione, che $(q, a, A\gamma) \vdash\!\!\!- (q, \varepsilon, \eta\gamma)$. Da ciò deriva che

$$(q, x, S) \overset{*}{\vdash\!\!\!-} (q, a, A\gamma) \vdash\!\!\!- (q, \varepsilon, \alpha).$$

Si osservi ora che, come caso particolare dell'asserto sopra dimostrato, abbiamo che, ponendo nella (4.1) $\alpha = \varepsilon$ otteniamo

$$S' \Longrightarrow x \text{ se e solo se } (q, x, S') \overset{*}{\vdash\!\!\!-} (q, \varepsilon, \varepsilon).$$

e quindi l'enunciato del teorema risulta dimostrato per ogni stringa $x \neq \varepsilon$.

Osserviamo infine che se $\varepsilon \in L(\mathcal{G})$ possiamo facilmente costruire una grammatica $\mathcal{G}'$ in GNF che genera il linguaggio $L(\mathcal{G}') = L(\mathcal{G}) - \{\varepsilon\}$ e quindi applicare la procedura descritta per ottenere un automa a pila $\langle \Sigma, \Gamma, Z_0, \delta, \{q_0\}, \emptyset \rangle$ che riconosce $L(\mathcal{G}')$. Da questo è facile arrivare ad un automa che riconosce $L(\mathcal{G})$: basta aggiungere uno stato q'_0, che diventa lo stato iniziale, su cui sarà definita la transizione $\delta(q'_0, \varepsilon, Z_0) = \{(q'_0, \varepsilon), (q_0, Z_0)\}$. □

Il prossimo risultato ci permette di verificare che gli automi a pila non deterministici caratterizzano effettivamente i linguaggi di tipo 2. Infatti, in esso si dimostra che ogni linguaggio accettato da un automa a pila non deterministico è non contestuale.

Teorema 4.18 *Sia L un linguaggio accettato mediante pila vuota da un automa a pila $\mathcal{M} = \langle \Sigma, \Gamma, Q, \delta, Z_0, q_0, \emptyset \rangle$, allora esiste una grammatica non contestuale $\mathcal{G}$ che lo genera, cioè $L = N(\mathcal{M}) = L(\mathcal{G})$.*

Dimostrazione. La dimostrazione è costruttiva, e consiste nel provare che il linguaggio generato dalla grammatica $\mathcal{G} = \langle \Sigma, V_N, P, S \rangle$, definita come appresso specificato, è proprio L. L'alfabeto Σ è scelto uguale a quello dell'automa a pila.

Definiamo

$$V_N = \{[q, A, p] \mid \text{ per ogni } q, p \in Q, A \in \Gamma\} \cup \{S\}.$$

Definiamo le produzioni P al seguente modo

- $S \longrightarrow [q_0, Z_0, p]$, per ogni $p \in Q$;
- per ogni $\delta(q, a, A)$ e per ogni $(q_1, B_1 \dots B_m) \in \delta(q, a, A)$, in cui $q, q_1 \in Q$, $A, B_1, \dots, B_m \in \Gamma$, $a \in \Sigma \cup \{\varepsilon\}$, introduciamo le produzioni

$$[q, A, q_{m+1}] \longrightarrow a[q_1 B_1 q_2] \dots [q_m B_m q_{m+1}] \text{ se } m > 0,$$
$$[q, A, q_1] \longrightarrow a \text{ altrimenti ;}$$

dove $q_2, \dots, q_{m+1} \in Q$.

Per comprendere la dimostrazione si può osservare che la definizione della grammatica è tale che una derivazione di una stringa x "simula" il comportamento di un automa a pila che accetta la stringa x. In particolare, se da $[q, A, p]$ deriviamo x, vuole dire che, a seguito di una sequenza di transizioni, la lettura della stringa x comporta la cancellazione del non terminale A dalla pila.

Più formalmente, dobbiamo dimostrare che

$$[q, A, p] \stackrel{*}{\Longrightarrow} x \quad \text{se e solo se} \quad (q, x, A) \stackrel{*}{\vdash} (p, \varepsilon, \varepsilon)$$

da cui otteniamo come caso particolare, che $[q_0, Z_0, p] \stackrel{*}{\Longrightarrow} x$ se e solo se $(q_0, x, Z_0) \stackrel{*}{\vdash} (p, \varepsilon, \varepsilon)$ per qualche p, cioè se e solo se l'automa accetta la stringa x per pila vuota.

Proviamo dapprima per induzione sul numero di passi che se $(q, x, A) \stackrel{i}{\vdash} (p, \varepsilon, \varepsilon)$, allora $[q, A, p] \stackrel{*}{\Longrightarrow} x$.

Passo base $(i = 1)$

Se $(q, x, A) \vdash (p, \varepsilon, \varepsilon)$ allora deve esistere per costruzione la produzione $[q, A, p] \longrightarrow x$.

Passo induttivo $(i > 1)$

Sia $x = ay$ e supponiamo che $(q_1, B_1 \dots B_m) \in \delta(q, a, A)$, per cui

$$(q, ay, A) \vdash (q_1, y, B_1 \dots B_m) \stackrel{i-1}{\vdash} (p, \varepsilon, \varepsilon)$$

Sia $y = y_1 \dots y_m$ dove ogni y_i determina la cancellazione di B_i dalla pila. Esistono stati $q_1, q_2, \dots, q_m$ tali che

$$\begin{array}{lcl} (q_1, y_1, B_1) & \overset{k_1}{\vdash} & (q_2, \varepsilon, \varepsilon) \\ (q_2, y_2, B_2) & \overset{k_2}{\vdash} & (q_3, \varepsilon, \varepsilon) \\ & \vdots & \\ (q_m, y_m, B_m) & \overset{k_m}{\vdash} & (q_{m+1}, \varepsilon, \varepsilon) \end{array}$$

(con $k_j \leq i - 1$ – vedi Figura 4.7). Per induzione a ciascuna di queste

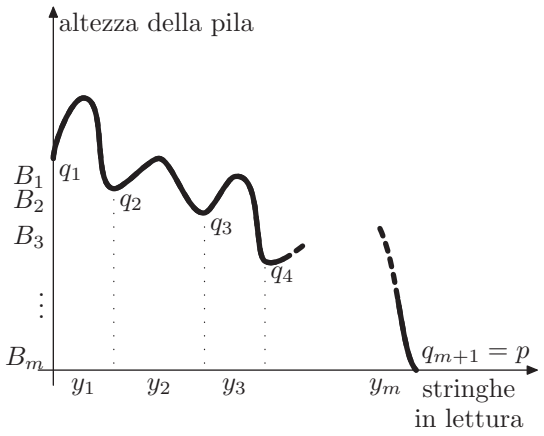


FIGURA 4.7 Altezza della pila durante la lettura della stringa $y = y_1 \dots y_m$..

computazioni deve corrispondere una derivazione:

$$\begin{array}{lcl} [q_1, B_1, q_2] & \overset{*}{\Longrightarrow} & y_1 \\ & \vdots & \\ [q_m, B_m, q_{m+1}] & \overset{*}{\Longrightarrow} & y_m \end{array}$$

e quindi, poiché abbiamo anche la produzione

$$[q, A, p] \longrightarrow a[q_1, B_1, q_2] \dots [q_m, B_m, p]$$

otteniamo

$$[q, A, p] \overset{*}{\Longrightarrow} a y_1 \dots y_m = ay = x.$$

Proviamo ora, sempre per induzione sul numero di passi, che vale anche:

$$\text{se } [q, A, p] \overset{i}{\Longrightarrow} x \quad \text{allora} \quad (q, x, A) \overset{*}{\longmapsto} (p, \varepsilon, \varepsilon).$$

Passo base $(i = 1)$
In questo caso abbiamo la produzione $[q, A, p] \longrightarrow x$ che, per costruzione, comporta l'esistenza di $(p, \varepsilon) \in \delta(q, x, A)$.

Passo induttivo $(i > 1)$
Per applicare l'induzione possiamo osservare che la derivazione di lunghezza i $[q, A, p] \overset{i}{\Longrightarrow} x$ può essere riscritta come:

$$[q, A, p] \Longrightarrow a[q_1 B_1 q_2] \dots [q_{n-1} B_{n-1} q_n][q_n B_n q_{n+1}] \overset{i-1}{\Longrightarrow} x,$$

con $q_{n+1} = p$. Decomponiamo ora x in n stringhe $x_1, \dots, x_n$ tali che

$$[q_j B_j q_{j+1}] \overset{k_j}{\Longrightarrow} x_j, \quad \text{con } k_j \leq i - 1, \quad j = 1, 2, \dots, n.$$

Per ipotesi induttiva valgono le relazioni seguenti:

$$(q_j, x_j, B_j) \overset{*}{\longmapsto} (q_{j+1}, \varepsilon, \varepsilon), \quad j = 1, \dots, n$$

e, quindi, anche le

$$(q_j, x_j, B_j \dots B_n) \overset{*}{\longmapsto} (q_{j+1}, \varepsilon, B_{j+1} \dots B_n), \quad j = 1, \dots, n. \quad (4.3)$$

Dal primo passo nella derivazione di x sappiamo che

$$(q, x, A) \longmapsto (q_1, x_1 \dots x_n, B_1 \dots B_n).$$

In base alla relazione 4.3 possiamo scrivere che

$$(q_1, x_1 \dots x_n, B_1 \dots B_n) \overset{*}{\longmapsto} (q_2, x_2 \dots x_n, B_2 \dots B_n) \overset{*}{\longmapsto} \cdots$$
$$\cdots \overset{*}{\longmapsto} (q_n, x_n, B_n) \overset{*}{\longmapsto} (p, \varepsilon, \varepsilon),$$

il che conclude la prova.

□

Esempio 4.9 Consideriamo l'automa a pila avente la funzione di transizione seguente:

$$\delta(q_0, 0, Z_0) = \{(q_0, XZ_0)\}$$
$$\delta(q_0, 0, X) = \{(q_0, XX)\}$$
$$\delta(q_0, 1, X) = \{(q_1, \varepsilon)\}$$
$$\delta(q_1, 1, X) = \{(q_1, \varepsilon)\}$$
$$\delta(q_1, \varepsilon, X) = \{(q_1, \varepsilon)\}$$
$$\delta(q_1, \varepsilon, Z_0) = \{(q_1, \varepsilon)\}$$

Tale automa riconosce il linguaggio $\{0^n 1^m \mid n \geq m \geq 1\}$.

Applicando il metodo illustrato nella dimostrazione del Teorema 4.18, otteniamo la seguente grammatica non contestuale che genera lo stesso linguaggio:

$$S \longrightarrow [q_0 Z_0 q_0] \qquad \text{(non feconda)}$$

$$S \longrightarrow [q_0 Z_0 q_1]$$

$$[q_0 Z_0 q_0] \longrightarrow 0[q_0 X q_0][q_0 Z_0 q_0] \quad \text{(non feconda)}$$

$$[q_0 Z_0 q_0] \longrightarrow 0[q_0 X q_1][q_1 Z_0 q_0] \quad \text{(non feconda)}$$

$$[q_0 Z_0 q_1] \longrightarrow 0[q_0 X q_0][q_0 Z_0 q_1] \quad \text{(non feconda)}$$

$$[q_0 Z_0 q_1] \longrightarrow 0[q_0 X q_1][q_1 Z_0 q_1]$$

$$[q_0 X q_0] \longrightarrow 0[q_0 X q_0][q_0 X q_0] \quad \text{(non feconda)}$$

$$[q_0 X q_0] \longrightarrow 0[q_0 X q_1][q_1 X q_0] \quad \text{(non feconda)}$$

$$[q_0 X q_1] \longrightarrow 0[q_0 X q_0][q_0 X q_1] \quad \text{(non feconda)}$$

$$[q_0 X q_1] \longrightarrow 0[q_0 X q_1][q_1 X q_1]$$

$$[q_0 X q_1] \longrightarrow 1$$

$$[q_1 X q_1] \longrightarrow 1$$

$$[q_1 X q_1] \longrightarrow \varepsilon$$

$$[q_1 Z_0 q_1] \longrightarrow \varepsilon.$$

Verificando se i non terminali sono raggiungibili e/o fecondi possiamo giungere alla forma ridotta seguente:

$$S \longrightarrow [q_0 Z_0 q_1]$$
$$[q_0 Z_0 q_1] \longrightarrow 0[q_0 X q_1][q_1 Z_0 q_1]$$
$$[q_0 X q_1] \longrightarrow 0[q_0 X q_1][q_1 X q_1] \mid 1$$
$$[q_1 X q_1] \longrightarrow 1 \mid \varepsilon$$
$$[q_1 Z_0 q_1] \longrightarrow \varepsilon$$

ed eliminando la produzione unitaria:

$$
\begin{aligned}
&S \longrightarrow 0[q_0Xq_1][q_1Z_0q_1]\\
&[q_0Xq_1] \longrightarrow 0[q_0Xq_1][q_1Xq_1] \mid 1\\
&[q_1Xq_1] \longrightarrow 1 \mid \varepsilon\\
&[q_1Z_0q_1] \longrightarrow \varepsilon.
\end{aligned}
$$

Applicando le produzioni possiamo vedere come viene simulato l'automa. Supponiamo di dover generare la stringa 00011.

- **Generazione**

$$
\begin{aligned}
&S\\
&0 \ldots\ldots\ldots\ldots\ldots\ldots [q_0Xq_1][q_1Z_0q_1]\\
&00 \ldots\ldots\ldots\ldots [q_0Xq_1][q_1Xq_1][q_1Z_0q_1]\\
&000 \ldots [q_0Xq_1][q_1Xq_1][q_1Xq_1][q_1Z_0q_1]\\
&0001 \ldots\ldots\ldots [q_1Xq_1][q_1Xq_1][q_1Z_0q_1]\\
&00011 \ldots\ldots\ldots\ldots\ldots [q_1Xq_1][q_1Z_0q_1]\\
&00011 \ldots\ldots\ldots\ldots\ldots\ldots\ldots [q_1Z_0q_1]\\
&00011 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \varepsilon
\end{aligned}
$$

- **Accettazione**

stato	caratteri letti	caratteri da leggere	pila
q_0		0	Z_0
q_0	0	0	XZ_0
q_0	00	0	XXZ_0
q_0	000	1	$XXXZ_0$
q_1	0001	1	XXZ_0
q_1	00011	ε	XZ_0
q_1	00011	ε	Z_0
q_1	00011	ε	ε

Esercizio 4.13 Definire un automa a pila non deterministico che riconosca il linguaggio $\{w\tilde{w} \mid w \in \{a,b\}^*\}$ e costruire la grammatica che lo genera in base al metodo indicato nel teorema precedente.

4.6 Automi a pila deterministici

Il modello degli automi a pila è stato definito come non deterministico e in effetti, come abbiamo visto nei Teoremi 4.17 e 4.18, l'equivalenza tra linguaggi

non contestuali e linguaggi accettati da automi a pila è stata dimostrata appunto considerando automi non deterministici. Il modello degli automi a pila deterministici, che qui introduciamo, risulta comunque di grande rilevanza in quanto, pur non consentendo di riconoscere tutti i linguaggi context free, viene ampiamente utilizzato quale modello di riferimento nell'analisi sintattica di programmi (vedi Sezione 4.7).

Definizione 4.12 *Un automa a pila* deterministico *è un automa a pila* $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, F, \delta \rangle$ *tale che,* $\forall a \in \Sigma$, $\forall Z \in \Gamma$, $\forall q \in Q$

$$\mid \delta(q, a, Z) \mid + \mid \delta(q, \varepsilon, Z) \mid \leq 1. \tag{4.4}$$

La condizione che deve essere soddisfatta dalla funzione di transizione di un automa a pila affinché questo risulti deterministico ha natura duplice: anzitutto, la cardinalità di $\delta(q, a, Z)$ non deve essere superiore a 1 e, in secondo luogo, se è definita una ε-transizione su un certo stato e per un certo simbolo di pila affiorante non deve essere contemporaneamente definita altra transizione per quello stato e quel simbolo di pila. Sebbene in questa sede non approfondiamo lo studio degli automi a pila deterministici, è bene chiarire, anche se informalmente, come tali dispositivi effettuano il calcolo. In particolare, la computazione di un automa a pila deterministico va intesa come una sequenza massimale di configurazioni, nel senso che essa procede fintanto che esistono transizioni definite. Una stringa è accettata da un automa a pila deterministico se e solo se essa dà luogo ad una computazione che termina in una configurazione $\langle q, \varepsilon, w \rangle$, con $q \in F$ e $w \in \Gamma^*$.[4] Si può verificare che gli automi a pila deterministici hanno un potere computazionale strettamente inferiore a quello degli automi a pila non deterministici osservando quanto segue.

1. Poiché la definizione di automa a pila deterministico è una specializzazione della definizione di automa a pila non deterministico la classe dei linguaggi accettati da automi a pila non deterministici include quella dei linguaggi accettati da automi a pila deterministici.

2. La classe dei linguaggi di tipo 2, vale a dire dei linguaggi accettati da automi a pila non deterministici, non è chiusa rispetto alla complementazione (vedi Esercizio 4.11).

3. La classe dei linguaggi accettati da automi a pila deterministici è chiusa rispetto alla complementazione. Al fine di mostrare tale proprietà descriviamo come, per ogni automa a pila deterministico $\mathcal{M} = \langle \Sigma, \Gamma, Z_0, Q, q_0, F, \delta \rangle$, sia possibile costruirne uno che accetta il linguaggio $\Sigma^* - L(\mathcal{M})$.

[4] È possibile dimostrare che, nel caso di automi a pila deterministici, le due modalità di accettazione viste, per stato finale e per pila vuota, non sono equivalenti: in particolare, esistono linguaggi accettabili da automi a pila deterministici per stato finale e che non sono accettati da alcun automa a pila deterministico per pila vuota.

Osserviamo che, data una stringa x di input, l'automa $\mathcal{M}$ può accettare x, rifiutarla o eseguire indefinitamente un ciclo di ε-transizioni. È tuttavia facile verificare che, per ogni automa a pila deterministico $\mathcal{M}$, ne esiste uno equivalente che termina ogni sua computazione. Assumeremo dunque, senza perdita di generalità, che $\mathcal{M}$ accetti o rifiuti ogni stringa che gli viene sottoposta. Possiamo allora costruire l'automa $\mathcal{M}' = \langle \Sigma, \Gamma, Z_0, Q', q_0, F', \delta' \rangle$ che accetta $\Sigma^* - L(\mathcal{M})$ semplicemente ponendo $Q' = Q \cup \{q_F\}$, $q_F \notin Q$, $F' = Q' - F$ e, per ogni $a \in \Sigma \cup \{\varepsilon\}$, $A \in \Gamma$, $q \in Q$:

(a) $\delta'(q, a, A) = (q_F, A)$, se $\delta(q, a, A)$ è indefinita,

(b) $\delta'(q, a, A) = \delta(q, a, A)$, altrimenti,

e ponendo inoltre $\delta'(q_F, a, A) = (q_F, A)$.

La classe dei linguaggi accettati da automi a pila deterministici dunque non coincide con quella dei linguaggi di tipo 2. Intuitivamente, un linguaggio separatore è dato da $L = \{w\tilde{w} \mid w \in \{a, b\}^+\}$, già precedentemente considerato (vedi Esempio 4.8). Tale linguaggio, di tipo 2, non può essere accettato da alcun automa a pila deterministico. Infatti, intuitivamente, durante la scansione dell'input, non è possibile individuare a priori, in maniera deterministica, dove termina la stringa w ed inizia $\tilde{w}$.

Da quanto osservato si evince che la classe dei linguaggi accettati da automi a pila deterministici è propriamente contenuta in quella dei linguaggi di tipo 2.

Esercizio 4.14 Dimostrare che la classe dei linguaggi context free deterministici non è chiusa rispetto a intersezione ed unione.

4.7 Analisi sintattica e linguaggi non contestuali deterministici

Come già osservato nella Sezione 2.4, la sintassi dei linguaggi di programmazione può essere specificata tramite una grammatica e ciò che viene tipicamente indicato come "programma sintatticamente corretto" altro non è che una stringa del linguaggio generato da tale grammatica. D'altra parte, il processo di traduzione di un programma consiste sostanzialmente, come è noto, nella costruzione del programma oggetto o eseguibile che corrisponde a un programma sorgente dato, tipicamente scritto in un linguaggio di programmazione ad alto livello. L'automazione di tale processo, argomento non approfondito in questo volume, richiede la conoscenza della grammatica formale che definisce il linguaggio di programmazione utilizzato.

Tralasciando vari dettagli implementativi, possiamo assumere che la traduzione di un programma consista in una attività preliminare di analisi e in una

successiva attività di generazione del codice. Nella fase di analisi (si parla in effetti di *analisi sintattica*, o *parsing*) viene esaminata la struttura sintattica del programma; in pratica, oltre a risolvere il problema del riconoscimento del programma come parola del linguaggio di programmazione, viene ricostruito l'albero di derivazione del programma, individuando una sequenza di produzioni della grammatica associata a tale linguaggio, la cui applicazione permette di generare il programma stesso a partire dall'assioma. A valle di tale analisi si procede con la generazione del codice, resa possibile proprio dalla conoscenza della struttura sintattica del programma, sotto forma del relativo albero sintattico.

I linguaggi di programmazione di interesse pratico possono essere considerati, in prima approssimazione, linguaggi di tipo 2. In realtà ci sono alcuni elementi sintattici, quali ad esempio la obbligatorietà e l'unicità della dichiarazione di una variabile, che non possono essere espressi con produzioni di tipo 2: formalmente, essi rendono tali linguaggi di tipo 1. L'analisi sintattica di una stringa di un linguaggio di tipo 1 è tuttavia un problema assai complesso, la cui soluzione comporta generalmente computazioni assai onerose. Per fortuna, è possibile individuare ed isolare gli aspetti contestuali della sintassi dei linguaggi di programmazione, in modo da suddividere la fase di parsing in due sottoprocessi (che tuttavia non lavorano in modo necessariamente sequenziale): nel primo si considerano solo le regole di tipo 2, nel secondo si prendono esplicitamente in esame gli aspetti contestuali utilizzando tecniche algoritmiche ad hoc.

È importante osservare che il linguaggio di tipo 2 suddetto deve essere deterministico: questa caratteristica rende possibile la costruzione di algoritmi efficienti per il parsing; per i linguaggi di tipo 2 non deterministici il problema è già troppo complesso. Infatti, un linguaggio di tipo 2 deterministico può essere accettato da un automa a pila deterministico in un numero lineare di passi, mentre, come vedremo nella Sezione 4.9, in generale gli algoritmi per il riconoscimento di un linguaggio context free possono richiedere tempo cubico. Anche se il problema dell'analisi sintattica appare più complesso, rimane tuttavia possibile costruire algoritmi che effettuano tale analisi operando una sola scansione del testo in ingresso, limitandosi a considerare in ogni momento un numero costante prefissato di simboli.

Il parsing dei linguaggi non contestuali deterministici è un problema classico nell'ambito dei linguaggi formali, e per esso sono state sviluppate varie tecniche algoritmiche, normalmente basate sull'idea di disporre di grammatiche le cui produzioni, oltre ad essere non contestuali, hanno una struttura predefinita che può essere sfruttata in pratica al fine di ricostruire efficientemente l'albero di derivazione del programma.

Nel seguito illustreremo, a titolo puramente esemplificativo e senza alcuna pretesa di completezza, i due metodi più classici di analisi sintattica: l'analisi discendente e l'analisi ascendente.

Esempio 4.10 Un analizzatore sintattico a *discesa ricorsiva* è un algoritmo di par-

sing che costruisce un albero di derivazione con metodo discendente (*top-down*) simulando l'intero processo di derivazione. In particolare, l'algoritmo legge uno o più caratteri della stringa da analizzare e individua quale sia la prima produzione da applicare all'assioma allo scopo di generare la stringa. Quindi, dopo l'applicazione di tale produzione, l'algoritmo legge altri caratteri della stringa e determina quale sia la produzione successiva, da applicare al non terminale più a sinistra[5] presente nella forma di frase corrente. La tecnica viene quindi reiterata fino al termine della scansione dell'input, quando la ricostruzione dell'albero sintattico è completa.

Una grammatica non contestuale è detta *grammatica LL(k)*[6] se esiste un analizzatore sintattico a discesa ricorsiva tale che l'esame dei prossimi k caratteri dell'input consente il riconoscimento della produzione da applicare alla forma di frase corrente. Un linguaggio non contestuale è detto *linguaggio LL(k)* se ammette una grammatica non contestuale LL(k). La denominazione "analisi a discesa ricorsiva" è dovuta al fatto che, data una grammatica LL(k), è possibile scrivere un programma ricorsivo che, esaminando ad ogni passo i prossimi k caratteri dell'input, costruisce, a partire dall'assioma, una derivazione sinistra della stringa in esame se questa appartiene al linguaggio o la rifiuta se essa non appartiene al linguaggio; in tal caso l'analizzatore è in grado di fornire anche qualche precisazione (diagnostica) sul motivo del rifiuto.

Data ad esempio la grammatica $\mathcal{G} = \langle\{a,b,c\},\{S,\},P,S\rangle$ le cui regole di produzione sono

$$\begin{array}{lcl} S & \longrightarrow & aSa \\ S & \longrightarrow & bSb \\ S & \longrightarrow & c \end{array}$$

è evidente che essa genera il linguaggio $L(\mathcal{G}) = \{wc\tilde{w} \mid w \in \{a,b\}^*\}$. Data allora la stringa $x = abcba$ è facile convincersi che un analizzatore a discesa ricorsiva individua la seguente derivazione sinistra per x esaminando un carattere alla volta:

$$S \implies aSb \implies abSba \implies abcba$$

$\mathcal{G}$ è dunque una grammatica LL(1). Si noti che la stessa proprietà non è goduta dalla seguente grammatica $\mathcal{G}' = \langle\{a,b,c\},\{S,\},P',S\rangle$ equivalente a $\mathcal{G}$:

$$\begin{array}{lcl} S & \longrightarrow & aaAaa \mid abAba \mid T \mid aca \mid bcb \mid c \\ T & \longrightarrow & bbBbb \mid baBab \mid S \\ A & \longrightarrow & S \mid aca \mid bcb \mid c \\ B & \longrightarrow & T \mid aca \mid bcb \mid c \end{array}$$

Per tale grammatica si rende necessario l'esame di due caratteri alla volta allo scopo di individuare la produzione da applicare. Perciò, benché $L(\mathcal{G}')$ sia LL(1), risulta che $\mathcal{G}'$ è LL(2).

---

[5] Poiché molte sono le derivazioni differenti associate a uno stesso albero sintattico, nella sua ricostruzione top-down è utile considerare la cosiddetta *derivazione sinistra*, in cui ad ogni passo si applica una produzione al non terminale più a sinistra. Sotto opportune ipotesi (vedi Sezione 4.8) esiste una corrispondenza biunivoca tra derivazioni sinistre ed alberi sintattici.

[6] La notazione LL è dovuta al fatto che la stringa in input è analizzata da sinistra verso destra (Left-to-right) e che viene ricostruita la derivazione sinistra (Leftmost derivation).

Concludiamo precisando che l'unione dei linguaggi LL(k), per tutti i $k > 0$ è un sottoinsieme proprio della classe dei linguaggi non contestuali deterministici; in altre parole, esistono linguaggi non contestuali deterministici che non sono LL(k) per alcun k. A titolo di esempio, possiamo considerare la seguente grammatica:

$$S \longrightarrow lSar \mid lSbr \mid lar \mid lbr$$

che produce stringhe del tipo $l^n((a+b)r)^n$, in cui cioè compare una sequenza di $n \geq 0$ caratteri l, seguita da una sequenza di n coppie di caratteri, in cui il primo è a o b e il secondo è r. Allo scopo di determinare quale sia la prima produzione da applicare per generare una data stringa, è necessario conoscere il carattere (a o b) di posizione $3n-1$. Dunque, per ogni k fissato, nessun parser LL(k) può riconoscere stringhe di lunghezza maggiore di $k+1$.

Esercizio 4.15 Verificare che la grammatica delle espressioni aritmetiche definita nell'Esempio 4.14 non è LL(k) per nessun k.

Esempio 4.11 Gli analizzatori sintattici più potenti e diffusi in pratica sono basati su tecniche di analisi ascendente (*bottom-up*). Tali tecniche mirano a ricostruire l'albero sintattico partendo dalle foglie e procedendo verso l'alto. In altre parole, se la stringa appartiene al linguaggio, essi ricostruiscono la sua derivazione destra[7], determinandone le relative produzioni in ordine inverso.

Corrispondentemente a quanto avviene per l'analisi discendente, anche in questo caso si possono introdurre classi di linguaggi analizzabili efficientemente con tecniche di analisi ascendente: i linguaggi LR(k).[8] Questi linguaggi ammettono grammatiche LR(k), le quali consentono di effettuare l'analisi ascendente esaminando k simboli dell'input alla volta.

In particolare, un parser LR(k) utilizza una pila in cui può inserire simboli terminali e non terminali. Ad ogni passo, assumendo che abbia già ricostruito le ultime r produzioni nella derivazione destra, il parser osserva i prossimi k caratteri in input, oltre al contenuto della pila. Utilizzando tale informazione, esso determina se, per un qualche $h > 0$, gli h simboli affioranti nella pila costituiscono la parte destra della $r+1$-esima produzione nella derivazione destra scandita al contrario. In tal caso, se $X \longrightarrow \gamma$ è tale produzione, il parser sostituisce sulla pila a tali simboli il non terminale X. In caso contrario, il primo dei k caratteri osservati viene inserito in pila.

Si osservi che la grammatica introdotta alla fine dell'Esempio 4.10 risulta essere LR(0).

Esercizio 4.16 Verificare l'ultima affermazione, vale a dire che la grammatica alla fine dell'Esempio 4.10 è LR(0).

[7] Nella derivazione destra ad ogni passo si applica una produzione al non terminale più a destra. Anche in questo caso, sotto opportune ipotesi (vedi Sezione 4.8), esiste una corrispondenza biunivoca tra derivazioni destre ed alberi sintattici.

[8] Left-to-right, Rightmost derivation. Si può dimostrare che la classe dei linguaggi LR(1) e quella dei linguaggi di tipo 2 deterministici coincidono.

4.8 Grammatiche e linguaggi ambigui

Qualunque sia la tecnica algoritmica utilizzata per l'analisi sintattica è importante che la stringa da analizzare sia associabile ad un unico albero sintattico. Ciò, sebbene alquanto ragionevole, non è sempre assicurato.

Definizione 4.13 *Una grammatica $\mathcal{G}$ si dice* ambigua *se esiste una stringa x in $L(\mathcal{G})$ derivabile con due diversi alberi sintattici.*

Poiché l'albero sintattico di una stringa corrisponde in qualche modo al significato della stringa stessa, l'univocità di questo albero è importante per comprendere senza ambiguità tale significato. Nel caso dei linguaggi di programmazione, il problema dell'ambiguità è evidentemente cruciale: per consentire la corretta interpretazione, e traduzione, di un programma scritto in un linguaggio ad alto livello, è necessario che la grammatica che definisce il linguaggio di programmazione sia non ambigua.

Esempio 4.12 Si consideri la grammatica

$$E \longrightarrow E + E \longrightarrow E - E \mid E * E \mid E/E \mid (E) \mid a.$$

Essa genera tutte le espressioni aritmetiche sulla variabile a, ma come si vede facilmente la stessa espressione può essere derivata con alberi di derivazione diversi. Ad esempio la stringa $a + a * a$ può venire derivata mediante i due alberi di Figura 4.8, corrispondenti a due diverse interpretazioni dell'espressione.

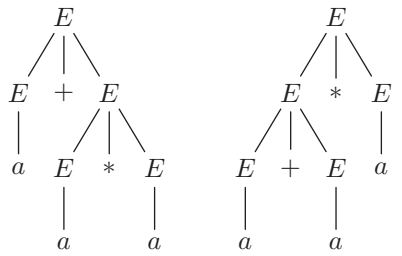


FIGURA 4.8 Alberi sintattici diversi per la stringa $a + a * a$.

L'eliminazione dell'ambiguità è uno dei principali problemi nello studio dei linguaggi di programmazione e dei metodi di analisi sintattica. In questa sede dedichiamo solo qualche breve considerazione accennando ai due metodi fondamentali per l'eliminazione dell'ambiguità: l'uso di *parentesi* e l'uso di *precedenza* tra gli operatori.

Le parentesi sono uno strumento molto utile per rendere esplicita la struttura dell'albero di derivazione di una stringa.

Esempio 4.13 Supponiamo di modificare la grammatica precedente nel seguente modo:

$$E \longrightarrow (E + E) \longrightarrow (E - E) \mid (E * E) \mid (E/E) \mid (E) \mid a.$$

I due diversi alberi di derivazione che davano origine alla stessa stringa, danno ora origine alle due stringhe

$$(a + (a * a))$$
$$((a + a) * a).$$

Come è noto dall'uso dell'aritmetica elementare, in luogo delle parentesi si può assumere una particolare regola di precedenza, che induce una lettura univoca delle stringhe. Ad esempio, la precedenza del "$*$" sul "$+$" fa sì che la stringa $a + a * a$ sia univocamente interpretata come $(a + (a * a))$. In alcuni casi è possibile modificare la grammatica in modo tale che essa rappresenti al suo interno la regola di precedenza introdotta.

Esempio 4.14 La seguente grammatica fornisce una definizione non ambigua delle espressioni aritmetiche utilizzabili in un programma. La grammatica rappresenta nella sua struttura le relazioni di precedenza definite tra gli operatori (nell'ordine non decrescente +,-,*,/) e in tal modo consente di utilizzare le parentesi soltanto quando esse strettamente necessario. Per semplicità assumiamo ancora di avere un unico simbolo di variabile: il simbolo a.

$$\begin{aligned} E &\longrightarrow E + T \mid E - T \mid T \\ T &\longrightarrow T * F \mid T/F \mid F \\ F &\longrightarrow (E) \mid a \end{aligned}$$

Il primo problema che nasce in relazione all'ambiguità, data una grammatica $\mathcal{G}$ non contestuale, è stabilire se $\mathcal{G}$ è ambigua. Purtroppo tale problema non ammette soluzione poiché, come vedremo nella Sezione 4.8.1, il problema dell'ambiguità di una grammatica non contestuale è indecidibile. In altre parole non esiste un algoritmo che, data una grammatica $\mathcal{G}$ non contestuale, stabilisca se $\mathcal{G}$ è ambigua o meno.

Un secondo problema di interesse in relazione all'ambiguità è, data una grammatica ambigua $\mathcal{G}_{\mathrm{A}}$, stabilire se esiste una grammatica non ambigua equivalente a $\mathcal{G}_{\mathrm{A}}$.

Tale problema assume particolare rilevanza in quanto esistono linguaggi per cui tutte le grammatiche sono ambigue.

Definizione 4.14 *Un linguaggio di tipo 2 si dice* inerentemente ambiguo *se tutte le grammatiche che lo generano sono ambigue.*

Anche il problema dell'inerente ambiguità di un linguaggio è indecidibile. Possiamo tuttavia mostrare un esempio di linguaggio inerentemente ambiguo per il quale questa sgradevole proprietà non solo può essere dimostrata, ma risulta sufficientemente intuitiva da poter essere compresa anche senza dimostrazione formale.

Esempio 4.15 Il linguaggio

$$\{a^n b^n c^m \mid n, m \geq 1\} \cup \{a^m b^n c^n \mid m, n \geq 1\}$$

è inerentemente ambiguo. Infatti è facile comprendere che, qualunque grammatica venga utilizzata, le stringhe del tipo $a^n b^n c^n$ che fanno parte del linguaggio possono sempre essere generate in due modi diversi.

4.8.1 Indecidibilità del problema dell'ambiguità di una grammatica

Al fine di mostrare la indecidibilità del problema in esame, consideriamo un nuovo problema, anch'esso indecidibile, noto come problema delle corrispondenze di Post (PCP), dal nome del logico matematico che lo ha definito. Il problema consiste nello stabilire se, dato un alfabeto Σ e date due sequenze di k parole $X = x_1, \ldots, x_k$ e $Y = y_1, \ldots, y_k$ costruite su Σ assegnato, esiste una sequenza di $m \geq 1$ interi $i_1, i_2, \ldots, i_m$ tale che risulti

$$x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}.$$

La sequenza $i_1, i_2, \ldots, i_m$ viene detta soluzione dell'istanza (X, Y) del problema.

Esempio 4.16 Consideriamo le due sequenze $1, 10111, 10$ e $111, 10, 0$ costruite sull'alfabeto $\{0, 1\}$. È immediato verificare che la sequenza di interi $2, 1, 1, 3$ costituisce una soluzione alla istanza di PCP considerata.

Esercizio 4.17 Date le sequenze $10, 011, 101$ e $101, 11, 011$ costruite sull'alfabeto $\{0, 1\}$, si verifichi che il PCP definito da tali sequenze non ammette soluzione.

L'indecidibilità del PCP può essere dimostrata per assurdo, provando che nell'ipotesi che esista un algoritmo risolutivo per PCP questo potrebbe essere sfruttato per risolvere problemi già noti come indecidibili su dispositivi di calcolo più generali degli automi a pila (come ad esempio per risolvere il problema della terminazione per le Macchine di Turing, studiate nel Capitolo 5).

Teorema 4.19 *Il problema delle corrispondenze di Post è indecidibile, ovvero, non esiste un algoritmo che, date due sequenze di k parole $x_1, \ldots, x_k$ e $y_1, \ldots, y_k$ costruite su un alfabeto Σ fissato, stabilisce se esiste una sequenza di $m \geq 1$ interi $i_1, i_2, \ldots, i_m$ tale che risulti*

$$x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}.$$

Dimostrazione. Omessa. □

Una tecnica simile consente di provare l'indecidibilità della ambiguità di una grammatica di tipo 2. Infatti, si sfrutta l'idea di mostrare che, se per assurdo potessimo decidere l'ambiguità di una grammatica di tipo 2, allora potremmo facilmente ottenere un algoritmo che risolve PCP, il che è assurdo.

Teorema 4.20 *Data una grammatica $\mathcal{G}$ di tipo 2, il problema di stabilire se $\mathcal{G}$ sia ambigua o meno è indecidibile.*

Dimostrazione. Supponiamo per assurdo che esista un algoritmo che decide il problema e proviamo come potremmo sfruttare tale algoritmo per risolvere il PCP. Sia allora $A = x_1, \ldots, x_k$ e $B = y_1, \ldots, y_k$ una istanza (generica) di PCP su un alfabeto Σ.

Dato l'alfabeto $\Sigma \cup \{a_1, a_2, \ldots, a_k\}$, con $a_i \notin \Sigma$ $i = 1, \ldots, k$, consideriamo il linguaggio $L' = L_A \cup L_B$ definito su Σ, in cui:

- $L_A = \{x_{i_1} x_{i_2} \cdots x_{i_m} a_{i_m} a_{i_{m-1}} \cdots a_{i_1} \mid m \geq 1\}$
- $L_B = \{y_{i_1} y_{i_2} \cdots y_{i_m} a_{i_m} a_{i_{m-1}} \cdots a_{i_1} \mid m \geq 1\}$.

Consideriamo ora la grammatica di tipo 2

$$\mathcal{G}' = \langle \{S, S_A, S_B\}, \Sigma \cup \{a_1, \ldots, a_k\}, P, S \rangle,$$

con produzioni P, per $i = 1, \ldots, k$:

$$\begin{array}{rcl} S & \longrightarrow & S_A \mid S_B \\ S_A & \longrightarrow & x_i S_A a_i \mid x_i a_i \\ S_B & \longrightarrow & y_i S_B a_i \mid y_i a_i. \end{array}$$

È immediato verificare che tale grammatica genera il linguaggio L' precedentemente definito.

Dimostriamo ora che l'istanza (A, B) di PCP ha soluzione se e solo se $\mathcal{G}'$ è ambigua.

Sia $i_1, \ldots, i_m$ una soluzione di PCP, per un certo $m \geq 1$. In tale ipotesi risulta ovviamente $x_{i_1} \cdots x_{i_m} a_{i_m} \cdots a_{i_1} = y_{i_1} \cdots x_{i_m} a_{i_m} \cdots a_{i_1}$. Tale stringa, appartenente a L', ammette due differenti alberi sintattici: il primo corrisponde alla derivazione

$$S \Longrightarrow S_A \Longrightarrow x_{i_1} S_A a_{i_1} \Longrightarrow x_{i_1} x_{i_2} S_A a_{i_2} a_{i_1} \overset{*}{\Longrightarrow} x_{i_1} \cdots x_{i_m} a_{i_m} \cdots a_{i_1},$$

mentre il secondo è associato alla derivazione

$$S \Longrightarrow S_B \Longrightarrow y_{i_1} S_B a_{i_1} \overset{*}{\Longrightarrow} y_{i_1} \cdots y_{i_m} a_{i_m} \cdots a_{i_1} = x_{i_1} \cdots x_{i_m} a_{i_m} \cdots a_{i_1}.$$

$\mathcal{G}'$ risulta dunque ambigua (vedi Definizione 4.13).

Per mostrare l'implicazione opposta, assumiamo che $\mathcal{G}'$ sia ambigua, e sia $z = w a_{i_m} \cdots a_{i_1}$, $m \geq 1$, una stringa di L' che ammette due distinti alberi sintattici: vogliamo ora provare che $i_1, \ldots, i_m$ è una soluzione dell'istanza (A, B) di PCP.

A causa della particolare struttura delle produzioni di $\mathcal{G}'$, possiamo asserire che i simboli a_j "dettano" le produzioni, ovvero per generare il simbolo a_j è necessario usare una fra le produzioni $S_A \to x_j S_A a_j$ e $S_B \to y_j S_B a_j$ (o

$S_A \to x_j a_j$ e $S_B \to y_j a_j$, qualora a_j sia il primo dei simboli a_i presenti nella parola generata). Ne segue che affinché z ammetta due differenti derivazioni è necessario che queste usino rispettivamente le produzioni su S_A e su S_B. L'ulteriore conseguenza è che la sottostringa w che costituisce prefisso di z può essere scritta sia come $x_{i_1} \cdots x_{i_m}$ sia come $y_{i_1} \cdots y_{i_m}$. La sequenza $i_1, \ldots, i_m$ è dunque soluzione dell'istanza (A, B) di PCP. □

La tecnica utilizzata nella prova del teorema precedente è potente e generale, e viene spesso utilizzata per dimostrare risultati di indecidibilità. Essa consiste nel "ridurre" un problema, già noto come indecidibile, al problema in esame, ovvero nel mostrare come un ipotetico algoritmo per la soluzione del problema in esame consentirebbe per assurdo di risolvere un problema indecidibile. Utilizzando ad esempio la tecnica di riduzione, in una maniera molto simile a quanto effettuato nella prova del teorema precedente, si può mostrare indecidibile il problema di stabilire se, date due grammatiche di tipo 2, l'intersezione dei linguaggi da esse generate sia vuota o meno.

Esercizio 4.18 Date due grammatiche di tipo 2 $\mathcal{G}_A$ e $\mathcal{G}_B$, mostrare che il problema di stabilire se $L(\mathcal{G}_A) \cap L(\mathcal{G}_A)$ sia vuoto o meno è indecidibile.

Un tale risultato negativo ci consente di provare in maniera alternativa che la classe dei linguaggi di tipo 2 non è chiusa rispetto all'intersezione (vedi Corollario 4.9. Infatti, se l'intersezione di due linguaggi di tipo 2 fosse di tipo 2, visto che è possibile decidere se un linguaggio di tipo 2 è vuoto o meno (vedi Teorema 4.13), potremmo allora decidere anche se l'intersezione di due linguaggi di tipo 2 sia vuota o meno, il che è invece indecidibile.

4.9 Algoritmi di riconoscimento di linguaggi non contestuali

Come visto nella Sezione 4.5, il modello di calcolo degli automi a pila deterministici non è sufficientemente potente da consentire il riconoscimento dell'intera classe dei linguaggi non contestuali e che, a tal fine, si rende necessario utilizzare modelli di calcolo non deterministici. Ciò, apparentemente, rende il problema del riconoscimento di tali linguaggi impossibile dal punto di vista pratico, in un contesto in cui possiamo fare riferimento a soli modelli deterministici.

In questa sezione mostriamo come una diversa estensione degli automi a pila deterministici permetta di ottenere un modello di calcolo ancora deterministico in grado di riconoscere i linguaggi non contestuali. L'estensione consiste nel rilassare i vincoli, imposti dalla politica LIFO, sulle possibilità di accesso ai dati memorizzati, permettendo ad ogni istante l'accesso diretto a uno qualunque di essi; in altre parole, si sostituisce alla pila un array di dimensione potenzialmente illimitata.

In questa sezione introduciamo un algoritmo deterministico per il riconoscimento dei linguaggi di tipo 2 che, utilizzando un array bidimensionale, sfrutta le caratteristiche del nuovo modello.

Tale algoritmo, introdotto da Cocke, Younger e Kasami e denominato CYK, data una grammatica non contestuale in forma normale di Chomsky $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ ed una stringa $x \in V_T^*$ (con $|\,x\,| = n$), determina in tempo $O(n^3)$ se x è derivabile in $\mathcal{G}$ o meno.

L'algoritmo, che applica il paradigma della Programmazione Dinamica, si basa sulle seguenti osservazioni:

1. Data $x = a_1, a_2, \ldots, a_n \in V_T^+$, sia $x_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq n - i + 1)$ la sua sottostringa $a_i, a_{i+1}, \ldots, a_{i+j-1}$ di lunghezza j che inizia dall'i-esimo carattere di x. Sia inoltre $A_{i,j} \subseteq V_N$ l'insieme dei simboli non terminali in $\mathcal{G}$ da cui è possibile derivare $x_{i,j}$:

$$A_{i,j} = \{A \in V_N \mid A \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x_{i,j}\}$$

2. Per ogni i $(1 \leq i \leq n)$ è immediato determinare $A_{i,1} = \{A \in V_N \mid A \longrightarrow a_i\}$ per ispezione di P.

3. Per ogni coppia i, j $(1 \leq i \leq n, 1 < j \leq n - i + 1)$ un non terminale A appartiene ad $A_{i,j}$ se e solo se esiste k $(1 \leq k \leq j - 1)$ tale che:

   (a) esiste $A' \in A_{i,k}$;

   (b) esiste $A'' \in A_{i+k,j-k}$;

   (c) esiste una produzione $A \longrightarrow A'A'' \in P$.

   Infatti, in tal caso la stringa $x_{i,j}$ è derivabile da A applicando dapprima la produzione $A \longrightarrow A'A''$ e derivando quindi separatamente i suoi primi k caratteri (la sottostringa $x_{i,k}$) da A' e gli altri $j - k$ (la sottostringa $x_{i+k,j-k}$) da A'' (Figura 4.9).

Al fine di costruire l'insieme $A_{i,j}$ $(1 \leq i \leq n, 1 < j \leq n - i)$ è necessario quindi far variare il valore k tra 1 e $j-1$ esaminando via via gli insiemi $A_{i,k}$ e $A_{i+k,j-k}$: è chiaro che, per poter far ciò, dovremo assumere che siano disponibili tutti gli insiemi $A_{r,k}$, con $1 \leq k \leq j - 1$ e $1 \leq r \leq i + j - k$.

Infine, si noti che x è derivabile in $\mathcal{G}$ se e solo se $S \in A_{1,n}$.

Passiamo ora ad esaminare in maggiore dettaglio la struttura dell'algoritmo CYK: l'algoritmo opera riempiendo un array T di dimensioni $n \times n$, in cui nella locazione $T[i, j]$ viene rappresentato l'insieme $A_{i,j}$, e verificando, alla fine di tale processo, se S compare nella locazione $T[1, n]$. È utile notare che la matrice T è triangolare, in quanto non risultano definiti gli insiemi $A_{i,j}$ per $j > n-i+1$, e che l'algoritmo costruisce la matrice stessa per colonne. È facile rendersi conto che l'Algoritmo 4.5, fissata una grammatica $\mathcal{G} = \langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{P}, \mathcal{S}\rangle$,

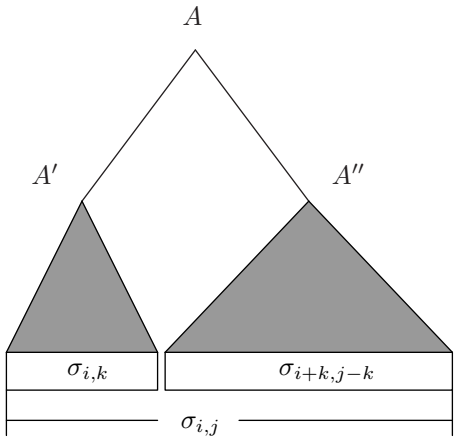


FIGURA 4.9 Derivazione di $x_{i,j}$.

esegue, per riconoscere una stringa x, un numero di passi pari a $O(\mid x \mid^3)$. Se invece $\mathcal{G}$ è considerata come parte integrante dell'input, allora il numero di passi risulta pari a $O(\mid x \mid^3 \cdot \mid V_N \mid^2 \cdot \mid P \mid)$.

Esempio 4.17 Consideriamo il linguaggio L delle stringhe palindrome di lunghezza pari sull'alfabeto $\{0, 1\}$. Una grammatica in forma normale di Chomsky per tale linguaggio ha le seguenti produzioni:

$$\begin{aligned} S &\longrightarrow Z'Z \mid U'U \mid ZZ \mid UU \\ Z' &\longrightarrow ZS \\ U' &\longrightarrow US \\ Z &\longrightarrow 0 \\ U &\longrightarrow 1 \end{aligned}$$

Vogliamo verificare, applicando l'algoritmo CYK, se la stringa 0110 appartiene ad L.

L'algoritmo costruisce una matrice T 4×4, colonna per colonna, da sinistra verso destra. La prima colonna, corrispondente al caso $j = 1$, viene riempita mediante l'ispezione delle produzioni che generano un terminale, risultando nella seguente situazione:

Z			
U			
U			
Z			

Ciò deriva dal fatto che i simboli 0 che compaiono come primo e quarto carattere sono derivati a partire soltanto da Z, mentre i restanti simboli 1 sono derivati soltanto a partire da U.

```
input Grammatica G, stringa x = a_1, ..., a_n ∈ V_T^*;
output Boolean;
begin
  for i := 1 to n do
    begin
      T[i,1] := ∅;
      for each A ⟶ a ∈ P do
        if a = a_i then T[i,1] = T[i,1] ∪ {A}
    end;
  for j := 2 to n − 1 do
    for i := 1 to n − j + 1 do
      begin
        T[i,j] := ∅;
        for k := 1 to j − 1 do
          for each B ∈ T[i,k] do
            for each C ∈ T[i + k, j − k] do
              for each D ∈ V_N do
                if D ⟶ BC ∈ P then T[i,j] := T[i,j] ∪ {D}
      end
  if S ∈ T[1,n] then
    return VERO
    else return FALSO
end.
```

Algoritmo 4.5: Algoritmo CYK di riconoscimento di linguaggi CF

Il riempimento della seconda colonna, corrispondente al caso $j = 2$, risulta in:

Z	$\emptyset$		
U	S		
U	$\emptyset$		
Z			

Tale situazione rappresenta il fatto che non esistono non terminali a partire dai quali è possibile derivare le sottostringhe 01 (corrispondente all'elemento $T[1,2]$) e 10 (corrispondente all'elemento $T[3,2]$) e che la sottostringa 11 (corrispondente all'elemento $T[2,2]$) può essere derivata a partire da S. In quest'ultimo caso, l'algoritmo considera la decomposizione della stringa 11 associata all'elemento $T[2,2]$ nelle due sottostringhe corrispondenti agli elementi $T[2,1]$ e $T[3,1]$, ciascuna delle quali è derivabile dal simbolo U, e, osservando che esiste la produzione $S \longrightarrow UU$, determina che S va inserito in $T[2,2]$.

Il riempimento della terza colonna, corrispondente al caso $j = 3$, risulta in:

Z	$\emptyset$	Z'	
U	S	$\emptyset$	
U	$\emptyset$		
Z			

Tale situazione rappresenta il fatto che non esistono non terminali a partire dai quali è possibile derivare la sottostringa 110 (corrispondente al-

l'elemento $T[2,3]$) e che la sottostringa 011 (corrispondente all'elemento $T[1,3]$) può essere derivata a partire da Z. L'algoritmo, in questo caso, determina che, in corrispondenza alla decomposizione della stringa 011 associata all'elemento $T[1,3]$ nelle due sottostringhe corrispondenti agli elementi $T[1,1]$ e $T[2,2]$, la prima sottostringa è derivabile dal simbolo Z e la seconda è derivabile dal simbolo S, ed osserva inoltre che esiste la produzione $Z' \longrightarrow ZS$, determinando quindi che che Z' va inserito in $T[1,3]$.

Infine, il riempimento della quarta colonna, corrispondente al caso $j = 4$, risulta in:

Z	$\emptyset$	Z'	S
U	S	$\emptyset$	
U	$\emptyset$		
Z			

Tale situazione rappresenta il fatto che l'intera stringa, corrispondente all'elemento $T[1,4]$, può essere derivata a partire da S. L'algoritmo verifica ciò osservando che, data la decomposizione della stringa nelle due sottostringhe corrispondenti agli elementi $T[1,3]$ e $T[4,1]$, la prima sottostringa è derivabile dal simbolo Z' e la seconda è derivabile dal simbolo Z, e, osservando inoltre che esiste la produzione $S \longrightarrow Z'Z$, determina quindi che che S va inserito in $T[1,4]$.

A questo punto, la presenza, una volta riempita la matrice, del simbolo S in $T[1,4]$ consente di determinare che in effetti la stringa è derivabile a partire da S ed appartiene quindi al linguaggio.

Esercizio 4.19 Mostrare, applicando l'algoritmo CYK, che la stringa $aaacbbbdedaba$ è derivabile nella grammatica non contestuale:

$$\begin{aligned} S &\longrightarrow AS \mid BS \mid A \mid B \\ A &\longrightarrow aAb \mid c \\ B &\longrightarrow dBd \mid e \end{aligned}$$

Capitolo 5

Macchine di Turing e calcolabilità secondo Turing

Le *Macchine di Turing* (MT) sono il modello di calcolo di riferimento fondamentale sia nell'ambito della teoria della calcolabilità sia in quello della teoria della complessità computazionale.

Quando il logico inglese A. M. Turing ha introdotto questo modello di calcolo (vedi Appendice A) egli si poneva l'obiettivo di formalizzare il concetto di calcolo allo scopo di stabilire l'esistenza di metodi algoritmici per il riconoscimento dei teoremi nell'ambito dell'Aritmetica. Nonostante il fatto che ciò peraltro accadesse in un periodo in cui i primi elaboratori elettronici non erano ancora stati progettati, il modello definito da Turing ha assunto un ruolo molto importante per lo studio dei fondamenti teorici dell'informatica perché in tale modello si associa una elevata semplicità di struttura e di funzionamento ad un potere computazionale che è, fino ad oggi, ritenuto il massimo potere computazionale realizzabile da un dispositivo di calcolo automatico.

Così come gli automi a stati finiti e gli automi a pila, di cui rappresenta un'estensione in termini di capacità operative, la macchina di Turing è definita come un dispositivo operante su stringhe contenute su una memoria esterna (nastro) mediante passi elementari, definiti da una opportuna funzione di transizione.

In questa sezione presenteremo dapprima la versione più semplice della macchina di Turing e successivamente versioni più complesse utili per approfondire aspetti particolari del concetto di calcolo. Inoltre definiremo le nozioni di calcolabilità e di decidibilità indotte dal modello della macchina di Turing.

5.1 Macchine di Turing a nastro singolo

Nella sua versione più tradizionale una macchina di Turing si presenta come un dispositivo che accede ad un *nastro* potenzialmente illimitato diviso in *celle* contenenti ciascuna un simbolo appartenente ad un dato *alfabeto* Γ dato, ampliato con il *carattere speciale* $\flat$ (blank) che rappresenta la situazione di cella non contenente caratteri. La macchina di Turing opera su tale nastro tramite una *testina*, la quale può scorrere su di esso in entrambe le direzioni, come illustrato in Figura 5.1. Su ogni cella la testina può leggere o scrivere caratteri appartenenti all'alfabeto Γ oppure il simbolo $\flat$.

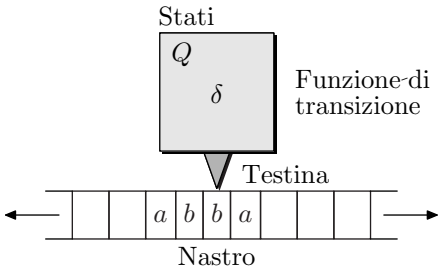


FIGURA 5.1 Macchina di Turing.

Ad ogni istante la macchina si trova in uno *stato* appartenente ad un insieme finito Q ed il meccanismo che fa evolvere la computazione della macchina è una *funzione di transizione* δ la quale, a partire da uno stato e da un carattere osservato sulla cella del nastro su cui è attualmente posizionata la testina, porta la macchina in un altro stato, determinando inoltre la scrittura di un carattere su tale cella ed eventualmente lo spostamento della testina stessa.

Analogamente a quanto fatto per i vari tipi di automi presentati nei capitoli precedenti, procediamo ora a definire in modo formale la versione deterministica del modello di calcolo che stiamo considerando.

Definizione 5.1 *Una macchina di Turing deterministica (MTD) è una sestupla* $\mathcal{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$*, dove* Γ *è l'*alfabeto dei simboli di nastro*,* $\flat \notin \Gamma$ *è un carattere speciale denominato* blank*,* Q *è un insieme finito e non vuoto di* stati*,* $q_0 \in Q$ *è lo* stato iniziale*,* $F \subseteq Q$ *è l'insieme degli* stati finali *e* δ *è la* funzione (parziale) di transizione*, definita come*

$$\delta : (Q - F) \times (\Gamma \cup \{\flat\}) \mapsto Q \times (\Gamma \cup \{\flat\}) \times \{d, s, i\}$$

in cui d, s e i indicano, rispettivamente, lo spostamento a destra, lo spostamento a sinistra e l'assenza di spostamento della testina.

Ad esempio, la Figura 5.2 illustra la transizione definita da $\delta(q_3, b) = (q_4, a, d)$.

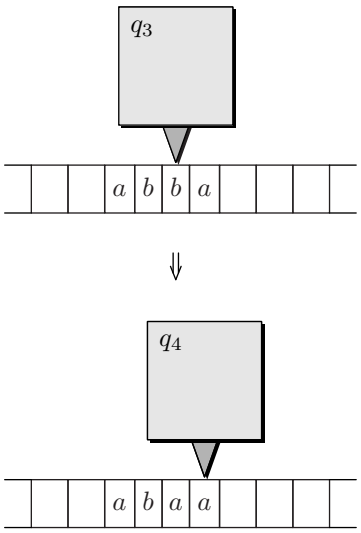


FIGURA 5.2 Transizione determinata da $\delta(q_3, b) = (q_4, a, d)$.

Nel seguito, dato un alfabeto Γ tale che $\flat \notin \Gamma$, si indicherà con $\bar{\Gamma}$ l'insieme $\Gamma \cup \{\flat\}$. Le macchine di Turing deterministiche possono venire utilizzate sia come strumenti per il calcolo di funzioni, sia per riconoscere o accettare stringhe su un *alfabeto di input* $\Sigma \subseteq \Gamma$, cioè per stabilire se esse appartengano ad un certo linguaggio oppure no. Si noti che, in realtà, anche questo secondo caso può essere visto come il calcolo di una particolare funzione, la *funzione caratteristica* del linguaggio, che, definita da Σ^* a $\{0, 1\}$, assume valore 1 per ogni stringa del linguaggio e 0 altrimenti.

Le macchine usate per accettare stringhe vengono dette di tipo *riconoscitore*, mentre quelle usate per calcolare funzioni vengono dette di tipo *trasduttore*. In entrambi i casi, all'inizio del calcolo, solo una porzione finita del nastro contiene simboli diversi da blank che costituiscono l'input del calcolo stesso. Di ciò si parlerà più estesamente nelle Sezioni 5.1.2 e 5.2.

Introduciamo ora i concetti di configurazione e di transizione fra configurazioni per le macchine di Turing (sia deterministiche che non deterministiche).

5.1.1 Configurazioni e transizioni delle Macchine di Turing

La *configurazione istantanea* (o più semplicemente *configurazione*) di una macchina di Turing è l'insieme delle informazioni costituito dal contenuto del nastro, dalla posizione della testina e dallo stato corrente.

Una possibile rappresentazione di una configurazione è data da una stringa di lunghezza finita contenente tutti i caratteri presenti sul nastro, oltre a

un simbolo speciale corrispondente allo stato interno, posto immediatamente prima del carattere individuato dalla posizione della testina.

Infatti, anche se la macchina di Turing è definita come un dispositivo operante su di un nastro infinito, come si è detto, in ogni momento solo una porzione finita di tale nastro contiene simboli diversi da $\flat$. Per tale ragione, si conviene di rappresentare nella configurazione la più piccola porzione (finita) di nastro contenente tutti i simboli non $\flat$ — includendo obbligatoriamente la cella su cui è posizionata la testina, — come illustrato in Figura 5.3.

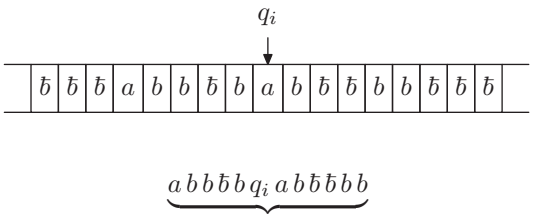


FIGURA 5.3 Esempio di configurazione istantanea.

Ciò premesso, definiamo il concetto di configurazione nel modo seguente.

Definizione 5.2 *Si definisce* configurazione istantanea *o* configurazione *di una macchina di Turing con alfabeto di nastro* Γ *ed insieme degli stati* Q*, una stringa* $c = xqy$*, con:*

1. $x \in \Gamma\bar{\Gamma}^* \cup \{\varepsilon\}$;
2. $q \in Q$;
3. $y \in \bar{\Gamma}^*\Gamma \cup \{\flat\}$.

L'interpretazione data ad una stringa xqy è che xy rappresenti il contenuto della sezione non vuota del nastro, che lo stato attuale sia q e che la testina sia posizionata sul primo carattere di y. Nel caso in cui $x = \varepsilon$ abbiamo che a sinistra della testina compaiono solo simboli $\flat$, mentre se $y = \flat$ sulla cella attuale e a destra della testina compaiono soltanto simboli $\flat$.

Nel seguito, per brevità, indicheremo con $\mathcal{L}_\Gamma$ il linguaggio $\Gamma\bar{\Gamma}^* \cup \{\varepsilon\}$ delle stringhe che possono comparire a sinistra del simbolo di stato in una configurazione, e con $\mathcal{R}_\Gamma$ il linguaggio $\bar{\Gamma}^*\Gamma \cup \{\flat\}$ delle stringhe che possono comparire alla sua destra.

Esercizio 5.1 Dati $\Gamma = \{a, b\}$ e $Q = \{q_0, q_1, q_2\}$, costruire un automa a stati finiti che riconosca tutte le stringhe che rappresentano configurazioni di una macchina di Turing con alfabeto Γ ed insieme degli stati Q.

Un particolare tipo di configurazione è rappresentato dalla *configurazione iniziale*. Indichiamo con tale termine una configurazione che, data una qualunque stringa $x \in \Gamma^*$, rappresenti stato e posizione della testina all'inizio di una computazione su input x. Per quanto riguarda lo stato, ricordiamo che per definizione esso dovrà essere q_0, mentre, prima di definire la posizione della testina, è opportuno stabilire una convenzione sulla maniera di specificare l'input $x \in \Gamma^*$ su cui la macchina deve operare. La convenzione universalmente adottata prevede che all'inizio della computazione la macchina abbia il nastro contenente tale input su una sequenza di $\mid x \mid$ celle contigue, che tutte le altre celle siano poste a $\b$ e che la testina sia inizialmente posizionata sul primo carattere di x. Queste considerazioni ci portano alla seguente definizione.

Definizione 5.3 *Una configurazione $c = xqy$ si dice* iniziale *se $x = \varepsilon$, $q = q_0$, $y \in \Gamma^+ \cup \{\b\}$.*

Altro tipo di configurazione di interesse è quella *finale*.

Definizione 5.4 *Una configurazione $c = xqy$, con $x \in \mathcal{L}_\Gamma$, $y \in \mathcal{R}_\Gamma$ si dice* finale *se $q \in F$.*

Quindi, una macchina di Turing si trova in una configurazione finale se il suo stato attuale è uno stato finale, indipendentemente dal contenuto del nastro e dalla posizione della testina.[1] Chiaramente, una configurazione descrive in modo statico la situazione in cui una macchina di Turing si trova in un certo istante. Per caratterizzare il comportamento "dinamico" di una macchina di Turing, che determina il suo passaggio da configurazione a configurazione, dobbiamo fare riferimento alla relativa funzione di transizione.

Osserviamo che, come nel caso degli automi visti nei capitoli precedenti, una funzione di transizione può essere rappresentata mediante *matrici di transizione* e *grafi di transizione*.

Nel caso della macchina di Turing, le colonne della matrice di transizione corrispondono ai caratteri osservabili sotto la testina (elementi di $\bar{\Gamma}$) e le righe ai possibili stati interni della macchina (elementi di Q), mentre gli elementi della matrice (nel caso di macchina di Turing deterministica) sono triple che rappresentano il nuovo stato, il carattere che viene scritto e lo spostamento della testina.

In Figura 5.1 è riportata una possibile matrice di transizione per una macchina di Turing deterministica con alfabeto di nastro $\Gamma = \{0, 1, *, \$\}$ ed insieme degli stati $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_F\}$, dove q_F è l'unico stato finale.

Nella rappresentazione della funzione di transizione mediante grafo i nodi sono etichettati con gli stati e gli archi con una tripla composta dal carattere letto, il carattere scritto e lo spostamento della testina. Più precisamente, nel grafo di transizione esiste un arco dal nodo q_i al nodo q_j, e tale arco è etichettato con la tripla $a, b \in \bar{\Gamma}, m \in \{s, d, i\}$ se e solo se $\delta(q_i, a) = (q_j, b, m)$.

[1] Per definizione della funzione di transizione, una macchina di Turing che si trovi in una configurazione finale non può effettuare ulteriori passi di computazione.

	0	1	*	$	$\b$
q_0	$(q_1, *, d)$	$(q_2, \$, d)$	-	-	$(q_F, \b, i)$
q_1	$(q_1, 0, d)$	$(q_1, 1, d)$	-	-	$(q_3, \b, d)$
q_2	$(q_2, 0, d)$	$(q_2, 1, d)$	-	-	$(q_4, \b, d)$
q_3	$(q_3, 0, d)$	$(q_3, 1, d)$	-	-	$(q_5, 0, s)$
q_4	$(q_4, 0, d)$	$(q_4, 1, d)$	-	-	$(q_6, 1, s)$
q_5	$(q_5, 0, s)$	$(q_5, 1, s)$	$(q_0, 0, d)$	-	$(q_5, \b, s)$
q_6	$(q_6, 0, s)$	$(q_6, 1, s)$	-	$(q_0, 1, d)$	$(q_6, \b, s)$
q_F	-	-	-	-	-

Tabella 5.1 Matrice di transizione.

In Figura 5.4 è riportato il grafo di transizione corrispondente alla matrice di Tabella 5.1.

Data una configurazione c, una applicazione della funzione di transizione δ della macchina di Turing deterministica $\mathcal{M}$ su c permette di ottenere la configurazione successiva c' secondo le modalità seguenti:

1. Se $c = xqay$, con $x \in \mathcal{L}_\Gamma$, $y \in \bar{\Gamma}^*\Gamma$, $a \in \bar{\Gamma}$, e se $\delta(q, a) = (q', a', d)$, allora $c' = xa'q'y$;

2. Se $c = xqa$, con $x \in \mathcal{L}_\Gamma$, $a \in \bar{\Gamma}$, e se $\delta(q, a) = (q', a', d)$, allora $c' = xa'q'\b$;

3. Se $c = xaqby$, con $xa \in \Gamma\bar{\Gamma}^*$, $y \in \bar{\Gamma}^*\Gamma \cup \{\varepsilon\}$, $b \in \bar{\Gamma}$, e se $\delta(q, b) = (q', b', s)$, allora $c' = xq'ab'y$;

4. Se $c = qby$, con $y \in \bar{\Gamma}^*\Gamma \cup \{\varepsilon\}$, $b \in \bar{\Gamma}$, e se $\delta(q, b) = (q', b', s)$, allora $c' = q'\b b'y$;

5. Se $c = xqay$, con $x \in \mathcal{L}_\Gamma$, $\beta \in \bar{\Gamma}^*\Gamma \cup \{\varepsilon\}$, $a \in \bar{\Gamma}$, e se $\delta(q, a) = (q', a', i)$, allora $c' = xq'a'y$.

Come già fatto per i modelli di macchina visti nei capitoli precedenti, la relazione tra c e c' (detta definerelazione di transizione) viene indicata per mezzo della notazione $c \underset{\mathcal{M}}{\vdash\!\!\!-} c'$.

È chiaro a questo punto che il comportamento di una macchina di Turing è definito dalle regole di transizione che fanno passare da una configurazione ad un'altra. Si noti che tali regole possono essere interpretate come *regole di riscrittura* del nastro.

Ad esempio la regola $\delta(q_1, a) = (q_2, b, s)$ applicata alla configurazione

$$abb\b bq_1ab\b\b bb$$

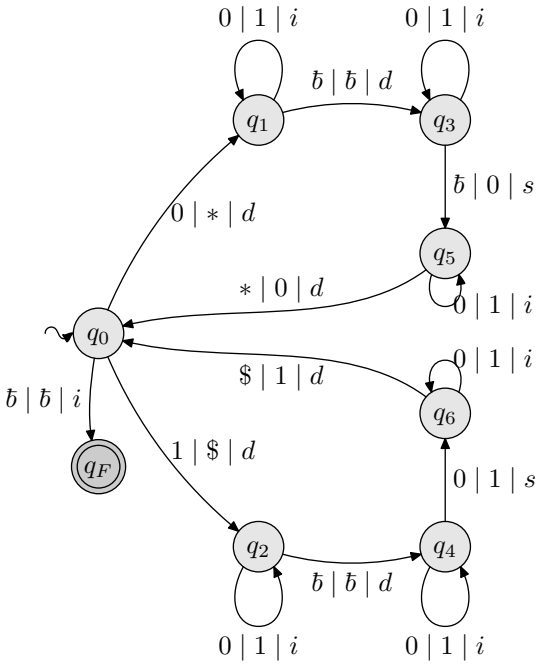


FIGURA 5.4 Grafo di transizione corrispondente alla matrice in Tabella 5.1.

porta alla configurazione

$$abb\underline{b}q_2bbb\underline{b}\underline{b}bb.$$

Esercizio 5.2 Considerata la macchina di Turing deterministica definita in Tabella 5.1 e in Figura 5.4 e assumendo la configurazione iniziale $q_0 10$:

1. indicare le prime dieci configurazioni che vengono raggiunte;
2. dire quale configurazione finale viene raggiunta;
3. descrivere informalmente il comportamento della macchina su un input generico.

5.1.2 Computazioni di Macchine di Turing

Le macchine di Turing deterministiche rappresentano uno strumento teorico potente e generale per affrontare in maniera adeguata il problema del riconoscimento di linguaggi. In tale contesto si parla delle macchine di Turing come di dispositivi *riconoscitori*. Si noti che, dato un *alfabeto di input* $\Sigma \subseteq \Gamma$, una macchina di Turing può essere vista come un dispositivo che classifica le stringhe in Σ^* in funzione del tipo di computazione indotto.

Definizione 5.5 *Data una macchina di Turing deterministica $\mathcal{M}$ con alfabeto Γ e stato iniziale q_0, e dato un alfabeto di input $\Sigma \subseteq \Gamma$, una stringa $x \in \Sigma^*$ è* accettata *(*rifiutata*) da $\mathcal{M}$ se esiste una computazione di accettazione (di rifiuto) di $\mathcal{M}$ con $c_0 = q_0 x$.*

Esercizio 5.3 Definire una macchina di Turing deterministica con alfabeto di input $\{0, 1\}$ che accetta tutte le stringhe di lunghezza dispari e rifiuta tutte le altre.

In realtà, la Definizione 5.5 non esaurisce l'insieme dei possibili esiti delle computazioni eseguite da $\mathcal{M}$. Infatti, a differenza di quanto si verifica per i modelli di automa visti nei capitoli precedenti, nel caso di una macchina di Turing può accadere che per essa non esista alcuna computazione massimale con $c_0 = q_0 x$; in altre parole può accadere che la computazione di $\mathcal{M}$ su input x non termini.

Si osservi inoltre che, mentre per ipotesi è possibile verificare in tempo finito (semplicemente osservando la computazione eseguita) se $\mathcal{M}$ accetta o rifiuta una stringa x, la verifica di questo terzo caso si presenta di soluzione meno immediata. Infatti, in tal caso l'osservazione di $\mathcal{M}$ non ci è di aiuto, in quanto, dopo una qualunque computazione non massimale di lunghezza finita, non possiamo sapere se tale computazione terminerà in un qualche istante futuro o meno.

In effetti, per risolvere tale problema sarebbe necessario avere a disposizione una diversa macchina di Turing $\mathcal{M}'$ la quale sia in grado di determinare in un tempo finito, date $\mathcal{M}$ e x, se la computazione eseguita da $\mathcal{M}$ su input x termina o meno. Come vedremo più avanti, tale problema, detto *problema della terminazione*, non è risolubile, nel senso che non esiste alcuna macchina di Turing con questa capacità.

Visto che una computazione di una macchina di Turing può terminare oppure no, appare necessario fornire alcune precisazioni che chiariscano l'ambito e la modalità con cui un linguaggio può essere riconosciuto. Per prima cosa occorre distinguere fra *riconoscimento* e *accettazione* di un linguaggio da parte di una macchina di Turing.

Definizione 5.6 *Sia $\mathcal{M} = \langle \Gamma, \bar{b}, Q, q_0, F, \delta \rangle$ una macchina di Turing deterministica. Diciamo che $\mathcal{M}$* riconosce *(*decide*) un linguaggio $L \in \Sigma^*$ (dove $\Sigma \subseteq \Gamma$) se e solo se per ogni $x \in \Sigma^*$ esiste una computazione massimale $q_0 x \overset{*}{\underset{\mathcal{M}}{\vdash}} wqz$, con $w \in \Gamma\bar{\Gamma}^* \cup \{\varepsilon\}$, $z \in \bar{\Gamma}^*\Gamma \cup \{\bar{b}\}$, e dove $q \in F$ se e solo se $x \in L$.*

La definizione precedente non esprime alcuna condizione per il caso $x \notin \Sigma^*$: è facile però osservare che potrebbe essere estesa introdotta una macchina di Turing $\mathcal{M}'$ che verifichi inizialmente se la condizione $x \in \Sigma^*$ è vera, ed operi poi nel modo seguente:

- se $x \in \Sigma^*$, si comporta come $\mathcal{M}$;
- altrimenti, se individua in x l'occorrenza di un qualche carattere in $\Gamma - \Sigma$, termina la propria computazione in un qualche stato non finale.

Si noti che data una macchina di Turing deterministica $\mathcal{M}$ non è detto che esista un linguaggio deciso da $\mathcal{M}$: infatti ciò è possibile se e solo se $\mathcal{M}$ si ferma per qualunque input x.

Definizione 5.7 *Sia $\mathcal{M} = \langle \Gamma, \bar{b}, Q, q_0, F, \delta \rangle$ una macchina di Turing deterministica. Diciamo che $\mathcal{M}$* accetta *un linguaggio $L \in \Sigma^*$ (dove $\Sigma \subseteq \Gamma$) se e solo se $L = \{x \in \Sigma^* \mid q_0 x \overset{*}{\underset{\mathcal{M}}{\vdash}} wqz\}$, con $w \in \Gamma\bar{\Gamma}^* \cup \{\varepsilon\}$, $z \in \bar{\Gamma}^*\Gamma \cup \{\bar{b}\}$, e $q \in F$.*

Esercizio 5.4

i) Definire una macchina di Turing deterministica che riconosce il linguaggio $L = \{w\tilde{w} \mid w \in \{a,b\}^+\}$.

ii) Definire una macchina di Turing deterministica che accetta il linguaggio L sopra definito e che per qualche stringa $x \in \{a,b\}^* - L$ cicla indefinitamente.

Vale la pena segnalare che le convenzioni poste nell'ambito dei problemi di riconoscimento di linguaggi non sono le uniche possibili.

Esercizio 5.5 Si consideri una definizione di accettazione o rifiuto di stringhe definita come segue: esistono due soli stati finali q_1, q_2, tutte le computazioni massimali terminano in uno stato finale ed una stringa x è accettata se $q_0 x \overset{*}{\underset{\mathcal{M}}{\vdash\!\!\!-}} wq_1z$, mentre è rifiutata se $q_0 x \overset{*}{\underset{\mathcal{M}}{\vdash\!\!\!-}} wq_2z$.

Mostrare che questa definizione è equivalente a quella data.

Esercizio 5.6 Si consideri una definizione di accettazione o rifiuto di stringhe sull'alfabeto Σ definita come segue: esiste un solo stato finale q_F, l'alfabeto di nastro contiene due simboli speciali $\mathcal{Y}, \mathcal{N} \notin \Sigma$, tutte le computazioni massimali terminano nello stato finale ed una stringa x è accettata se $q_0 x \overset{*}{\underset{\mathcal{M}}{\vdash\!\!\!-}} q_F\mathcal{Y}$, mentre è rifiutata se $q_0 x \overset{*}{\underset{\mathcal{M}}{\vdash\!\!\!-}} q_F\mathcal{N}$.

Mostrare che questa definizione è equivalente a quella data.

5.2 Calcolo di funzioni e Macchine di Turing deterministiche

Consideriamo ora le macchine di Turing deterministiche come *trasduttori*, cioè come dispositivi generali capaci di realizzare il calcolo di funzioni parziali, definite su domini qualunque.

Consideriamo innanzi tutto il caso in cui si vogliano definire trasduttori per il calcolo di funzioni su stringhe.

Definizione 5.8 *Dato un trasduttore $\mathcal{M} = \langle \Gamma, ß, Q, q_0, F, \delta \rangle$ ed una funzione $f : \Sigma^* \mapsto \Sigma^*$ ($\Sigma \subseteq \Gamma$), $\mathcal{M}$ calcola la funzione f se e solo se per ogni $x \in \Gamma^*$:*

1. *se $x \in \Sigma^*$ e $f(x) = y$ allora $q_0 x \overset{*}{\underset{\mathcal{M}}{\vdash\!\!\!-}} xßqy$, con $q \in F$;*
2. *se $x \notin \Sigma^*$ oppure se $x \in \Sigma^*$ e $f(x)$ non è definita, allora, assumendo la configurazione iniziale q_0x, o non esistono computazioni massimali o esistono computazioni massimali che non terminano in uno stato finale.*

Esempio 5.1 La macchina di Turing deterministica rappresentata in Figura 5.4 calcola la funzione identità, vale a dire essa realizza la computazione $q_0 x \overset{*}{\vdash\!\!\!-} xßq_F x$, $\forall x \in \{0,1\}^*$.

Nel caso più generale in cui si vogliano definire funzioni da un arbitrario dominio $\mathcal{D}$ ad un arbitrario codominio $\mathcal{C}$ ci si può ricondurre ai trasduttori operanti su stringhe individuando opportune *codifiche* degli elementi di tali domini sotto forma di stringhe di caratteri.

Nel caso del calcolo di una funzione intera f, ad esempio, si può stabilire che l'alfabeto sia $\{0,1,2,3,4,5,6,7,8,9\}$, ma si può usare un qualunque altro

alfabeto Σ e codificare i numeri interi in base $\mid \Sigma \mid$. Pertanto per il calcolo del valore $m = f(n)$ la configurazione iniziale sarà $q_0 x$, dove x è la codificazione dell'intero n nell'alfabeto Σ prescelto. Per quanto riguarda la rappresentazione dell'output sul nastro, la configurazione finale sarà $x\b qy$, con $q \in F$, dove y è la codificazione di m nell'alfabeto Σ.[2]

Esercizio 5.7 Si consideri la funzione $d : \mathbb{N} \mapsto \mathbb{N}$ tale che $d(n) = 2n$. Definire due macchine di Turing che calcolano d assumendo che i naturali siano rispettivamente rappresentati in notazione binaria e in notazione decimale.

Quanto detto si applica naturalmente anche al caso del calcolo di funzioni con più argomenti. A tal fine è sufficiente osservare che una funzione a più argomenti, ad esempio da $\mathcal{D}_1 \times \mathcal{D}_2 \mapsto \mathcal{C}$, può essere vista come una funzione ad un solo argomento definita su un opportuno prodotto cartesiano, ad esempio l'insieme $\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2$.

Una semplice codifica per gli elementi del dominio così ottenuto potrà essere costituita dalla concatenazione delle codifiche dei diversi argomenti separate da un opportuno simbolo appositamente introdotto.

Esercizio 5.8 Definire una macchina di Turing che calcola la somma di due numeri naturali rappresentati in notazione binaria e separati dal simbolo #. Ad esempio, partendo dalla configurazione iniziale $q_0 10\#111$, la macchina termina nella configurazione finale $10\#111\b q1001$.

Un caso particolarmente significativo di calcolo di funzioni parziali si presenta quando il codominio della funzione è di cardinalità due. Tale caso infatti può essere interpretato come il calcolo del valore di verità di un predicato o, come già visto nella sezione precedente, come il riconoscimento di un linguaggio.

Esercizio 5.9 Definire una macchina di Turing che, dati due naturali in notazione binaria, separati dal simbolo #, calcola la funzione:

$$\min(n_1, n_2) = \begin{cases} \mathsf{T} & \text{se } n_1 < n_2 \\ \mathsf{F} & \text{se } n_1 \geq n_2 \end{cases}$$

[2]Successivamente, nel Capitolo 8, vedremo come nella Teoria della Complessità assuma rilevanza anche considerare codifiche non "prolisse", che rappresentino cioè l'argomento del calcolo di una funzione mediante stringhe di lunghezza sufficientemente limitata rispetto al valore dell'argomento. Ad esempio, vedremo che non è opportuno utilizzare la notazione unaria per rappresentare i numeri naturali in quanto essa è ridondante dato che richiede un numero di simboli esponenziale rispetto a una notazione in base maggiore o eguale a due.

5.3 Calcolabilità secondo Turing

Come già osservato, le macchine di Turing sono state introdotte con lo scopo di fornire una definizione formale del concetto di calcolo. Ciò ha permesso di definire in particolare le nozioni di linguaggio riconosciuto da una macchina di Turing e di funzione calcolata da una macchina di Turing. Avendo a disposizione quindi un concetto formale di algoritmo, possiamo riprendere le definizioni introdotte nella Sezione 2.5 e riformularle con riferimento alle macchine di Turing, sfruttando le definizioni date nella sezione precedente.

Definizione 5.9 *Un linguaggio è detto* decidibile secondo Turing *(*T-decidibile*) se esiste una macchina di Turing che lo riconosce.*

Come abbiamo visto nell'Esercizio 5.4 i linguaggi costituiti da stringhe palindrome sono T-decidibili.

In modo naturale la definizione di T-decidibilità può essere estesa ai problemi di decisione. Ad esempio, tenendo conto che gli automi a stati finiti sono evidentemente macchine di Turing di tipo particolare, possiamo dire che il problema dell'appartenenza di una stringa ad un linguaggio di tipo 3 è un problema decidibile. Esempi di problemi non T-decidibili verranno forniti nel seguito.

Osserviamo infine una ulteriore estensione del concetto di T-decidibilità, che può essere applicata alla valutazione di predicati, vale a dire di funzioni con codominio {VERO, FALSO}. Ad esempio, date una qualunque grammatica $\mathcal{G}$ di tipo 3 e una qualunque stringa x definita sull'alfabeto di $\mathcal{G}$, il predicato Appartiene$(\mathcal{G}, x)$ che risulta VERO se $x \in L(\mathrm{G})$ e FALSO altrimenti è T-decidibile.

Come detto nella Sezione 5.1, le macchine di Turing possono anche non arrestarsi su particolari valori dell'input. In tal caso è opportuno fare riferimento al concetto di semidecidibilità.

Definizione 5.10 *Un linguaggio è detto* semidecidibile secondo Turing *(*T-semidecidibile*) se esiste una macchina di Turing che lo accetta.*

Si noti che, poiché ogni linguaggio riconosciuto da una MT risulta anche accettato da essa, ogni linguaggio T-decidibile è anche T-semidecidibile. Il viceversa non è tuttavia vero: esempi di linguaggi T-semidecidibili ma non T-decidibili saranno illustrati nel Capitolo 7.

Naturalmente, anche il concetto di T-semidecidibilità può essere esteso ai problemi decisionali e ai predicati.

Esercizio 5.10 Date due stringhe x ed y, definiamo il predicato PiùLunga(x, y), che risulta VERO se $\mid x \mid > \mid y \mid$ e FALSO altrimenti. Dimostrare, definendo un'opportuna MT, che il predicato PiùLunga è T-decidibile.

Concetti analoghi possono essere definiti considerando le macchine di Turing non come riconoscitori ma come trasduttori.

Definizione 5.11 *Una funzione è detta* calcolabile secondo Turing *(*T-calcolabile*)se esiste una macchina di Turing che la calcola.*

Ad esempio, la funzione illustrata nell'Esempio 5.4 è T-calcolabile.

Esercizio 5.11 Dimostrare, definendo un'opportuna MT, che la funzione $f : \Sigma^* \rightarrow \Sigma^*$ tale che $f(x) = \tilde{(}x)$ è T-calcolabile.

Si noti che, in base alla Definizione 5.8, una funzione T-calcolabile non è necessariamente una funzione totale.

Inoltre, come detto nella sezione precedente, adottando opportune rappresentazioni dell'input e dell'output, ovvero degli argomenti e del valore della funzione, il concetto di T-calcolabilità può essere riferito a funzioni n-arie definite su domini e codomini arbitrari, ad esempio funzioni booleane, funzioni sui naturali, ecc.

È interessante osservare che i concetti di decidibilità e di calcolabilità basati sul modello di calcolo introdotto da Turing hanno una validità che travalica il modello stesso. Più precisamente, come viene descritto in Appendice A, tutti i tentativi fatti nell'ambito della logica matematica e dell'informatica teorica di definire concetti di calcolo basati su modelli e paradigmi alternativi alle macchine di Turing (come ad esempio quelli che introdurremo nei capitoli successivi, vale a dire le macchine a registri, le funzioni ricorsive, ecc.) hanno condotto a caratterizzare classi di insiemi decidibili e di funzioni calcolabili equivalenti a quelle introdotte da Turing. Questa circostanza ha condotto alla formulazione di un postulato che, seppure indimostrabile,[3] è unanimemente ritenuto valido: tale postulato, noto come *Tesi di Church-Turing*, afferma che ogni procedimento algoritmico espresso nell'ambito di un qualunque modello di calcolo è realizzabile mediante una macchina di Turing. In base alla tesi di Church-Turing in generale è dunque possibile parlare di insieme decidibile e di funzione calcolabile senza fare esplicito riferimento alla macchina di Turing.

5.4 Macchine di Turing multinastro

Il modello di macchina di Turing introdotto precedentemente, poiché estremamente elementare, risulta in molti casi di non agevole utilizzazione nello studio della risolubilità dei problemi. Ad esempio, se confrontiamo un automa a pila con una macchina di Turing ci rendiamo conto che nel primo caso esiste una chiara separazione tra il nastro di input e la memoria di lavoro, cosa che non si verifica nel secondo caso.

Un'utile variante della macchina di Turing classica è rappresentata dalla macchina di Turing *a più nastri* o *multinastro* (*MTM*), in cui si hanno più

[3] Una dimostrazione di tale postulato richiederebbe infatti che venisse dimostrata l'equivalenza rispetto alle macchine di Turing di tutti i possibili modelli di calcolo.

nastri contemporaneamente accessibili in scrittura e lettura che vengono consultati e aggiornati tramite più testine (una per nastro). Ad esempio, nel caso del riconoscimento delle stringhe palindrome risulta utile disporre di due nastri su entrambi i quali è riportata la stringa in ingresso, poiché la verifica della palindromia può essere efficientemente effettuata leggendo un nastro da sinistra verso destra e l'altro da destra verso sinistra.

Anche se una macchina di Turing multinastro, come vedremo, non aggiunge potere computazionale al modello delle macchine di Turing, essa consente di affrontare determinate classi di problemi in modo particolarmente agile.

Una tipica situazione di interesse, nella quale si rivela l'utilità della macchina multinastro, prevede la presenza di un nastro di input a sola lettura, vari nastri di lavoro utilizzabili in lettura ed in scrittura ed infine un nastro di output a sola scrittura: i nastri di input e di output sono accessibili in una sola direzione.

5.4.1 Descrizione e funzionamento delle MTM

Definizione 5.12 *Una macchina di Turing* a k nastri *($k \geq 2$) è una sestupla* $\mathcal{M}^{(k)} = \left\langle \Gamma, \text{ƀ}, Q, q_0, F, \delta^{(k)} \right\rangle$[4] *dove* $\Gamma = \bigcup_{i=1}^{k} \Gamma_i$ *è l'unione dei* k alfabeti di nastro $\Gamma_1, \ldots, \Gamma_k$ *non necessariamente distinti,* Q, q_0 *ed* F *hanno lo stesso significato che nel caso della macchina di Turing ad 1 nastro (vedi Definizione 5.1). La* funzione di transizione $\delta^{(k)}$ *è infine definita come*

$$\delta^{(k)} : (Q - F) \times \bar{\Gamma}_1 \times \ldots \times \bar{\Gamma}_k \mapsto Q \times \bar{\Gamma}_1 \times \ldots \times \bar{\Gamma}_k \times \{d, s, i\}^k .$$

Il senso di questa definizione di $\delta^{(k)}$ è che la macchina esegue una transizione a partire da uno stato interno q_i e con le k testine — una per nastro — posizionate sui caratteri $a_{i_1}, \ldots, a_{i_k}$, cioè se $\delta^{(k)}(q_i, a_{i_1}, \ldots, a_{i_k}) = (q_j, a_{j_1}, \ldots, a_{j_k}, z_{j_1}, \ldots, z_{j_k})$ si porta nello stato q_j, scrive i caratteri $a_{j_1}, \ldots, a_{j_k}$ sui rispettivi nastri e fa compiere alle testine i rispettivi spostamenti — a destra, a sinistra o nessuno spostamento, come specificato dagli $z_{j_\ell} \in \{d, s, i\}$, $\ell = 1, \ldots, k$.

Uno schema di massima per la macchina di Turing multinastro è illustrato nella Figura 5.5.

5.4.2 Configurazioni, transizioni e computazioni di una MTM

Nel caso di macchina multinastro la *configurazione istantanea* deve descrivere, analogamente al caso di macchina a un nastro (vedi Sezione 5.1.1), lo stato interno, i contenuti dei nastri e le posizioni delle testine. A volte, per comodità, si usano macchine in cui i diversi nastri adottano alfabeti fra loro differenti.

[4] Si noti che, per semplificare la notazione, utilizzeremo l'apice (k) che indica il numero di nastri di una macchina di Turing multinastro solo quando ciò sia reso necessario dal contesto.

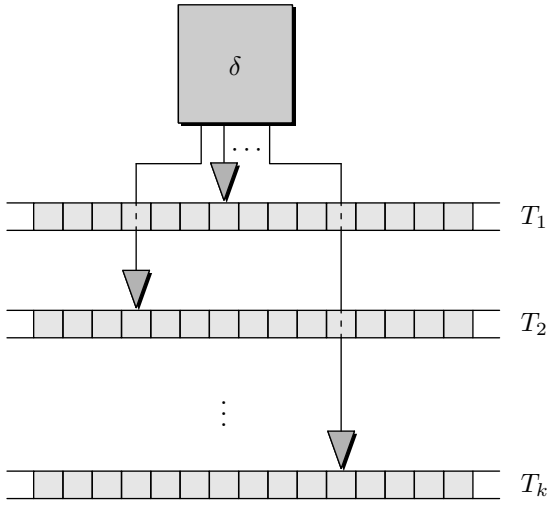


FIGURA 5.5 Macchina di Turing a più nastri.

In ogni caso per ciascun nastro si rappresenta, come per le macchine di Turing ad un solo nastro, la minima porzione contenente caratteri distinti da $\bar{b}$.

Sia data la macchina di Turing multinastro $\mathcal{M}^{(k)} = \left\langle \Gamma, \bar{b}, Q, q_0, F, \delta^{(k)} \right\rangle$, con $\Gamma = \bigcup_{i=1}^{k} \Gamma_i$.

Definizione 5.13 *Una* configurazione istantanea *di una macchina di Turing multinastro è una stringa del tipo*

$$q \# \alpha_1 \uparrow \beta_1 \# \alpha_2 \uparrow \beta_2 \# s \# \alpha_k \uparrow \beta_k$$

in cui, coerentemente al caso di macchine di Turing ad un solo nastro, $\alpha_i \in \Gamma_i \bar{\Gamma}_i^* \cup \{\varepsilon\}$ *e* $\beta_i \in \bar{\Gamma}_i^* \Gamma_i \cup \{\bar{b}\}$*, per* $i = 1, \ldots, k$*, essendo* $\uparrow$ *il simbolo che indica la posizione di ogni testina e* $\#$ *un separatore non appartenente al alcun* $\bar{\Gamma}_i$ *che marca l'inizio di ogni stringa* $\alpha_i \uparrow \beta_i$*.*

Per il concetto di configurazione *finale* possiamo estendere in modo naturale la definizione data nel caso di macchina di Turing ad un solo nastro.

Definizione 5.14 *Una configurazione* $q \# \alpha_1 \uparrow \beta_1 \# \alpha_2 \uparrow \beta_2 \# s \# \alpha_k \uparrow \beta_k$ *si dice* finale *se* $q \in F$*.*

Il concetto di configurazione iniziale di una macchina di Turing multinastro richiede invece qualche precisazione. È naturale pensare che all'inizio della

computazione la macchina di Turing multinastro contenga l'input x su di un nastro — ad esempio il primo, assumendo quindi che $x \in \Gamma_1^*$ — e che tutti gli altri nastri contengano solo simboli $\bar{b}$. Tuttavia risulta comodo adottare la convenzione che tali nastri contengano un *simbolo iniziale* $Z_0 \notin \bigcup_{i=1}^{k} \bar{\Gamma}_i$ sempre presente all'inizio di una computazione. Le posizioni iniziali delle testine saranno tali che sul primo nastro il carattere osservato sia il primo carattere della stringa di input x e che sugli altri nastri il carattere osservato sia sempre Z_0.

Allora, ridefinendo $\bar{\Gamma}_i$, $i = 2, \ldots, k$, come segue:

$$\bar{\Gamma}_i = \Gamma_i \cup \{\bar{b}, Z_0\}$$

e tenendo conto di tale nuovo significato in tutte le definizioni date precedentemente possiamo introdurre il concetto di configurazione iniziale.

Definizione 5.15 *Una configurazione* $q \# \alpha_1 \uparrow \beta_1 \# \alpha_2 \uparrow \beta_2 \# s \# \alpha_k \uparrow \beta_k$ *si dice* iniziale *se* $\alpha_i = \varepsilon$, $i = 1, \ldots, k$, $\beta_1 \in \Gamma_1^*$, $\beta_i = Z_0$, $i = 2, \ldots, k$ *e* $q = q_0$.

Definizione 5.16 *L'applicazione della funzione di transizione* $\delta^{(k)}$ *ad una configurazione si dice* transizione *o* mossa *o* passo computazionale *di una macchina di Turing multinastro.*

Per indicare la transizione fra configurazioni si userà la stessa notazione $\underset{\mathcal{M}}{\vdash\!\!\!-\!\!\!-}$ definita in Sezione 5.1.1 per le macchine di Turing.

Il concetto di *computazione* per una macchina di Turing multinastro è del tutto analogo a quello delle macchine di Turing, per cui vale quanto detto in Sezione 5.1.2. Valgono inoltre le considerazioni svolte in Sezione 5.1.2 a proposito di macchine usate come dispositivi riconoscitori. Per il caso di dispositivi trasduttori, si può estendere facilmente quanto scritto nella Sezione 5.2 specificando quale sia la convenzione da adottare sul tipo desiderato di configurazione finale (vedi Definizione 5.8).

Ad esempio, per una macchina di Turing multinastro $\mathcal{M}$ a k nastri, possiamo dire che $\mathcal{M}$ calcola la funzione $f_{\mathcal{M}}(x)$ definita come

$$q_0 \# \uparrow x \# \uparrow Z_0 \# s \# \uparrow Z_0 \underset{\mathcal{M}}{\vdash\!\!\!-\!\!\!-} q \# \uparrow x \# \uparrow f_{\mathcal{M}}(x) \# \uparrow \bar{b} \# s \# \uparrow \bar{b}$$

dove $q \in F$.

Infine, osserviamo che nel caso particolare di macchina di Turing multinastro con un nastro di input a sola lettura monodirezionale, h nastri di lavoro e un nastro di output a sola scrittura monodirezionale, si indicano spesso con Σ l'alfabeto (del nastro) di input, con Γ l'alfabeto (dei nastri) di lavoro (assumendo cioè $\Gamma = \Gamma_1 = s = \Gamma_h$) e con O l'alfabeto (del nastro) di output. In questo caso si fa uso del simbolo iniziale Z_0 solo sui nastri di lavoro, in quanto il nastro di output è del tipo a sola scrittura.

Esercizio 5.12 Come è possibile semplificare la definizione della funzione di transizione nel caso di macchina di Turing multinastro con nastro di input unidirezionale a sola lettura, nastro di output unidirezionale a sola scrittura e $h \geq 0$ nastri di lavoro?

Concludiamo la presente sezione con un dettagliato esempio di macchina di Turing multinastro che riconosce una classe particolare di stringhe palindrome.

Esempio 5.2 Come esempio di macchina di Turing multinastro, consideriamo il caso di una macchina in grado di riconoscere stringhe del tipo $xc\tilde{x}$, dove $x \in \{a, b\}^+$.

Possiamo osservare che da un punto di vista algoritmico il riconoscimento è facilitato dalla presenza del carattere "c" che, non appartenendo all'alfabeto su cui è costruita la sottostringa x, marca la fine di x e l'inizio di $\tilde{x}$.

Per riconoscere questo tipo di stringa si può usare una macchina con due nastri (Figura 5.6), uno di input, monodirezionale e a sola lettura, ed uno di lavoro, utilizzato come pila.

La macchina funziona nel modo seguente: viene dapprima scandito l'input copiandolo sul nastro di lavoro fino a quando si incontra il separatore c. Quindi continua la scansione dell'input, mentre ora il nastro di lavoro viene scandito in direzione opposta alla precedente, e si confrontano i caratteri in input con quelli presenti sul nastro di lavoro.

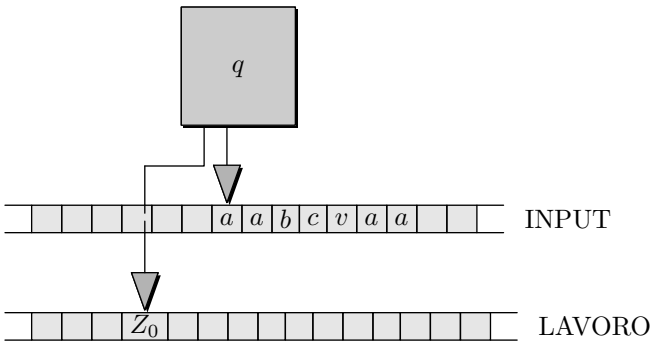


FIGURA 5.6 Macchina di Turing multinastro per riconoscere stringhe palindrome.

L'alfabeto del nastro di input (o, più brevemente, alfabeto di input) della macchina è $\Sigma = \{a, b, c\}$, mentre quello del nastro di lavoro è $\Gamma = \{a, b\}$, perché occorre scrivere x sulla nastro di lavoro per poterla poi verificare quando si legge $\tilde{x}$. La configurazione iniziale della macchina è allora data da $q_0 \# \uparrow xc\tilde{x} \# \uparrow Z_0$.

La macchina ha tre stati. Il primo, q_0, usato per la scansione della sottostringa x, il secondo, q_1, per la scansione di $\tilde{x}$ e il terzo, q_2, come stato finale. Abbiamo quindi $Q = \{q_0, q_1, q_2\}$ e $F = \{q_2\}$.

Passiamo ora a descrivere le regole di transizione. Una prima coppia di regole riguarda il caso in cui la testina del nastro di lavoro è situata sul simbolo iniziale Z_0 e la testina del nastro di input è posizionata sul primo carattere; ci si trova ovviamente

in fase di copiatura. Se il primo carattere della stringa di input è a, allora la testina del nastro di input si deve spostare di una cella a destra, mentre sul nastro di lavoro si deve sostituire Z_0 con a e ci si deve spostare a destra, come in Figura 5.7.

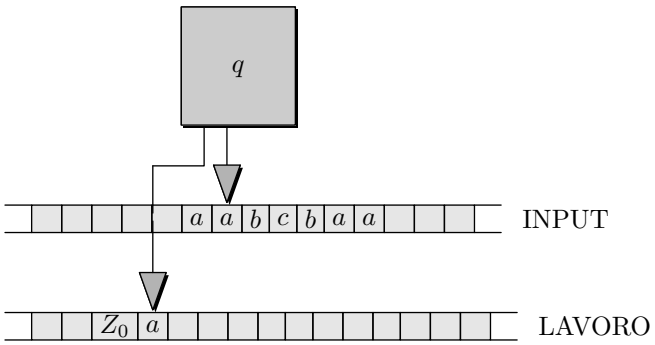


FIGURA 5.7 Lettura del primo carattere della stringa palindroma.

La regola risultante è quindi

$$\delta(q_0, a, Z_0) = (q_0, a, d),$$

in cui il codominio della δ ha arità 3 poiché, a causa delle caratteristiche della macchina, che ad ogni passo di computazione sposta invariabilmente la testina del nastro di input a destra, è sufficiente specificare le sole azioni relative al nastro di lavoro. Se invece il primo carattere della stringa di input è b si ha l'analoga regola

$$\delta(q_0, b, Z_0) = (q_0, b, d).$$

Dopo aver letto il primo carattere della stringa di input, si è sostituito il simbolo iniziale del nastro di lavoro, e la testina di lavoro è necessariamente posizionata su un simbolo $\bar{b}$. Si hanno quindi due regole del tutto analoghe alle precedenti, con la sola differenza che il carattere del nastro di lavoro che viene sostituito è $\bar{b}$ anziché Z_0:

$$\delta(q_0, a, \bar{b}) = (q_0, a, d)$$
$$\delta(q_0, b, \bar{b}) = (q_0, b, d).$$

Le quattro regole appena descritte consentono di leggere tutta la stringa x fino al carattere c escluso, copiandola sul nastro di lavoro. Non appena si legge il carattere c, si deve passare dalla fase di copiatura, in cui lo stato è q_0, alla fase di verifica (Figura 5.8), in cui lo stato è q_1. Ciò è espresso dalla regola

$$\delta(q_0, c, \bar{b}) = (q_1, \bar{b}, s).$$

Durante la fase di verifica, la macchina di Turing continua a scorrere il nastro di input da sinistra verso destra, mentre al tempo stesso esamina le celle del nastro di lavoro da destra verso sinistra. Ad ogni passo, viene verificato che i caratteri letti sui

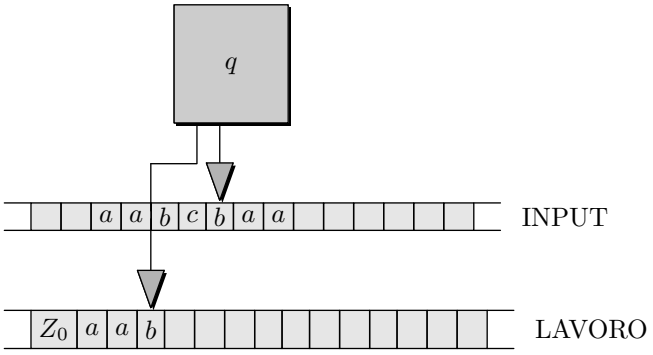


FIGURA 5.8 Fase di verifica dopo la lettura del carattere c.

due nastri siano uguali. È facile rendersi conto che la stringa è palindroma se e solo se tale condizione è vera fino a quando l'input non è stato letto completamente e se a fine lettura la stringa copiata sul nastro di lavoro durante la prima fase è stata riletta completamente.

Nel generico passo della scansione di verifica, in cui la stringa costruita sul nastro di lavoro viene scandita a ritroso, si possono presentare due situazioni diverse.

La prima situazione corrisponde al caso in cui i due caratteri letti sul nastro di input e sul nastro di lavoro siano uguali, dal che deriva che la sottostringa potrebbe essere palindroma e che la verifica va continuata. Ciò viene rappresentato dalle due regole

$$\delta(q_1, a, a) = (q_1, a, s)$$
$$\delta(q_1, b, b) = (q_1, b, s),$$

che, per l'appunto, fanno sì che la verifica continui sui caratteri successivi e dalla regola

$$\delta(q_1, \text{b}, \text{b}) = (q_2, \text{b}, i),$$

che corrisponde alla verifica del fatto che la lettura dell'input e della stringa sul nastro di lavoro sono terminate contemporaneamente. La conseguente accettazione della stringa è rappresentata dalla transizione nello stato finale q_2.

L'altra situazione possibile corrisponde al caso in cui i caratteri siano diversi, e si possa quindi immediatamente dedurre che la stringa non sia palindroma: tale eventualità viene gestita non definendo le corrispondenti transizioni e facendo sìquindi che la macchina di Turing termini la sua computazione in $q_1 \notin F$.

A titolo esemplificativo, si riportano di seguito le computazioni massimali corrispondenti ai due input *bacab* e *acb*.

$q_0 \# \uparrow bacab \# \uparrow Z_0$
$q_0 \# b \uparrow acab \# b \uparrow \text{ƀ}$
$q_0 \# ba \uparrow cab \# ba \uparrow \text{ƀ}$
$q_1 \# bac \uparrow ab \# b \uparrow a$
$q_1 \# baca \uparrow b \# \uparrow ba$
$q_1 \# bacab \uparrow \text{ƀ} \# \uparrow \text{ƀ}ba$
$q_2 \# bacab\text{ƀ} \uparrow \text{ƀ} \# \uparrow \text{ƀ}ba$

$q_0 \# \uparrow acb \# \uparrow Z_0$
$q_0 \# a \uparrow cb \# a \uparrow \text{ƀ}$
$q_1 \# ac \uparrow b \# \uparrow a$

Si osservi che mentre la prima computazione termina con una configurazione finale, per cui è una computazione accettante, la seconda termina con una configurazione non finale, per cui è una computazione rifiutante.

Esercizio 5.13 Definire una macchina di Turing multinastro che riconosce le stringhe palindrome del tipo $x\tilde{x}$, dove $x \in \{a, b\}^+$. Si noti che in questo caso non si hanno separatori che consentono di individuare l'inizio della sottostringa $\tilde{x}$, per cui ogni carattere letto dopo il primo potrebbe essere il primo carattere di $\tilde{x}$. Ne segue che non è possibile utilizzare una macchina come quella dell'esempio precedente per riconoscere tali stringhe.

5.4.3 Equivalenza tra MTM e MT

Si è accennato sopra al fatto che le macchine di Turing multinastro hanno lo stesso potere computazionale delle macchine a un solo nastro. In questa sezione viene presentata la traccia della dimostrazione di come una macchina di Turing multinastro possa venir simulata tramite una macchina di Turing ad un solo nastro.

Nella simulazione si fa riferimento a macchine di Turing multitraccia. Tali dispositivi sono macchine di Turing ad un nastro aventi il nastro stesso suddiviso in *tracce*: queste ultime possono essere immaginate disposte sul nastro come le piste di un registratore multipista (vedi Figura 5.9. Così, se la macchina ha m tracce, la testina con una singola operazione può accedere, per leggere e/o scrivere, agli m caratteri disposti sulle tracce in corrispondenza della testina stessa.

È molto semplice modificare una macchina di Turing in modo che simuli una macchina con nastro ad m tracce. Se assumiamo, per generalità, che sulle tracce si usino rispettivamente gli alfabeti $\bar{\Gamma}_1, \bar{\Gamma}_2, \ldots, \bar{\Gamma}_m$, basta adottare per la macchina un alfabeto $\bar{\Gamma}$ tale che $|\bar{\Gamma}| = |\bar{\Gamma}_1 \times s \times \bar{\Gamma}_m|$, definendo una funzione

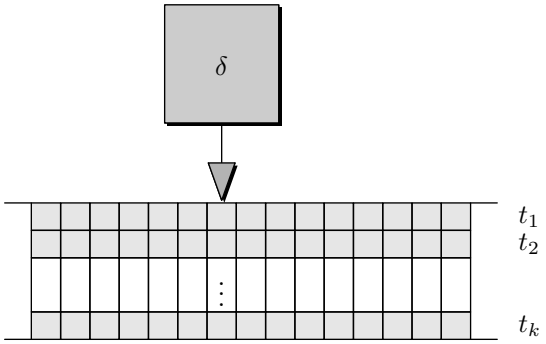


FIGURA 5.9 Macchina di Turing multitraccia.

iniettiva dall'insieme $\bar{\Gamma}_1 \times s \times \bar{\Gamma}_m$ all'insieme $\bar{\Gamma}$: tale funzione assumerà in particolare il simbolo $\bar{b}$ di $\bar{\Gamma}$ alla m-pla $(\bar{b}, \bar{b}, \ldots, \bar{b})$. Una notazione utile per indicare gli elementi di $\bar{\Gamma}$ che corrispondono a elementi di $\bar{\Gamma}_1 \times s \times \bar{\Gamma}_m$, e che sarà usata più avanti nel volume, è la seguente:

$$\begin{bmatrix} a_{i_1} \\ a_{i_2} \\ \vdots \\ a_{i_m} \end{bmatrix}$$

dove $a_{i_j} \in \bar{\Gamma}_j$.

Passiamo ora a dimostrare il seguente teorema.

Teorema 5.1 *Data una macchina di Turing* $\mathcal{M} = \left\langle \Gamma, \bar{b}, Q, q_0, F, \delta^{(k)} \right\rangle$ *a* k *nastri, esiste una macchina a un nastro che simula* t *passi di* $\mathcal{M}$ *in* $O(t^2)$ *transizioni usando un alfabeto di dimensione* $O((2\,|\Gamma\,|)^k)$.

Dimostrazione. La dimostrazione è costruttiva e consiste nel definire una macchina di Turing $\mathcal{M}' = \langle \Gamma', \bar{b}, Q', {q'}_0, F', \delta' \rangle$ che simula $\mathcal{M}$.

$\mathcal{M}'$ è una macchina a un solo nastro diviso in $2k$ tracce: di queste k sono destinate a contenere i caratteri presenti nei k nastri di $\mathcal{M}$ e le rimanenti a marcare con il carattere "$\downarrow$" le posizioni delle k testine sui k nastri di $\mathcal{M}$. La situazione è esemplificata in Figura 5.10.

È chiaro che questo nastro multitraccia consente di rappresentare il contenuto di ciascun nastro di $\mathcal{M}$ (nelle tracce di posizione pari) e la posizione di ciascuna delle sue testine (nelle tracce di posizione dispari). La struttura

$\cdots$	b	b	b	b	$\downarrow$	b	b	b	b	$\cdots$	t_1
$\cdots$	b	b	b	a	b	b	a	b	b	$\cdots$	t_2
$\cdots$	b	b	b	b	$\downarrow$	b	b	b	b	$\cdots$	t_3
$\cdots$	b	b	b	b	a	b	c	b	b	$\cdots$	t_4
$\cdots$	b	b	$\downarrow$	b	b	b	b	b	b	$\cdots$	t_5
$\cdots$	b	d	e	d	d	e	b	b	b	$\cdots$	t_6
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\cdots$	b	b	b	b	b	$\downarrow$	b	b	b	$\cdots$	t_{2k-1}
$\cdots$	f	f	g	h	g	b	b	b	b	$\cdots$	t_{2k}

FIGURA 5.10 Nastro della macchina di Turing multitraccia che simula una macchina di Turing multinastro a k nastri.

di una singola cella del nastro di $\mathcal{M}'$, con le $2k$ tracce messe in evidenza, è illustrata in Figura 5.11.

Come osservato sopra, è allora sufficiente adottare un alfabeto di nastro Γ' di cardinalità opportuna e fissare una corrispondenza iniettiva dalle $2k$-ple di caratteri di una cella sul nastro multitraccia agli elementi di Γ' per ottenere una macchina di Turing con nastro rappresentante le $2k$ tracce desiderate. La situazione del nastro di $\mathcal{M}'$ all'inizio della computazione è rappresentata in Figura 5.12.

La funzione di transizione della $\mathcal{M}$ ha elementi

$$\delta^{(k)}(q_i, a_{i_1}, \ldots, a_{i_k}) = (q_j, a_{j_1}, \ldots, a_{j_k}, z_{j_1}, \ldots, z_{j_k})$$

con $z_{j_1}, \ldots, z_{j_k} \in \{d, s, i\}$. La corrispondente δ' deve, in corrispondenza di ogni elemento della $\delta^{(k)}$, individuare lo stato corrente e la posizione del marcatore per ogni traccia per poi effettuare la transizione, cioè scrivere caratteri, spostare marcatori in base alla $\delta^{(k)}$ e passare ad un altro stato interno.

Passiamo ora a valutare i costi della simulazione in termini di tempo (numero di transizioni) e in termini di cardinalità dell'alfabeto della macchina $\mathcal{M}'$.

Ad ogni passo di $\mathcal{M}$, la macchina di Turing $\mathcal{M}'$ deve eseguire una sequenza di passi che consiste nel percorrere la porzione di nastro compresa tra i due marcatori più lontani. Dal momento che ad ogni passo ogni marcatore si sposta al più di una casella, per cui i marcatori si possono allontanare reciprocamente al più di due caselle, dopo t passi di $\mathcal{M}$ i marcatori si possono essere allontanati

$c_{i_1} \in \{\text{ƀ}, \downarrow\}$	t_1
$a_{i_1} \in \Gamma_1 \cup \{\text{ƀ}\}$	t_2
$c_{i_2} \in \{\text{ƀ}, \downarrow\}$	t_3
$a_{i_2} \in \Gamma_2 \cup \{\text{ƀ}, Z_0\}$	t_4
$c_{i_3} \in \{\text{ƀ}, \downarrow\}$	t_5
$a_{i_3} \in \Gamma_3 \cup \{\text{ƀ}, Z_0\}$	t_6
$\vdots$	
$c_{i_k} \in \{\text{ƀ}, \downarrow\}$	t_{2k-1}
$a_{i_k} \in \Gamma_k \cup \{\text{ƀ}, Z_0\}$	t_{2k}

FIGURA 5.11 Struttura della singola cella di nastro di $\mathcal{M}'$.

$\cdots$	ƀ	ƀ	$\downarrow$	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	$\cdots$	t_1
$\cdots$	ƀ	ƀ	a	b	b	b	a	ƀ	ƀ	$\cdots$	t_2
$\cdots$	ƀ	ƀ	$\downarrow$	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	$\cdots$	t_3
$\cdots$	ƀ	ƀ	Z_0	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	$\cdots$	t_4
$\cdots$	ƀ	ƀ	$\downarrow$	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	$\cdots$	t_5
$\cdots$	ƀ	ƀ	Z_0	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	$\cdots$	t_6
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\cdots$	ƀ	ƀ	$\downarrow$	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	$\cdots$	t_{2k-1}
$\cdots$	ƀ	ƀ	Z_0	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	$\cdots$	t_{2k}

FIGURA 5.12 Situazione iniziale sul nastro della $\hat{\mathcal{M}}$ per l'input $x = abbba$.

al più di $2t$ caselle, spostandosi a destra o a sinistra: quindi se $\mathcal{M}$ esegue t passi, $\mathcal{M}'$ ne esegue al più $2\sum_{i=1}^{t} i = (t^2 + t) = O(t^2)$.

Per quanto riguarda invece il costo della simulazione in termini del numero di caratteri, abbiamo, osservando che $|\bar{\Gamma}_1| = |\Gamma_1| + 1$ e che $|\bar{\Gamma}_i| = |\Gamma_i| + 2$ per $2 \leq i \leq k$, e moltiplicando le cardinalità degli alfabeti in giuoco (vedi Figura 5.11)

$$|\bar{\Gamma}'| = 2^k (|\Gamma_1| + 1) \prod_{i=2}^{k} (|\Gamma_i| + 2) = O((2 \max_{1 \leq i \leq k} |\Gamma_i|)^k) \tag{5.1}$$

Si noti che in caso di macchina di Turing multinastro con nastro di output monodirezionale e a sola scrittura, come anche in altre varianti di macchine di Turing multinastro (più nastri a sola lettura e/o scrittura, più nastri monodirezionali, etc.), l'Equazione 5.1 rimane asintoticamente valida, subendo variazioni solo nelle costanti. □

Come conseguenza del teorema precedente, possiamo asserire che, data una macchina di Turing a più nastri che riconosce o accetta un linguaggio L (o, operando come trasduttore, calcola una funzione f), esiste una macchina di Turing a un nastro che consegue lo stesso risultato computazionale, cioè riconosce o accetta lo stesso linguaggio L (ovvero calcola la stessa funzione f).

5.5 Macchine di Turing non deterministiche

Così come per gli altri tipi di automi considerati in precedenza, una importante variante nella struttura delle macchine di Turing è offerta dall'introduzione del non determinismo. Come visto nei Capitoli 3 e 4, il non determinismo può in alcuni casi accrescere il potere computazionale di un dispositivo di calcolo, mentre in altri ciò non avviene. Ad esempio, gli automi a stati finiti non deterministici hanno lo stesso potere computazionale degli automi deterministici (vedi Teorema 3.1), mentre gli automi a pila non deterministici sono più potenti di quelli deterministici (vedi Sezione 4.5).

Nel caso delle macchine di Turing vedremo che, se non abbiamo limitazioni nelle risorse (tempo, memoria) utilizzate dalla macchina, il modello non deterministico è equivalente dal punto di vista computazionale a quello deterministico.

Come vedremo nel Capitolo 9, in alcuni casi in cui abbiamo invece limitazioni di risorse, come ad esempio nel caso di macchine di Turing che operano in tempo polinomiale, la determinazione della relazione tra il potere computazionale della macchina non deterministica e di quella deterministica rappresenta tuttora un importante problema aperto.

Nella trattazione che seguirà le macchine di Turing non deterministiche saranno considerate solo come dispositivi per problemi decisionali.

5.5.1 Descrizione e funzionamento delle MTND

Nel caso di macchina di Turing non deterministica (MTND) la funzione di transizione δ_N è definita, coerentemente con le Definizioni 3.7 (per gli automi a stati finiti) e 4.11 (per gli automi a pila), come funzione (parziale) sull'insieme delle parti del codominio della funzione di transizione deterministica.

Definizione 5.17 *Una* Macchina di Turing non deterministica $\mathcal{M}$ *è una sestupla* $\mathcal{M} = \langle \Gamma, \b, Q, q_0, F, \delta_N \rangle$*, in cui* Γ *è l'*alfabeto dei simboli di nastro*,* $\b \notin \Gamma$ *è un carattere speciale denominato* blank*,* Q *è un insieme finito e non vuoto di* stati*,* $q_0 \in Q$ *è lo* stato iniziale*,* $F \subseteq Q$ *è l'insieme degli* stati finali *e la funzione di transizione* δ_N *è una funzione parziale* $\delta_N : Q \times \bar{\Gamma} \mapsto \mathcal{P}(Q \times \bar{\Gamma} \times \{d, s, i\})$.

Esempio 5.3 Si consideri la macchina di Turing non deterministica $\mathcal{M}$ con $\Gamma = \{a, b, c, d\}$, $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}\}$, $F = \{q_{10}\}$ e la funzione δ_N definita come segue:

	a	b	c	d	$\b$
q_0	$\{(q_0, a, d), (q_1, c, d)\}$	$\{(q_0, b, d), (q_2, c, d)\}$	–	–	–
q_1	$\{(q_1, a, d), (q_3, d, s)\}$	$\{(q_1, b, d)\}$	–	–	–
q_2	$\{(q_2, a, d)\}$	$\{(q_2, b, d), (q_3, d, s)\}$	–	–	–
q_3	$\{(q_3, a, s)\}$	$\{(q_3, b, s)\}$	$\{(q_4, c, d)\}$	–	–
q_4	$\{(q_6, c, d)\}$	$\{(q_7, c, d)\}$	–	–	–
q_5	$\{(q_5, a, d)\}$	$\{(q_5, b, d)\}$	–	$\{(q_7, d, d)\}$	–
q_6	$\{(q_6, a, d)\}$	$\{(q_6, b, d)\}$	–	$\{(q_8, d, d)\}$	–
q_7	–	$\{(q_9, d, s)\}$	–	$\{(q_7, d, d)\}$	–
q_8	$\{(q_9, d, s)\}$	–	–	$\{(q_8, d, d)\}$	–
q_9	$\{(q_3, a, s)\}$	$\{(q_3, b, s)\}$	$\{(q_{10}, x, i)\}$	$\{(q_9, d, s)\}$	–
q_{10}	–	–	–	–	–

La macchina di Turing $\mathcal{M}$, che ha grado di non determinismo $\nu(\mathcal{M}) = 2$, data una stringa di input $x \in \{a, b\}^*$, la accetta se e solo se esiste una stringa $y \in \{a, b\}^*$ con $|y| \geq 2$ tale che $x = uyyv$, con $u, v \in \{a, b\}^*$.

Date due configurazioni c, c', indicheremo ancora con $c \underset{\mathcal{M}}{\vdash\!\!\!-} c'$ il fatto che c e c' siano legate dalla relazione di transizione definita dalla macchina non deterministica $\mathcal{M}$, o, in altre parole, la possibilità di ottenere c' in corrispondenza ad una "scelta di applicazione di δ_N su c. Ad esempio, nel caso della macchina di Turing non deterministica dell'Esempio 5.3, abbiamo che, mediante applicazione della δ_N, $aabq_2bbab \underset{\mathcal{M}}{\vdash\!\!\!-} aaq_3bdbab$. Inoltre, abbiamo anche $aabq_2bbab \underset{\mathcal{M}}{\vdash\!\!\!-} aabbq_2bab$.

Alla luce di quanto detto, possiamo estendere alle macchine di Turing non deterministiche i concetti di computazione, computazione massimale, computazione di accettazione e computazione di rifiuto introdotti nella Sezione 2.5.

Esempio 5.4 Con riferimento alla macchina di Turing non deterministica vista nell'Esempio 5.3 si può osservare che a partire dalla configurazione iniziale q_0abab essa dà luogo alle seguenti computazioni.

1. $q_0abab \underset{\mathcal{M}}{\vdash\!\!\!-} cq_1bab \underset{\mathcal{M}}{\vdash\!\!\!-} cbq_1ab \underset{\mathcal{M}}{\vdash\!\!\!-} cq_3bdb \underset{\mathcal{M}}{\vdash\!\!\!-} q_3cbdb \underset{\mathcal{M}}{\vdash\!\!\!-} cq_4bdb \underset{\mathcal{M}}{\vdash\!\!\!-} ccq_5db \underset{\mathcal{M}}{\vdash\!\!\!-} ccdq_7b \underset{\mathcal{M}}{\vdash\!\!\!-} ccq_9dd \underset{\mathcal{M}}{\vdash\!\!\!-} ccq_{10}dd$
2. $q_0abab \underset{\mathcal{M}}{\vdash\!\!\!-} cq_1bab \underset{\mathcal{M}}{\vdash\!\!\!-} cbq_1ab \underset{\mathcal{M}}{\vdash\!\!\!-} cbaq_1b \underset{\mathcal{M}}{\vdash\!\!\!-} cbabq_1ß$
3. $q_0abab \underset{\mathcal{M}}{\vdash\!\!\!-} aq_0bab \underset{\mathcal{M}}{\vdash\!\!\!-} acq_2ab \underset{\mathcal{M}}{\vdash\!\!\!-} acaq_2b \underset{\mathcal{M}}{\vdash\!\!\!-} acq_3ad \underset{\mathcal{M}}{\vdash\!\!\!-} aq_3cad \underset{\mathcal{M}}{\vdash\!\!\!-} acq_4ad \underset{\mathcal{M}}{\vdash\!\!\!-} accq_6d \underset{\mathcal{M}}{\vdash\!\!\!-} accdq_8ß$
4. $q_0abab \underset{\mathcal{M}}{\vdash\!\!\!-} aq_0bab \underset{\mathcal{M}}{\vdash\!\!\!-} acq_2ab \underset{\mathcal{M}}{\vdash\!\!\!-} acaq_2b \underset{\mathcal{M}}{\vdash\!\!\!-} acabq_2ß$
5. $q_0abab \underset{\mathcal{M}}{\vdash\!\!\!-} aq_0bab \underset{\mathcal{M}}{\vdash\!\!\!-} abq_0ab \underset{\mathcal{M}}{\vdash\!\!\!-} abcq_1b \underset{\mathcal{M}}{\vdash\!\!\!-} abcbq_1ß$
6. $q_0abab \underset{\mathcal{M}}{\vdash\!\!\!-} aq_0bab \underset{\mathcal{M}}{\vdash\!\!\!-} abq_0ab \underset{\mathcal{M}}{\vdash\!\!\!-} abaq_0b \underset{\mathcal{M}}{\vdash\!\!\!-} abacq_1ß$
7. $q_0abab \underset{\mathcal{M}}{\vdash\!\!\!-} aq_0bab \underset{\mathcal{M}}{\vdash\!\!\!-} abq_0ab \underset{\mathcal{M}}{\vdash\!\!\!-} abaq_0b \underset{\mathcal{M}}{\vdash\!\!\!-} ababq_0ß.$

Naturalmente, come nel caso degli automi a stati finiti e degli automi a pila non deterministici, anche il comportamento di una MTND può essere descritto mediante un albero, i cui rami sono costituiti da tutte le computazioni che la macchina realizza a partire dalla configurazione iniziale. In Figura 5.13 l'insieme delle computazioni elencate nell'Esempio 5.3 viene rappresentato come albero di computazione.

Sia $\mathcal{M} = \langle \Gamma, ß, Q, q_0, F, \delta_{\mathrm{N}} \rangle$ una macchina di Turing non deterministica.

Definizione 5.18 *Dato un alfabeto $\Sigma \subseteq \Gamma$, una stringa $x \in \Sigma^*$ è* accettata *dalla macchina $\mathcal{M}$ se esiste una computazione accettante $c_0, \ldots, c_n$ di $\mathcal{M}$, con $c_0 = \{q_0x\}$.*

Definizione 5.19 *Dato un alfabeto $\Sigma \subseteq \Gamma$, una stringa $x \in \Sigma^*$ è* rifiutata *dalla macchina $\mathcal{M}$ se tutte le computazioni di $\mathcal{M}$, a partire dalla configurazione $\{q_0x\}$, sono rifiutanti.*

Esempio 5.5 Come si può vedere, la stringa *abab* è accettata dalla MTND dell'Esempio 5.3 in quanto una delle computazioni (la numero 2, in particolare), termina in una configurazione di accettazione.

Esempio 5.6 La stringa *ab* è rifiutata dalla MTND dell'Esempio 5.3 poiché, a partire dalla configurazione iniziale $\{q_0ab\}$, tutte le computazioni sono rifiutanti.

1. $q_0ab \underset{\mathcal{M}}{\vdash\!\!\!-} aq_0b \underset{\mathcal{M}}{\vdash\!\!\!-} aq_0bß$
2. $q_0ab \underset{\mathcal{M}}{\vdash\!\!\!-} cq_1b \underset{\mathcal{M}}{\vdash\!\!\!-} cbq_1ß$

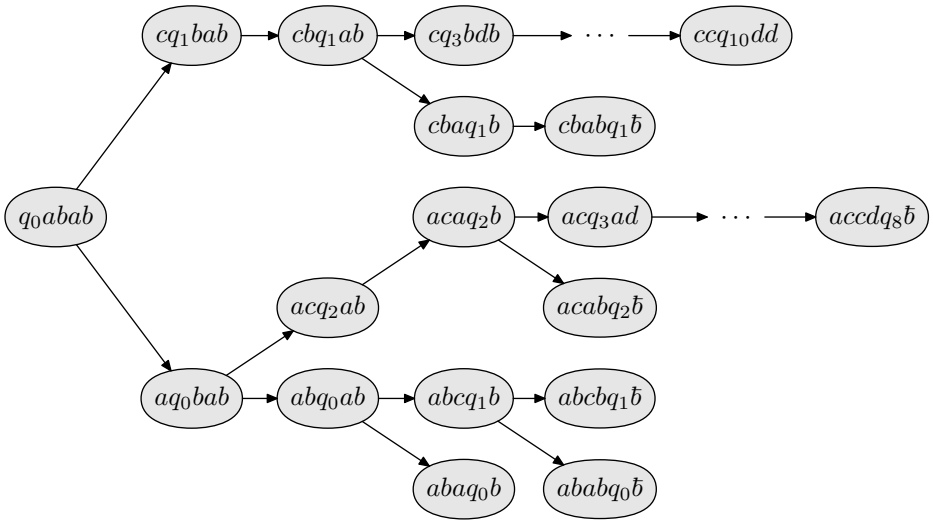


FIGURA 5.13 Albero di computazione della MTND dell'Esempio 5.3 su input $abab$.

---

3. $q_0ab \underset{\mathcal{M}}{\vdash} aq_0b \underset{\mathcal{M}}{\vdash} acq_2\flat$

Si osservi, come al solito, la asimmetria delle condizioni di accettazione e di rifiuto di una stringa: nel primo caso, è sufficiente raggiungere una configurazione finale, nel secondo caso, è necessario che si giunga, nel corso della computazione, solo a configurazioni che non siano finali e sulle quali, al tempo stesso, non sia applicabile la funzione di transizione δ_{N}. Le implicazioni di tale asimmetria risulteranno più chiare nei Capitoli 8 e 9, nei quali considereremo il problema di caratterizzare il numero di transizioni richieste da macchine di Turing, deterministiche e non, per accettare o rifiutare stringhe. In quella sede verificheremo quali problemi insorgano per operare tale caratterizzazione nel caso di macchine non deterministiche.

Come annunciato precedentemente, non prendiamo in considerazione in questa sede l'utilizzazione di MTND come trasduttori, cioè come dispositivi per il calcolo di funzioni. Infatti, come si può comprendere tenendo conto del funzionamento di una macchina non deterministica, un trasduttore non deterministico può pervenire a differenti risultati operando diverse scelte nell'albero delle computazioni, dando così luogo al calcolo di funzioni multivalore, argomento che esula dagli scopi del presente volume.

5.5.2 Equivalenza tra MT ed MTND

Si è visto nella Sezione 5.4.3 che il tentativo di rendere le macchine di Turing più potenti tramite l'aggiunta di un numero arbitrario di nastri ha avuto l'esito di renderle più efficienti, ma non computazionalmente più potenti. Lo stesso vale per le macchine di Turing non deterministiche: benché queste possano essere notevolmenti più efficienti delle macchine di Turing deterministiche, esse non sono computazionalmente più potenti, poiché una macchina di Turing può simulare una macchina di Turing non deterministica.

La simulazione si ottiene facendo in modo che la macchina di Turing visiti tutti i nodi dell'albero delle computazioni. Come vedremo, la visita deve venir condotta in ampiezza e non in profondità, per evitare che la macchina di Turing, visitando un ramo infinito, non possa esaminare altri cammini nell'albero. Consideriamo per semplicità, ma senza perdere generalità, la simulazione di una macchina di Turing non deterministica $\mathcal{M}$ ad 1 nastro: per la simulazione si può usare una macchina di Turing deterministica $\mathcal{M}_{\mathrm{D}}$ con tre nastri, uno di input e due di lavoro, di cui il primo per memorizzare i possibili percorsi dalla radice fino ad un generico nodo e il secondo per la simulazione vera e propria (vedi Figura 5.14). Ad ogni transizione, $\mathcal{M}_{\mathrm{D}}$ controlla lo stato di $\mathcal{M}$ raggiunto: se esso è finale si ferma, altrimenti procede nella visita.

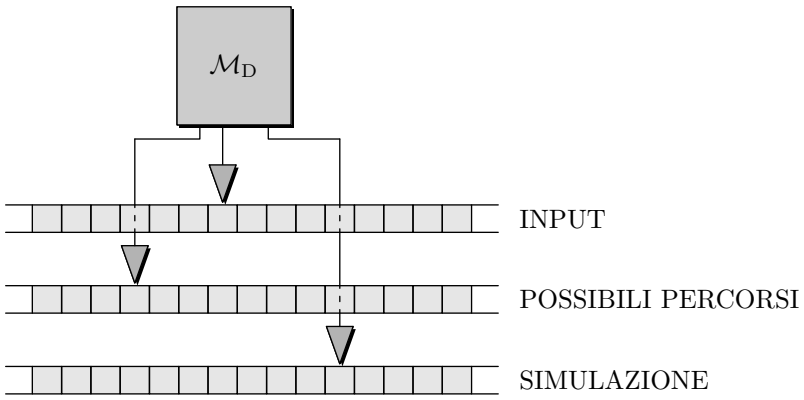


FIGURA 5.14 Macchina di Turing deterministica che simula una macchina di Turing non deterministica

---

Teorema 5.2 *Per ogni macchina di Turing non deterministica $\mathcal{M}$ esiste una macchina di Turing deterministica $\mathcal{M}_{\mathrm{D}}$ a 3 nastri equivalente.*

Dimostrazione. Esaminiamo l'uso dei 3 nastri di $\mathcal{M}_{\mathrm{D}}$. Nel primo nastro di $\mathcal{M}_{\mathrm{D}}$ viene copiata la configurazione iniziale di $\mathcal{M}$.

Il secondo nastro di $\mathcal{M}_{\mathrm{D}}$ viene usato in modo più articolato. In generale, per ogni coppia stato-simbolo di $\mathcal{M}$ esiste un certo numero, limitato da $d = \nu(\mathcal{M})$, di scelte per il passo successivo e quindi di configurazioni deterministiche successive.

Per ogni $i > 0$, un cammino di computazione di lunghezza i si può quindi individuare mediante una sequenza di i interi compresi tra 1 ed d, ciascuno dei quali determina la transizione da scegliere tra le al più d alternative possibili ad ogni passo.[5] Sul secondo nastro vengono allora generate, in ordine lessicografico, tutte le stringhe di lunghezza via via crescente — prima la stringa di lunghezza zero, poi tutte quelle di lunghezza uno, poi tutte quelle di lunghezza due, ecc. — composte di interi compresi tra 1 ed d. Ogni possibile computazione di $\mathcal{M}$ è rappresentata da una di tali stringhe.

Si osservi che, in generale, non tutte queste stringhe rappresentano cammini di computazione: ciò è determinato dal fatto che non è detto che per ogni configurazione deterministica attraversata siano possibili tutte le d scelte per il passo di computazione successivo.

Sul terzo nastro di $\mathcal{M}_{\mathrm{D}}$ viene eseguita la simulazione vera e propria, ed inizialmente esso contiene solo simboli $\flat$.

La simulazione procede per fasi, una fase per ogni sequenza generata sul nastro 2. Assumendo che $i_1, i_2, \ldots, i_k$ sia la sequenza contenuta su tale nastro, vengono eseguite le seguenti azioni:

1. l'input x è copiato dal nastro 1 al nastro 3;
2. si pone $j = 1$;
3. se $|\,\delta_{\mathrm{N}}(q, a)\,| < i_j$ la fase termina con esito negativo;
4. altrimenti, detta (q', a', r) la i_j-esima terna (nell'ordinamento arbitrario prefissato) in $\delta_{\mathrm{N}}(q, a)$, si applica la corrispondente transizione sul nastro 3;
5. si incrementa j di 1;
6. se $j < k$, si torna al passo 2;
7. se $j = k$ e si è raggiunta una configurazione finale, si termina con esito positivo, altrimenti la fase termina con esito negativo.

Assumiamo che in $\mathcal{M}$ esista un cammino che porta ad una configurazione finale e sia l la sua lunghezza: esiste allora sicuramente una fase della simulazione che è associata a una sequenza, di lunghezza l, sul secondo nastro che individua tale cammino. La simulazione di questa sequenza porterà ad uno stato finale. Se, viceversa, un siffatto cammino non esiste, allora $\mathcal{M}_{\mathrm{D}}$ non potrà mai raggiungere uno stato finale.

[5] A tale scopo è sufficiente introdurre un ordinamento (arbitrario) sugli elementi di $Q \times \bar{\Gamma} \times \{d, s, i\}$.

Se la macchina non deterministica $\mathcal{M}$ accetta una stringa in $k \geq 0$ passi, allora $\mathcal{M}_{\mathrm{D}}$ esegue al più $\sum_{j=0}^{k} jd^j$ passi. Tenendo conto del fatto che

$$\sum_{j=0}^{k} d^j = \frac{d^{k+1} - 1}{d - 1}$$

e che, derivando entrambi i membri dell'eguaglianza rispetto a d, si ottiene

$$\sum_{j=1}^{k} jd^{j-1} = \frac{kd^{k+1} - (k+1)d^k + 1}{(d-1)^2}$$

abbiamo che $\sum_{j=0}^{k} jd^j = d\sum_{j=1}^{k} jd^{j-1} = O(kd^k)$. □

Come si vede, la simulazione fornita nel teorema precedente comporta un aumento esponenziale del tempo di accettazione di una stringa rispetto al tempo richiesto dalla macchina non deterministica. Attualmente non è noto se esista una simulazione significativamente più efficiente di quella presentata. Come vedremo nel Capitolo 9, ci sono valide ragioni per ritenere che non sia possibile simulare una macchina di Turing non deterministica tramite una macchina di Turing deterministica con un numero di passi sub-esponenziale. Tuttavia ciò non è mai stato dimostrato, e la verità o meno di tale congettura rappresenta un importante problema aperto nell'informatica teorica.

La dimostrazione del Teorema 5.2 ci permette di introdurre una ulteriore interpretazione del non determinismo. Da tale dimostrazione possiamo osservare che una macchina di Turing deterministica è in grado di simulare efficientemente una macchina di Turing non deterministica se è in possesso di una informazione aggiuntiva (la stringa di caratteri in $\{1, \ldots, d\}$) che la "guida verso una configurazione di accettazione, se tale configurazione esiste.

Se L è il linguaggio accettato da una macchina di Turing non deterministica $\mathcal{M}$ allora, per ogni stringa $x \in L$, una sequenza di scelte $c \in \{1, \ldots, d\}^*$ che conduca $\mathcal{M}$ con input x ad una configurazione di accettazione può essere vista come una "prova" dell'appartenenza di x ad L.

Tutto ciò ci permette di concludere che, mentre per determinare se una stringa x verifica una determinata proprietà (cioè se $x \in L$) appare necessario cercare l'esistenza di una opportuna prova c, data una stringa c, la verifica che essa costituisce una prova dell'appartenenza di x a L può essere effettuata in tempo sensibilmente inferiore. Una discussione più approfondita di questi aspetti sarà svolta nel Capitolo 9.

Esercizio 5.14 Formalizzare il concetto di macchina di Turing multinastro non deterministica e dimostrare che le macchine di Turing multinastro non deterministiche sono equivalenti alle macchine di Turing multinastro.

5.6 Riduzione delle Macchine di Turing

Dopo aver visto che macchine di Turing deterministiche con un solo nastro possono simulare macchine non deterministiche e/o a più nastri, apparentemente più potenti, osserviamo che, addirittura, le macchine di Turing possono essere semplificate ulteriormente senza che si perda nulla in potere computazionale.

Una prima semplificazione si ottiene limitando le caratteristiche del nastro di una macchina di Turing: non più infinito in entrambe le direzioni bensì finito in una direzione ed infinito nell'altra. Si parla in questo caso di macchina di Turing con nastro *semi-infinito*.

Teorema 5.3 *Per ogni macchina di Turing* $\mathcal{M} = \langle \Gamma, \bar{b}, Q, q_0, F, \delta \rangle$ *esiste una macchina di Turing* $\mathcal{M}'$ *equivalente con nastro semi-infinito.*

Dimostrazione. Sia $\mathcal{M}' = \langle \Gamma', \bar{b}, Q', q_0', F', \delta' \rangle$. Si assuma un determinato punto sul nastro di $\mathcal{M}$ come origine e, di conseguenza, si consideri il nastro come suddiviso in due nastri semi-infiniti, comprendenti l'uno tutte le celle a destra e l'altro tutte le celle a sinistra dell'origine. Questa sarà scelta in modo che, nella configurazione iniziale, l'input x sia contenuto nelle prime $|\,x\,|$ celle del nastro di destra, mentre l'altro nastro è vuoto.

Il nastro di $\mathcal{M}'$ viene organizzato in modo da rappresentare i due nastri in cui abbiamo suddiviso quello di $\mathcal{M}$ e la posizione della testina. A tal fine, consideriamo il nastro di $\mathcal{M}'$ diviso in tre tracce di cui la superiore rappresenta il seminastro sinistro di $\mathcal{M}$, la inferiore il seminastro destro e la centrale la posizione della testina (vedi Figura 5.15). La prima cella del nastro di $\mathcal{M}'$ contiene un simbolo speciale che segnala l'inizio del nastro, ad esempio $\bar{b}$ in corrispondenza della tracce superiore e inferiore e $*$ in corrispondenza di quella centrale. Inoltre sulla traccia centrale tutte le altre celle conterranno il simbolo $\bar{b}$, eccetto che in corrispondenza della cella osservata dalla testina di $\mathcal{M}$, dove sarà presente il carattere $\uparrow$. All'inizio della computazione la traccia inferiore contiene l'input di $\mathcal{M}$, quella superiore solo una sequenza di $\bar{b}$, mentre quella centrale contiene tutti $\bar{b}$ fatta eccezione per la prima cella, ove è presente il simbolo $*$, e per la seconda cella, ove è presente il simbolo $\uparrow$.

L'insieme degli stati Q' contiene due stati $q_i^{\uparrow}$ e $q_i^{\downarrow}$ per ciascuno stato $q_i \in Q$. Se $\mathcal{M}$ a un certo istante si trova nello stato q_i allora $\mathcal{M}'$ durante la simulazione di $\mathcal{M}$ si troverà nello stato $q_i^{\uparrow}$ o in $q_i^{\downarrow}$, a seconda che il carattere osservato in $\mathcal{M}$ sia rappresentato, rispettivamente, sulla traccia superiore o inferiore del nastro di $\mathcal{M}'$. L'insieme Q' contiene anche $2\,|\,Q\,|^2$ stati del tipo $q_{i,j}^{\uparrow}$ e $q_{i,j}^{\downarrow}$ in cui $\mathcal{M}'$ transisce provvisoriamente durante la simulazione della transizione di $\mathcal{M}$ da q_i a q_j, che $\mathcal{M}'$ deve effettuare in due passi. In base a quanto detto, per lo stato iniziale si ha $q_0' = q_0^{\downarrow}$.

Per ogni regola $\delta(q_h, a_i) = (q_j, a_k, d)$ in $\mathcal{M}$, in $\mathcal{M}'$ sono definite le

transizioni:

$$\delta'\left(q_h^{\uparrow}, \begin{bmatrix} a_i \\ \uparrow \\ c \end{bmatrix}\right) = \left(q_{h,j}^{\uparrow}, \begin{bmatrix} a_k \\ \text{ƀ} \\ c \end{bmatrix}, s\right) \quad \forall c \in \bar{\Gamma}$$

$$\delta'\left(q_{h,j}^{\uparrow}, \begin{bmatrix} c' \\ \text{ƀ} \\ c'' \end{bmatrix}\right) = \left(q_j^{\uparrow}, \begin{bmatrix} c' \\ \uparrow \\ c'' \end{bmatrix}, i\right) \quad \forall c', c'' \in \bar{\Gamma},$$

che simulano la transizione di $\mathcal{M}$ nel caso in cui a si trovi sulla traccia superiore,

$$\delta'\left(q_h^{\downarrow}, \begin{bmatrix} c \\ \uparrow \\ a_i \end{bmatrix}\right) = \left(q_{h,j}^{\downarrow}, \begin{bmatrix} c \\ \text{ƀ} \\ a_k \end{bmatrix}, d\right) \quad \forall c \in \bar{\Gamma}$$

$$\delta'\left(q_{h,j}^{\downarrow}, \begin{bmatrix} c' \\ \text{ƀ} \\ c'' \end{bmatrix}\right) = \left(q_j^{\downarrow}, \begin{bmatrix} c' \\ \uparrow \\ c'' \end{bmatrix}, i\right) \quad \forall c', c'' \in \bar{\Gamma},$$

che simulano la transizione di $\mathcal{M}$ nel caso in cui a si trovi sulla traccia inferiore,

$$\delta'\left(q_{h,j}^{\uparrow}, \begin{bmatrix} \text{ƀ} \\ * \\ \text{ƀ} \end{bmatrix}\right) = \left(q_j^{\downarrow}, \begin{bmatrix} \text{ƀ} \\ \uparrow \\ \text{ƀ} \end{bmatrix}, d\right)$$

$$\delta'\left(q_{h,j}^{\downarrow}, \begin{bmatrix} \text{ƀ} \\ * \\ \text{ƀ} \end{bmatrix}\right) = \left(q_j^{\uparrow}, \begin{bmatrix} \text{ƀ} \\ \uparrow \\ \text{ƀ} \end{bmatrix}, d\right),$$

che simulano transizioni di $\mathcal{M}$ che portano la testina da una cella rappresentata sulla traccia superiore di $\mathcal{M}'$ ad una cella rappresentata sulla traccia inferiore (o viceversa).

Analoghe transizioni vanno introdotte in corrispondenza a regole del tipo $\delta(q_h, a_i) = (q_j, a_k, s)$.

Nel caso invece di regole del tipo $\delta(q_h, a_i) = (q_j, a_k, i)$, in $\mathcal{M}'$ sono definite le transizioni:

$$\delta'\left(q_h^{\uparrow}, \begin{bmatrix} a_i \\ \uparrow \\ c \end{bmatrix}\right) = \left(q_j^{\uparrow}, \begin{bmatrix} a_k \\ \uparrow \\ c \end{bmatrix}, i\right) \quad \forall c \in \bar{\Gamma}$$

$$\delta'\left(q_h^{\downarrow}, \begin{bmatrix} c \\ \uparrow \\ a_i \end{bmatrix}\right) = \left(q_j^{\downarrow}, \begin{bmatrix} c \\ \uparrow \\ a_k \end{bmatrix}, i\right) \quad \forall c \in \bar{\Gamma}$$

L'insieme degli stati finali di $\mathcal{M}'$, infine, è definito come $F' = \{q_i^{\uparrow}, q_i^{\downarrow} \mid q_i \in F\}$. □

Esercizio 5.15 Dimostrare, per induzione sulla lunghezza della computazione, che $\mathcal{M}'$ simula correttamente $\mathcal{M}$.

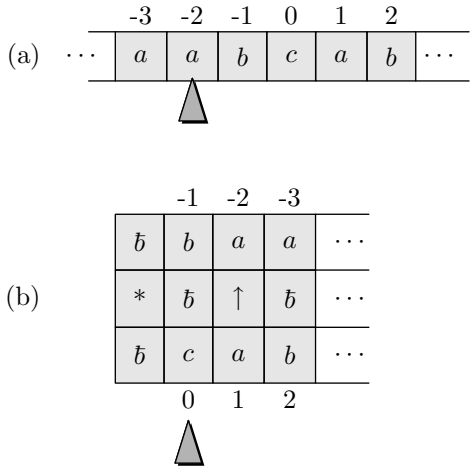


FIGURA 5.15 (a) Nastro di una macchina di Turing e (b) nastro semi- infinito a tre tracce usato per la sua simulazione.

Una seconda versione di macchina di Turing semplificata si ottiene riducendo la dimensione dell'alfabeto. Tenendo conto della possibilità di codificare qualunque carattere di un alfabeto finito utilizzando due soli simboli, è semplice mostrare che è possibile ridurre la cardinalità dell'alfabeto Γ a 1 e, quindi, la cardinalità di $|\bar{\Gamma}|$ a 2. Data una macchina $\mathcal{M} = \langle \Gamma, ƀ, Q, q_0, F, \delta \rangle$, si può infatti costruire la macchina equivalente $\mathcal{M}' = \langle \Gamma', ƀ, Q', q_0', F', \delta' \rangle$, con $|\Gamma'| = 1$, definendo una opportuna codifica dei caratteri di $\bar{\Gamma}$ tramite quelli di $\bar{\Gamma}'$. Ad esempio, possiamo codificare l'insieme $\bar{\Gamma} = \{a, b, c, d, ƀ\}$ tramite $\bar{\Gamma}' = \{1, ƀ\}$ come illustrato in Tabella 5.2.

Inoltre, per ogni regola di transizione della macchina $\mathcal{M}$, prevediamo una serie di regole di transizione della macchina $\mathcal{M}'$ che svolgono le seguenti azioni:

- determinazione del carattere originario osservato;
- eventuale modifica di tale carattere;
- eventuale spostamento della testina in corrispondenza della codifica del carattere più a destra o più a sinistra di quello osservato correntemente.

$\flat$	=	$\flat\flat\flat$
a	=	$\flat\flat 1$
b	=	$\flat 1\flat$
c	=	$\flat 11$
d	=	$1\flat\flat$

Tabella 5.2 Esempio di codifica per ridurre la cardinalità dell'alfabeto di una macchina di Turing.

Ad esempio, assumendo sempre la codifica vista in Tabella 5.2, la regola $\delta(q_i, a) = (q_k, b, d)$ della macchina $\mathcal{M}$ dà luogo, nella macchina $\mathcal{M}'$, alle seguenti regole:

$$\begin{aligned}
\delta'(q_i, \flat) &= (q_{i,\flat}, \flat, d) \\
\delta'(q_{i,\flat}, \flat) &= (q_{i,\flat\flat}, \flat, d) \\
\delta'(q_{i,\flat\flat}, 1) &= (q_{i,\flat\flat 1}, \flat, s) \\
\delta'(q_{i,\flat\flat 1}, \flat) &= \left(q'_{i,\flat\flat 1}, \flat, s\right) \\
\delta'\left(q'_{i,\flat\flat 1}, \flat\right) &= \left(q''_{i,\flat\flat 1}, \flat, d\right) \\
\delta'\left(q''_{i,\flat\flat 1}, 1\right) &= \left(q'''_{i,\flat\flat 1}, 1, d\right) \\
\delta'\left(q'''_{i,\flat\flat 1}, \flat\right) &= (q_k, \flat, d)\,,
\end{aligned}$$

dove il "significato" degli stati utilizzati è il seguente:

1. $\mathcal{M}'$ in q_i: $\mathcal{M}$ in q_i con la testina di$\mathcal{M}'$ sul primo carattere della codifica del carattere corrente di $\mathcal{M}$;
2. $\mathcal{M}'$ in $q_{i,\flat}$: come sopra, ma la testina di $\mathcal{M}'$, dopo aver letto il simbolo $\flat$, si trova sul secondo carattere della codifica del carattere corrente di $\mathcal{M}$;
3. $\mathcal{M}'$ in $q_{i,\flat\flat}$: come sopra, ma la testina di $\mathcal{M}'$, dopo aver letto la coppia di simboli $\flat\flat$, si trova sul terzo ed ultimo carattere della codifica del carattere corrente di $\mathcal{M}$;
4. $\mathcal{M}'$ in $q_{i,\flat\flat 1}$: come sopra, ma la testina di $\mathcal{M}'$, dopo aver letto i tre simboli $\flat\flat 1$, ha aggiornato il terzo carattere della codifica del carattere corrente di $\mathcal{M}$;

5. $\mathcal{M}'$ in $q'_{i,\text{ƀƀ1}}$: come sopra, ma la testina di $\mathcal{M}'$, dopo aver letto i tre simboli ƀƀ1, ha aggiornato il secondo carattere della codifica del carattere corrente di $\mathcal{M}$;

6. $\mathcal{M}'$ in $q''_{i,\text{ƀƀ1}}$: come sopra, ma la testina di $\mathcal{M}'$, dopo aver letto i tre simboli ƀƀ1, ha aggiornato tutti i caratteri della codifica del carattere corrente di $\mathcal{M}$;

7. $\mathcal{M}'$ in $q'''_{i,\text{ƀƀ1}}$: come sopra, si è aggiornato il contenuto del nastro e si deve spostare la testina una cella a destra;

8. $\mathcal{M}'$ in q_k: la transizione è stata simulata, $\mathcal{M}$ si trova in q_i con la testina di$\mathcal{M}'$ sul primo carattere della codifica del carattere corrente di $\mathcal{M}$.

È possibile dunque giungere alla dimostrazione del seguente teorema.

Teorema 5.4 *Per ogni macchina di Turing* $\mathcal{M} = \langle \Gamma, ƀ, Q, q_0, F, \delta \rangle$ *esiste una macchina di Turing* $\mathcal{M}' = \langle \Gamma', ƀ, Q', q'_0, F', \delta' \rangle$*, con* $\mid \Gamma' \mid = 1$*, equivalente a* $\mathcal{M}$*.*

Esercizio 5.16 Completare nei dettagli la dimostrazione precedente.

Come si è visto in questo risultato, la riduzione del numero dei caratteri comporta un aumento del numero di stati. Non è difficile verificare che una relazione di questo genere si presenta frequentemente nelle macchine a stati. Una situazione analoga si è presentata nell'ambito dello studio degli automi a pila, quando si è visto (vedi Teorema 4.17) che è possibile realizzare, per ogni linguaggio CF, un automa a pila con un solo stato, a patto di aumentare considerevolmente il numero di simboli di pila utilizzati.

5.7 Descrizione linearizzata delle Macchine di Turing

Nella trattazione delle macchine di Turing svolta fino a questo punto, non abbiamo mai sfruttato la possibilità di creare macchine più complesse a partire da macchine più elementari già definite. Questa possibilità, non molto dissimile da quella di realizzare automi a stati finiti più complessi tramite l'operazione di concatenazione di automi più semplici, si rivela utile nell'analisi delle proprietà delle MT che vedremo nel resto di questo capitolo.

Il concetto che è alla base dell'uso di macchine più elementari nella sintesi di nuove macchine è quello di composizione per diramazione condizionata. Essa consiste nel far subentrare ad una macchina arrestatasi in un stato finale altre macchine, a seconda del carattere osservato nello stato finale dalla prima macchina.

Supponiamo di avere a disposizione le seguenti tre macchine: la prima, $\mathcal{M}$ (vedi Figura 5.16), che fa passare dalla configurazione $q_0 x a_i$ alla configurazione $x q_F a_i$ con $a_i \in \Gamma = \{0, 1\}$ e $x \in \Gamma^*$.

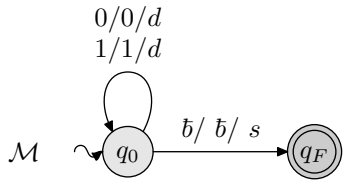


FIGURA 5.16 Macchina $\mathcal{M}$.

La seconda, $\mathcal{M}_1$ (vedi Figura 5.17), che semplicemente scrive il carattere 1.

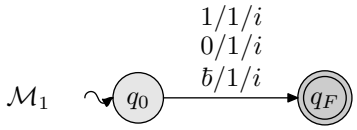


FIGURA 5.17 Macchina $\mathcal{M}_1$.

La terza, $\mathcal{M}_2$ (vedi Figura 5.18), che semplicemente scrive il carattere 0.

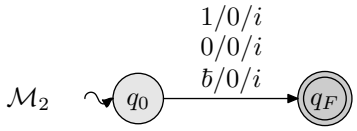
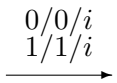


FIGURA 5.18 Macchina $\mathcal{M}_2$.

Orbene, se vogliamo far subentrare la macchina $\mathcal{M}_1$ quando la macchina $\mathcal{M}$ si arresta, non abbiamo da fare altro che collegare lo stato finale di $\mathcal{M}$ con lo stato iniziale di $\mathcal{M}_1$ mediante la transizione



Se invece vogliamo collegare a scelta $\mathcal{M}_1$ o $\mathcal{M}_2$ a seconda che il carattere a_i sia 0 o 1, possiamo comporre tali macchine costruendo il diagramma in Figura 5.19

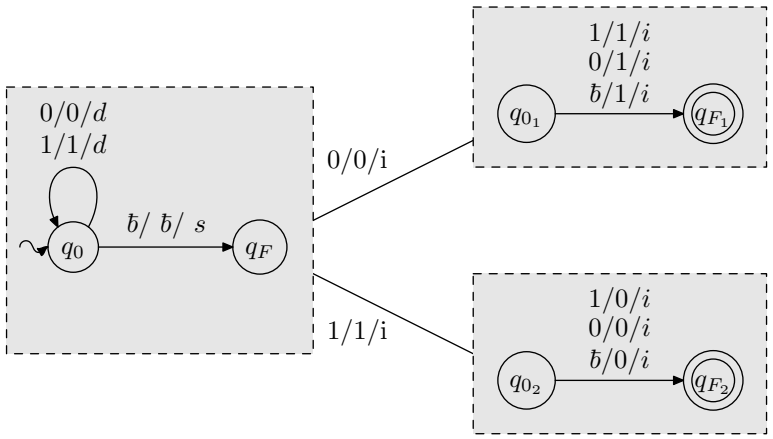
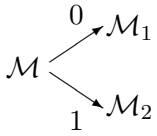


FIGURA 5.19 Composizione delle macchine $\mathcal{M}$, $\mathcal{M}_1$, $\mathcal{M}_2$.

La composizione di $\mathcal{M}$, $\mathcal{M}_1$, $\mathcal{M}_2$ è rappresentabile anche come



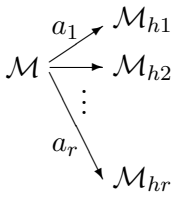
Gli stati finali della macchina così ottenuta sono q_{F1} e q_{F2}, mentre q_0 è lo stato iniziale.

Questa nozione intuitiva di composizione di macchine può essere formalizzata come segue.

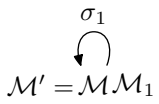
Definizione 5.20 *Siano* $\mathcal{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *e* $\mathcal{M}_i = \langle \Gamma_i, \flat, Q_i, q_{0i}, F_i, \delta_i \rangle$, $i \in \{1, \ldots, n\}$, $1 \leq i \leq n$ *macchine di Turing tali che* $\Gamma = \Gamma_1 = \cdots = \Gamma_n = \{a_1, \ldots, a_r\}$ *e* $Q \cap Q_h = Q_i \cap Q_j = \emptyset$, *per* $1 \leq h \leq n$ *e* $1 \leq i < j \leq n$.[6] *Diremo che* $\mathcal{M}' = \langle \Gamma', \flat, Q', q_0', F', \delta' \rangle$ *è ottenuta tramite* composizione per diramazione condizionata *da* $\mathcal{M}$, $\mathcal{M}_1$, ..., $\mathcal{M}_n$ *se valgono le seguenti proprietà:* $\Gamma' = \Gamma$, $Q' = Q \cup Q_1 \cup \cdots \cup Q_n$, $q_0' = q_0$, $F' = F_1 \cup \cdots \cup F_n$, $\delta'(q, a) = \delta(q, a)$ $\forall q \in Q - F$, $\delta'(q, a) = \delta_i(q, a)$ $\forall q \in Q_i, 1 \leq i \leq n$, $\delta'(q, a_j) = (q_{0h_j}, a_j, i)$, $1 \leq h_j \leq n, 1 \leq j \leq r$, $\forall q \in F$.

La macchina $\mathcal{M}'$ può essere così rappresentata:

[6] Se tali proprietà non valgono possono essere semplicemente "imposte" poiché se gli stati interni non sono disgiunti, essi possono essere ridenominati e se gli alfabeti non sono coincidenti possiamo estenderli banalmente al minimo alfabeto che li include tutti.

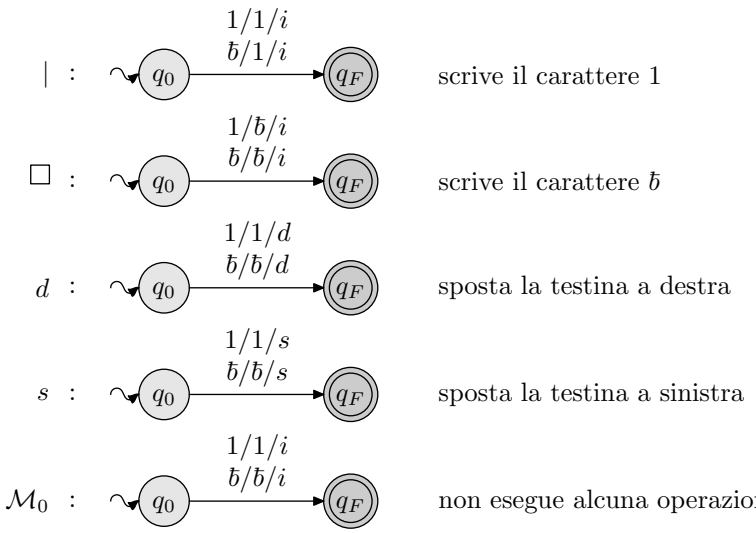


È chiaro che può accadere che alcune delle macchine $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_r$ coincidano fra loro. Inoltre, nel caso particolare $n = 1$ abbiamo $\mathcal{M}' = \mathcal{M}\mathcal{M}_1$, ove l'indicazione della diramazione condizionata scompare. Un altro caso particolare interessante è quando una delle macchine verso cui $\mathcal{M}$ si dirama coincide con $\mathcal{M}$ stessa e tutte le altre coincidono fra loro. In tal caso, la rappresentazione grafica diventa:

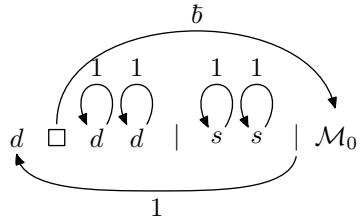


In quanto segue vogliamo mostrare come alcune macchine elementari, composte per diramazione, permettano di sintetizzare tutte le macchine di Turing ad un nastro e due caratteri. Poiché sappiamo che le macchine di Turing ad un nastro e due caratteri possono simulare macchine di Turing di tipo più generale, ne consegue la possibilità di esprimere qualunque macchina di Turing come un diagramma in cui figurano solo macchine elementari.

Definizione 5.21 *Chiamiamo* macchine di Turing elementari *le macchine* $|$, $\square$, d, s, $\mathcal{M}_0$ *definite sull'alfabeto* $\Gamma = \{1\}$ *nel modo seguente:*



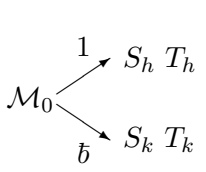
Esempio 5.7 Sintetizziamo con macchine elementari la macchina $\mathcal{C}$ che realizza la computazione[7] $q_0 ß x ß \overset{*}{\vdash\!\!\!-} ß x q_F ß x$, dove $x \in \{1\}^*$.



Teorema 5.5 *Ogni macchina di Turing è rappresentabile come composizione per diramazione delle macchine elementari e, viceversa, ogni diagramma di composizione di macchine elementari è una rappresentazione di una macchina di Turing.*

Dimostrazione. La seconda parte è del tutto ovvia poiché, come abbiamo visto, componendo diagrammi di stato di macchine di Turing, si ottengono ancora diagrammi di stato di macchine di Turing. Per dimostrare la prima parte osserviamo che ad ogni stato non finale di una macchina di Turing possiamo associare un piccolo diagramma che utilizza le macchine elementari. Sia q_i un generico stato e siano $\delta(q_i, 1) = (q_h, a_h, t_h)$ e $\delta(q_i, ß) = (q_k, a_k, t_k)$ dove $a_h, a_k \in \{1, ß\}$ e $t_h, t_k \in \{s, d, i\}$ i relativi valori della funzione di transizione.

A tale stato possiamo far corrispondere il diagramma:



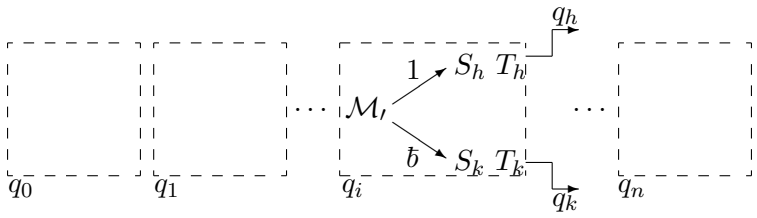
$$\text{dove} \quad S_\ell = \begin{cases} | & \text{se } a_\ell = 1 \\ \square & \text{se } a_\ell = ß \end{cases}$$

$$\text{e} \quad T_\ell = \begin{cases} d & \text{se } t_\ell = d \\ s & \text{se } t_\ell = s \\ \mathcal{M}_0 & \text{se } t_\ell = i \end{cases}$$

ed $\ell = h$ o, rispettivamente, $\ell = k$. Il diagramma corrispondente allo stato q_i dovrà poi collegarsi ai diagrammi corrispondenti agli stati q_h e q_k. Ripetendo per ogni stato lo stesso ragionamento e connettendo i diagrammi otteniamo lo schema generale:

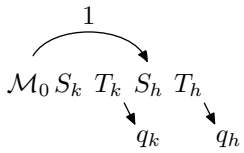
[7] Si noti che, per evitare di indicare esplicitamente gli stati delle macchine ottenute per composizione di macchine elementari, nel rappresentare le configurazioni iniziali e finali di una computazione ci limiteremo ad indicare con una sottolineatura il simbolo osservato dalla testina. Scriveremo dunque:

$$x\underline{a_i}y \overset{*}{\vdash\!\!\!-} u\underline{a_j}v \qquad \text{anziché :} \qquad xq_0a_iy \overset{*}{\vdash\!\!\!-} uq_Fa_jv.$$



Per quanto riguarda gli stati finali, ad essi potrà essere associata semplicemente la macchina $\mathcal{M}_0$. □

Si noti che dalla dimostrazione segue che per realizzare una macchina con n stati, occorrono al più $5n$ macchine elementari. Inoltre, se ogni stato non finale viene rappresentato nella forma linearizzata:

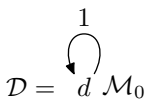


si vede intuitivamente che ogni macchina di Turing può essere rappresentata da un *diagramma linearizzato* in cui le connessioni sono stabilite mediante salti condizionati o incondizionati.

L'uso dei diagrammi linearizzati per le macchine di Turing ci consente di individuare con un identificatore $\mathcal{X}$ una determinata macchina e di usare tale identificatore nella sintesi di macchine più complesse, in luogo del diagramma linearizzato della stessa macchina $\mathcal{X}$. Si noti che, in questo contesto, definiremo il comportamento delle macchine di Turing che introdurremo semplicemente fornendo la configurazione iniziale e quella finale. Nel caso che le macchine venissero avviate su una configurazione iniziale differente, il loro comportamento non sarebbe predeterminato.

Esempio 5.8

i) macchina $\mathcal{D}$: cerca il primo spazio a destra.

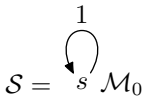


La macchina $\mathcal{D}$ realizza la seguente computazione:

$$\sim q_0 1 \ \ldots \ 1 \ \flat \ \overset{*}{\vdash\!\!-} \ \sim \ 1 \ \ldots \ 1 \ q_F \flat$$

(il simbolo $\sim$ indica un carattere che è indifferentemente 1 o $\flat$)

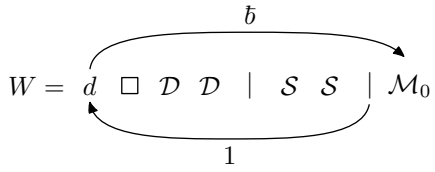
ii) macchina $\mathcal{S}$: cerca il primo spazio a sinistra.



La macchina $\mathcal{S}$ realizza la seguente computazione:

$$\text{ƀ } 1 \ \ldots \ 1 \ q_0 \sim \overset{*}{\vdash\!\!-} \ \text{ƀ} q_F 1 \ \ldots \ 1 \ \sim$$

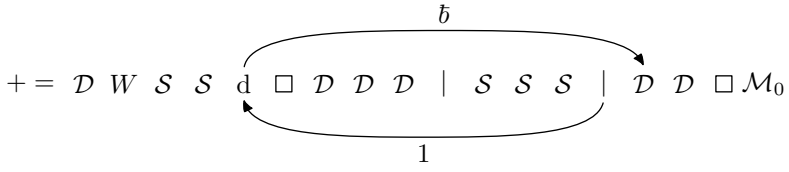
iii) macchina W: copia una stringa appartenente ad 1^*.



La macchina $\mathcal{W}$ realizza la seguente computazione:

$$q_0 \text{ƀ} \overbrace{1 \ \ldots \ 1}^{n} \ \text{ƀ} \overset{*}{\vdash\!\!-} \ \text{ƀ} \overbrace{1 \ \ldots \ 1}^{n} \ q_F \text{ƀ} \overbrace{1 \ \ldots \ 1}^{n}$$

iv) macchina $+$: somma due interi rappresentati in notazione unaria.



La macchina $+$ realizza la seguente computazione:

$$q_0 \text{ƀ} \underbrace{1 \ \ldots \ 1}_{n+1} \ \text{ƀ} \ \underbrace{1 \ \ldots \ 1}_{m+1} \ \text{ƀ} \overset{*}{\vdash\!\!-} \ \text{ƀ} \ \underbrace{1 \ \ldots \ 1}_{n+1} \ \text{ƀ} \ \underbrace{1 \ \ldots \ 1}_{m+1} \ \text{ƀ} \ \underbrace{1 \ \ldots \ 1}_{m+n+1} \ q_F \text{ƀ}$$

Esercizio 5.17 Dare la descrizione linearizzata delle macchine che calcolano la sottrazione e il prodotto. Si adottino le medesime convenzioni per la configurazione iniziale e per la configurazione finale adottate nell'Esempio 5.8 iv).

5.8 La macchina di Turing universale

Nelle pagine precedenti abbiamo introdotto la macchina di Turing come un dispositivo "astratto", dotato di un meccanismo estremamente elementare e abbiamo definito i concetti di riconoscimento e accettazione di linguaggi e di calcolo di funzioni mediante macchine di Turing. Da questa sezione in poi cercheremo di caratterizzare la potenza delle macchine di Turing individuando prima una funzione estremamente potente che è calcolabile secondo Turing e successivamente una funzione non calcolabile secondo Turing. Infine dimostreremo che i linguaggi di tipo 0 coincidono con i linguaggi accettati dalle macchine di Turing.

Innanzitutto ricordiamo il concetto di calcolo di una funzione da parte di un trasduttore fornito nella Definizione 5.8 ed estendiamo tale concetto a funzioni di più argomenti. Sia $m : (\Sigma^*)^n \mapsto \Sigma^*$. Diciamo che la macchina $\mathcal{M}$ calcola la funzione m se essa realizza la computazione $q_0 x_1 \text{ƀ} \dots \text{ƀ} x_n \overset{*}{\underset{\mathcal{M}}{\vdash}} x_1 \text{ƀ} \dots \text{ƀ} x_n \text{ƀ} q y$ con q stato finale se e solo se $m(x_1, \dots, x_n) = y$.

Definizione 5.22 *Una macchina di Turing* $\mathcal{U} = \langle \Gamma, \text{ƀ}, Q', \delta', q'_0, F' \rangle$ *si dice* macchina di Turing universale *se essa calcola una funzione* $u : (\Gamma^*)^(n+1) \mapsto \Gamma^*$ *con la seguente proprietà: data una qualunque macchina di Turing* $\mathcal{M} = \langle \Gamma, \text{ƀ}, Q, \delta, q_0, F \rangle$ *che calcola la funzione* $m : (\Gamma^*)^n \mapsto \Gamma^*$ *esiste una stringa* $c_{\mathcal{M}} \in \Gamma^*$ *(codificazione di* $\mathcal{M}$*) tale che*

$$u(c_{\mathcal{M}}, x_1, \dots, x_n) = m(x_1, \dots, x_n).$$

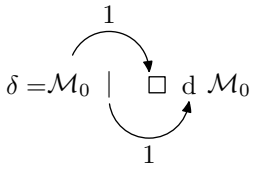
Una macchina di Turing universale è dunque in grado di simulare ogni altra macchina di Turing. Al fine di dimostrare l'esistenza di una macchina di Turing universale, nonché fornirne una descrizione, procederemo nel seguente modo:

1. Innanzi tutto prenderemo in considerazione MT con nastro semi-infinito e alfabeto costituito dagli unici due simboli 1 e ƀ. Ciò non costituisce una perdita di generalità in quanto ad ogni MT possiamo dapprima applicare il risultato del Teorema 5.3 che consente di ricavare una macchina equivalente con nastro semi-infinito e quindi applicare a quest'ultima il risultato del Teorema 5.4 che permette di ridurre l'alfabeto ad un solo simbolo.

2. In secondo luogo proveremo che è possibile fornire una descrizione linearizzata di ogni MT del tipo suddetto utilizzando tre soli tipi di MT elementari e composizione per diramazione sul simbolo 1. A tal fine, introdurremo una nuova macchina δ e mostreremo nel successivo Lemma 5.6 come ognuna delle macchine elementari introdotte nella sezione precedente sia definibile come composizione delle macchine δ, s e $\mathcal{M}_\prime$, utilizzando la composizione per diramazione solo sul simbolo 1.

3. Infine, mostreremo come la descrizione linearizzata di una MT possa essere rappresentata essa stessa mediante una stringa sull'alfabeto $\{1, \text{ƀ}\}$ e come possa essere definita una macchina che, assumendo in input tale descrizione, simuli il comportamento della macchina descritta.

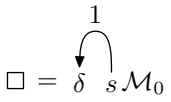
Lemma 5.6 *Indichiamo con δ la macchina che, qualunque sia il carattere letto, 1 o ƀ, lo inverte e sposta a destra la testina come di seguito illustrato:*



Si considerino inoltre le macchine elementari s ed $\mathcal{M}_\prime$. È possibile fornire una descrizione linearizzata delle macchine $|$, $\square$ e d che utilizza solo le macchine δ, s ed $\mathcal{M}_\prime$ e sfrutta la composizione per diramazione solo sul simbolo 1.

Dimostrazione. Le descrizioni linearizzate di d e $\square$ sono le seguenti.

$$d = \delta\ s\ \delta\, \mathcal{M}_0$$

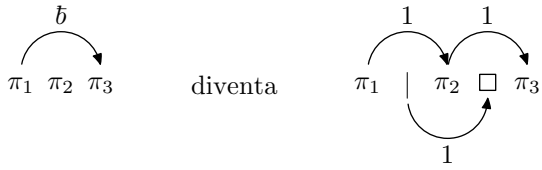


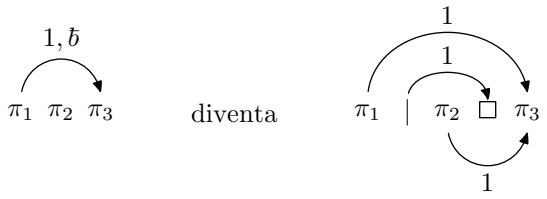
La descrizione linearizzata della macchina $|$ viene lasciata per esercizio. $\square$

Esercizio 5.18 Fornire la descrizione linearizzata della macchina $|$ utilizzando solo le macchine δ, s ed $\mathcal{M}_\prime$ e sfruttando la diramazione per composizione solo sul simbolo 1.

Per semplificare ulteriormente la descrizione linearizzata delle MT basata su macchine elementari possiamo agire nel seguente modo. Innanzi tutto conveniamo che la macchina $\mathcal{M}_\prime$ venga sempre eliminata: quando si trova tra due descrizioni può essere chiaramente soppressa; al termine di una descrizione linearizzata, si può assumere per convenzione che essa sia sempre presente. In tal modo le macchine elementari si riducono a δ ed s.

In secondo luogo, possiamo normalizzare i salti e ridurci ai salti condizionati su 1, nel seguente modo:





A seguito delle trasformazioni precedenti, la descrizione linearizzata di una qualunque MT $\mathcal{M}$ operante su simboli 1 e $ƀ$ consiste in una sequenza di simboli δ ed s intercalati da salti condizionati sul simbolo 1. Tale sequenza può essere assimilata ad un programma basato sulle sole "istruzioni" δ ed s e su salti condizionati sul simbolo 1.

Potremo ora rappresentare una tale sequenza come una parola $c_{\mathcal{M}}$ sull'alfabeto $\{1, ƀ\}$ procedendo come segue:

- Le "istruzioni" s e δ saranno rispettivamente codificate con le stringhe 1 e 11.
- Una "istruzione" di salto condizionato sul simbolo 1 alla "istruzione" n-esima sarà codificata mediante la stringa 1^{n+3}.
- Il codice di una istruzione è separato da $ƀ$ dal codice dell'istruzione successiva.
- La stringa 1^3 rappresenta la fine della sequenza di istruzioni.

Quanto abbiamo visto ci ha consentito di costruire il codice $c_{\mathcal{M}}$ di una data macchina $\mathcal{M}$ (operante su nastro semi-infinito) in termini di una stringa sull'alfabeto $\{1, ƀ\}$. La macchina universale assumerà in input il codice $c_{\mathcal{M}}$ di una macchina data seguito dalla sequenza $x_1, \ldots, x_n$ su cui la macchina $\mathcal{M}$ è destinata ad operare. Essa dovrà fornire come risultato la stringa $m(x_1, \ldots, x_n)$; per tale motivo, nella configurazione finale non dovrà più comparire il codice $c_{\mathcal{M}}$.

Per realizzare tale computazione la macchina universale $\mathcal{U}$ dovrà utilizzare alcuni caratteri speciali, ad esempio $1'$ e $ƀ'$, per segnare sul codice $c_{\mathcal{M}}$ quale istruzione della macchina $\mathcal{M}$ vada simulata e per indicare sulla stringa corrispondente al nastro della macchina $\mathcal{M}$ su quale carattere essa sia posizionata. Per ottenere l'arresto della computazione si conviene che esso sia determinato da un salto ad una istruzione non presente.

Le configurazioni della macchina universale si presentano dunque nel seguente modo:

- configurazione iniziale

$$\overbrace{\underline{1}\ \ldots\ 1\ ƀ\ 1\ \ldots\ 1\ ƀ\ \ldots\ ƀ\ 1\ \ldots\ 1\ ƀ\ 111}^{c_{\mathcal{M}}}\ ƀ\ \overbrace{\sim\qquad\qquad\sim}^{x_1,\ldots,x_n}$$

- configurazioni intermedie

$$1 \ \ldots \ 1 \ \text{ƀ} \ 1' \ \ldots \ 1 \ \text{ƀ} \ \ldots \ \text{ƀ} \ 1 \ \ldots \ \underline{1} \ \text{ƀ} \ 111 \ \text{ƀ} \ \sim \ \ldots \ \sim' \ \ldots \ \sim$$

- configurazione finale

$$\text{ƀ} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{ƀ} \ \text{ƀ} \ \underbrace{\sim \ \ldots \ \simeq \ \ldots \ \sim}_{m(x)} \text{ƀ} \ldots \text{ƀ}$$

Utilizzando la costruzione precedentemente esposta si perviene al seguente risultato.

Teorema 5.7 *Esiste una macchina di Turing* $\mathcal{U} = \langle \Gamma', \text{ƀ}, Q', \delta', q_0', F' \rangle$ *tale che data ogni macchina di Turing* $\mathcal{M} = \langle \{1\}, \text{ƀ}, Q, \delta, q_0, F \rangle$*:*

$$u(c_{\mathcal{M}}, x) = m(x) \quad x \in \{1, \text{ƀ}\}^*.$$

Dimostrazione. Per la dimostrazione completa del risultato si rende necessario definire il comportamento della macchina $\mathcal{U}$ che, scandendo via via le "istruzioni" presenti nel "programma" $c_{\mathcal{M}}$, esegue le operazioni corrispondenti sulla porzione di nastro che corrisponde al nastro della macchina $\mathcal{M}$. La definizione dettagliata del comportamento di $\mathcal{U}$ viene lasciata al lettore. □

Esercizio 5.19 Costruire la macchina $\mathcal{U}$.

Si noti che, per omogeneità di trattazione e per coerenza con il concetto di universalità si è chiesto che la descrizione della macchina $c_{\mathcal{M}}$ venga cancellata al termine dell'elaborazione. Tuttavia ciò non corrisponde a quanto accade nell'interpretazione di una macchina universale come un "interprete" di un linguaggio di programmazione. Tale interpretazione è, tuttavia, abbastanza suggestiva ed ha un preciso corrispondente nell'ambito dei linguaggi di programmazione (si consideri ad esempio la funzione EVAL nel LISP). In tal caso la distruzione del programma non solo non viene effettuata ma, anzi, viene accuratamente evitata.

Come ulteriore osservazione è opportuno notare che qualora la macchina $\mathcal{M}$ riceva in input una n-pla di argomenti la macchina universale potrà essere modificata in modo che calcoli la funzione:

$$u(c_{\mathcal{M}}, x_1, \ldots, x_n) = m(x_1, \ldots, x_n).$$

5.9 Il problema della terminazione

Il potere computazionale delle macchine di Turing è stato discusso nella Sezione 5.3, in cui sono stati introdotti i concetti di funzione T-calcolabile e di linguaggio T-decidibile. Al fine di illustrare i limiti del potere computazionale di una macchina di Turing mostriamo ora un esempio di funzione non T-calcolabile. Si consideri il problema decisionale seguente, noto come *problema della terminazione* (*halting problem*): date una macchina di Turing $\mathcal{M}$ e una stringa x, stabilire se $\mathcal{M}$ termina la computazione avendo x come input. Si noti che tale problema ha un notevole impatto applicativo. Dal punto di vista pratico infatti sarebbe di grande interesse disporre di un algoritmo in grado di decidere, dato un programma in un qualsiasi linguaggio di programmazione e dato un particolare input, se il programma termina su tale input. Il risultato che vedremo in questa sezione mostra invece che non essendo il problema risolubile con una macchina di Turing, esso non potrà essere risolto con alcun metodo algoritmico.

Teorema 5.8 *[Indecidibilità del problema della terminazione[8]] Siano dati un alfabeto* Γ *ed una codificazione che associa ad ogni macchina di Turing* $\mathcal{M} = \langle \Gamma, \flat, Q, \delta, q_0, F \rangle$ *una sua codifica* $c_{\mathcal{M}} \in \Gamma^*$*. La funzione*

$$h(c_{\mathcal{M}}, x) = \begin{cases} 1 & \textit{se } \mathcal{M} \textit{ termina su input } x \\ 0 & \textit{se } \mathcal{M} \textit{ non termina su input } x \end{cases}$$

non è T-calcolabile.

Dimostrazione. Supponiamo per assurdo che esista una macchina di Turing $\mathcal{H}$ capace di calcolare la funzione h. Possiamo allora costruire una macchina di Turing $\mathcal{H}'$ che calcola la funzione h' tale che:

$$h'(c_{\mathcal{M}}) = \begin{cases} 1 & \text{se } \mathcal{M} \text{ termina su input } c_{\mathcal{M}} \\ 0 & \text{se } \mathcal{M} \text{ non termina su input } c_{\mathcal{M}} \end{cases}$$

È facile osservare che la macchina $\mathcal{H}'$ è la composizione di una macchina $\mathcal{C}$ che duplica una stringa in input e della macchina $\mathcal{H}$ (vedi Esercizio 5.20).

A questo punto possiamo definire una nuova macchina $\mathcal{H}''$ risultante dalla composizione della macchina $\mathcal{H}'$ e di una macchina $\mathcal{E}$ che cicla se la macchina $\mathcal{H}'$ termina con la testina posizionata sul carattere 1 e si arresta senza effettuare alcuna operazione se la macchina $\mathcal{H}'$ termina con la testina posizionata sul carattere 0 (vedi Esercizio 5.20). La macchina $\mathcal{H}''$ pertanto calcola una funzione h'' tale che $h''(c_{\mathcal{M}}) = 0$ se e solo se la macchina $\mathcal{M}$ su input $c_{\mathcal{M}}$ non termina; se al contrario la macchina $\mathcal{M}$ su input $c_{\mathcal{M}}$ termina, allora la macchina $\mathcal{H}''$ non termina.

[8] Questo risultato è noto anche come *indecidibilità del predicato della terminazione.*

Verifichiamo ora che se forniamo in input alla macchina $\mathcal{H}''$ la propria descrizione $c_{\mathcal{H}''}$ otteniamo in ogni caso una contraddizione. Infatti la macchina $\mathcal{H}''$ con input $c_{\mathcal{H}''}$ dovrebbe arrestarsi proprio nel caso in cui la computazione che essa effettua non termina, e viceversa.

L'assurdo ottenuto implica che la macchina $\mathcal{H}''$ ottenuta componendo le macchine $\mathcal{C}$, $\mathcal{H}$ ed $\mathcal{E}$ non può esistere e da ciò si deriva immediatamente che la macchina $\mathcal{H}$ non può esistere e pertanto la funzione h non è T-calcolabile. □

Esercizio 5.20 Costruire le macchine $\mathcal{C}$ ed $\mathcal{E}$.

Come già accennato, l'indecidibilità del problema della terminazione ha delle importanti conseguenze nell'ambito della teoria della programmazione. In particolare, esso indica chiaramente che non è possibile costruire un perfetto sistema di "debugging" che sia in grado di determinare se un programma termini o meno sull'input dato.

Si noti che il problema della terminazione, pur non essendo decidibile, risulta semidecidibile.

Esercizio 5.21 Dimostrare la semidecidibilità del problema della terminazione.

Nel Capitolo 7 vedremo ulteriori esempi di funzioni non calcolabili e di problemi non decidibili ed approfondiremo ulteriormente questi concetti. Si ricordi inoltre che un primo esempio di problema indecidibile (l'ambiguità della grammatiche non contestuali) è già stato presentato nella Sezione 4.8.1.

5.10 Linguaggi di tipo 0 e MT

Dopo aver studiato le proprietà delle macchine di Turing mostriamo ora un risultato che è stato preannunciato nella Sezione 2.5.3, e cioè che le macchine di Turing hanno un potere computazionale necessario e sufficiente per consentire l'accettazione dei linguaggi di tipo 0.

Dimostriamo separatamente la necessità e la sufficienza.

Teorema 5.9 *Se $\mathcal{G}$ è una grammatica di tipo 0, e $L = L(\mathcal{G})$ è il linguaggio da essa generato, esiste una macchina di Turing non deterministica a due nastri $\mathcal{M}_L$ che accetta L.*

Dimostrazione. La dimostrazione viene solamente accennata. Sia $\mathcal{G} = \langle V_N, V_T, P, S\rangle$. La macchina $\mathcal{M}_L$ opera nel seguente modo. Data una stringa $w \in V_T^*$, la configurazione iniziale di $\mathcal{M}_L$ è $q_0\#\uparrow w\#\uparrow S$.

Ad ogni passo, in modo non deterministico $\mathcal{M}_L$ applica sulla forma di frase ϕ presente sul secondo nastro tutte le possibili produzioni in P, rimpiazzando ϕ con una nuova forma di frase ϕ' derivabile da ϕ. Quindi verifica se ϕ' coincide

con w: solo se la verifica dà esito positivo la macchina entra in uno stato finale di accettazione. □

Esercizio 5.22 Esemplificare la costruzione della macchina $\mathcal{M}_L$ su una grammatica per $\{w\tilde{w} \mid w \in V_T^*\}$.

Si noti che il metodo utilizzato dalla macchina $\mathcal{M}_L$ è assimilabile al metodo di analisi sintattica discendente illustrato nel caso dei linguaggi di tipo 2 (vedi Sezione 4.7).

Chiaramente, dal teorema precedente deriva immediatamente il seguente corollario.

Corollario 5.10 *I linguaggi di tipo* 0 *sono semidecidibili secondo Turing.*

Dimostriamo ora che il potere computazionale delle macchine di Turing è sufficiente per il riconoscimento dei linguaggi di tipo 0.

Teorema 5.11 *Se $\mathcal{M}$ è una macchina di Turing che accetta il linguaggio L allora esiste una grammatica $\mathcal{G}_L$ di tipo 0 tale che $L = L(\mathcal{G}_L)$.*

Dimostrazione. Sia $\mathcal{M} = \langle \Gamma, \text{ƀ}, Q, \delta, q_0, F \rangle$ la macchina di Turing che accetta il linguaggio $L \subseteq \Sigma^*$, con $\Sigma \subseteq \Gamma$. Ciò significa che per ogni $x \in \Sigma^*$, la macchina $\mathcal{M}$ termina in uno stato $q \in F$ se e solo se $x \in L$. Assumiamo per semplicità che L non contenga la stringa vuota.

La grammatica $\mathcal{G}_L$ che genera L opera intuitivamente nel seguente modo:

1. Produce una forma di frase che codifica una stringa $x \in \Sigma^*$.
2. Simula il comportamento che la macchina $\mathcal{M}$ avrebbe con input x.
3. Genera la stringa x se e solo se x è stata accettata da $\mathcal{M}$ nel corso della simulazione.

La grammatica $\mathcal{G}_L$ utilizza un alfabeto di simboli terminali $V_T = \Sigma$ e un alfabeto V_N di simboli non terminali che contiene l'assioma S, alcuni simboli ausiliari $A_1, \ldots, A_5$, un simbolo

$$\begin{bmatrix} \\ q \end{bmatrix} \text{ per ogni } q \in Q,$$

nonché simboli che rappresentano coppie di caratteri in $\Sigma \cup \{\text{ƀ}\} \times \bar{\Gamma}$ e che, per facilitare l'intuizione, rappresenteremo nel seguente modo:

$$\begin{bmatrix} a_i \\ b_j \end{bmatrix} \text{ ove } a_i \in \Sigma \cup \{\text{ƀ}\} \text{ e } b_j \in \bar{\Gamma}.$$

Tale rappresentazione ci permetterà di costruire forme di frase interpretabili come due piste di un unico nastro. Per illustrare le produzioni mostriamo più in dettaglio come opera la grammatica.

Fase 1 Per ogni stringa $x = a_{i_1} \dots a_{i_n} \in \Sigma^*$ la grammatica genera una rappresentazione della configurazione iniziale $q_0 x$ di $\mathcal{M}$. Tale rappresentazione è strutturata nel seguente modo:

$$A_1 \begin{bmatrix} \\ q_0 \end{bmatrix} \begin{bmatrix} a_{i_1} \\ a_{i_1} \end{bmatrix} \dots \begin{bmatrix} a_{i_n} \\ a_{i_n} \end{bmatrix} A_3$$

Fase 2 In questa fase vengono generati su entrambe le piste un numero arbitrario di $\bar{b}$, sia verso destra sia verso sinistra, ottenendo così forme di frase del tipo

$$A_1 \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} \dots \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} \begin{bmatrix} \\ q_0 \end{bmatrix} \begin{bmatrix} a_{i_1} \\ a_{i_1} \end{bmatrix} \dots \begin{bmatrix} a_{i_n} \\ a_{i_n} \end{bmatrix} \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} \dots \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} A_3$$

Fase 3 Osserviamo che, per costruzione, il contenuto della pista inferiore corrisponde al contenuto del nastro della macchina $\mathcal{M}$ che deve accettare o meno la stringa x. In questa fase, quindi, viene attuata la simulazione della macchina $\mathcal{M}$ applicando, sulla pista inferiore, le regole di transizione della macchina stessa.

Fase 4 Si passa dalla Fase 3 alla Fase 4 se la macchina $\mathcal{M}$ accetta la stringa x, cioè entra in una configurazione finale del tipo $b_1 \dots b_k q_\ell b_{k+1} \dots b_m$, con $b_i \in \bar{\Gamma}$, $i \in \{1, \dots, m\}$, $q_\ell \in F$. La forma di frase che corrisponde a tale configurazione finale sarà del seguente tipo:

$$A_1 \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} \dots \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} \begin{bmatrix} c_1 \\ b_1 \end{bmatrix} \dots \begin{bmatrix} c_k \\ b_k \end{bmatrix} \begin{bmatrix} \\ q_\ell \end{bmatrix} \begin{bmatrix} c_{k+1} \\ b_{k+1} \end{bmatrix} \dots \begin{bmatrix} c_m \\ b_m \end{bmatrix} \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} \dots \begin{bmatrix} \bar{b} \\ \bar{b} \end{bmatrix} A_3$$

ove $c_i \in \Sigma \cup \{\bar{b}\}$, $i \in \{1, \dots, m\}$, e la stringa x è una sottostringa di $c_1 \dots c_m$. A questo punto, mediante un'opportuna serie di produzioni, dalla forma sentenziale viene generata la stringa x eliminando tutti i simboli non terminali.

La Fase 1 si realizza mediante le seguenti produzioni:

1. $S \to A_1 \begin{bmatrix} \\ q_0 \end{bmatrix} A_2$
2. $A_2 \to \begin{bmatrix} a \\ a \end{bmatrix} A_2$ per ogni $a \in \Sigma$
3. $A_2 \to A_3$.

La Fase 2 si realizza mediante le seguenti produzioni:

4. $A_3 \to \begin{bmatrix} \text{ƀ} \\ \text{ƀ} \end{bmatrix} A_3$

5. $A_1 \to A_1 \begin{bmatrix} \text{ƀ} \\ \text{ƀ} \end{bmatrix}$.

Per la Fase 3 occorrono le seguente produzioni, basate sulle regole di transizione della macchina $\mathcal{M}$:

6. $\begin{bmatrix} \\ q_i \end{bmatrix} \begin{bmatrix} c_1 \\ b_j \end{bmatrix} \begin{bmatrix} c_l \\ b_j \end{bmatrix} \to \begin{bmatrix} c_l \\ b_k \end{bmatrix} \begin{bmatrix} \\ q_h \end{bmatrix}$

   per ogni $c_i \in \Sigma \cup \{\text{ƀ}\}$ ed ogni b_j, q_i, b_k, q_h tali che $\delta(q_i, b_j) = (q_h, b_k, d)$

7. $\begin{bmatrix} c_r \\ b_s \end{bmatrix} \begin{bmatrix} \\ q_i \end{bmatrix} \begin{bmatrix} c_l \\ b_j \end{bmatrix} \to \begin{bmatrix} \\ q_h \end{bmatrix} \begin{bmatrix} c_r \\ b_s \end{bmatrix} \begin{bmatrix} c_l \\ b_k \end{bmatrix}$

   per ogni $c_r, c_l \in \Sigma \cup \{\text{ƀ}\}$, $b_s \in \bar{\Gamma}$ ed ogni b_j, q_i, b_k, q_h tali che $\delta(q_i, b_j) = (q_h, b_k, s)$

8. $\begin{bmatrix} \\ q_i \end{bmatrix} \begin{bmatrix} c_l \\ b_j \end{bmatrix} \to \begin{bmatrix} \\ q_h \end{bmatrix} \begin{bmatrix} c_l \\ b_k \end{bmatrix}$

   per ogni $c_l \in \Sigma \cup \{\text{ƀ}\}$ ed ogni b_j, q_i, b_k, q_h tali che $\delta(q_i, b_j) = (q_h, b_k, i)$.

9. $\begin{bmatrix} \\ q_l \end{bmatrix} \to A_4 A_5$ per ogni $q_\ell \in F$

10. $\begin{bmatrix} c_i \\ b_l \end{bmatrix} \to A_4 c_i$ per ogni $c_i \in \Sigma$ e per ogni $b_l \in \bar{\Gamma}$

11. $\begin{bmatrix} \text{ƀ} \\ b_l \end{bmatrix} A_4 \to A_4$ per ogni $b_l \in \bar{\Gamma}$

12. $\begin{bmatrix} c_i \\ b_l \end{bmatrix} \to c_i A_5$ per ogni $c_i \in \Sigma$ e per ogni $b_l \in \bar{\Gamma}$

13. $\begin{bmatrix} \text{ƀ} \\ b_l \end{bmatrix} \to A_5$ per ogni $b_l \in \bar{\Gamma}$

14. $A_1 A_4 \to \varepsilon$

15. $A_5 A_3 \to \varepsilon$.

Si lascia al lettore il compito di modificare la grammatica per tener conto del caso in cui la macchina M accetti la stringa vuota ε (v. Esercizio 5.23). Non è difficile verificare che tale grammatica genera x se e solo se x è accettato dalla macchina $\mathcal{M}$. □

Esercizio 5.23 Modificare la grammatica introdotta nella dimostrazione precedente in modo da tener conto anche del caso in cui $\varepsilon \in L$.

5.11 Linguaggi di tipo 1 e automi lineari

Nel corso della trattazione svolta finora, abbiamo visto che, come preannunciato nella Sezione 2.5.3, per varie classi di linguaggi definite in base alla gerarchia di Chomsky è possibile individuare un dispositivo di riconoscimento caratterizzante. In particolare abbiamo visto che gli automi a stati finiti riconoscono esattamente i linguaggi regolari, gli automi a pila non deterministici quelli non contestuali e le macchine di Turing consentono di accettare tutti e soli i linguaggi di tipo 0. In questa sezione vedremo come è possibile individuare un automa che caratterizza la classe dei linguaggi contestuali. A tale scopo vedremo come le dimostrazioni date nella sezione precedente, che ci hanno permesso di associare ad una grammatica di tipo 0 una MT che accetta il linguaggio da essa generato e viceversa, possono essere specializzate nel caso in cui la grammatica sia di tipo 1. Ciò ci permetterà di dimostrare innanzi tutto che i linguaggi di tipo 1 sono decidibili e successivamente di definire un automa in grado di riconoscere tutti e soli i linguaggi di tipo 1.

Teorema 5.12 *Se $\mathcal{G}$ è una grammatica di tipo* 1 *che genera il linguaggio $L = L(\mathcal{G})$ esiste una macchina di Turing $\mathcal{M}_L$ che riconosce L.*

Dimostrazione. La dimostrazione si ispira a quella del Teorema 5.9. Data la grammatica $\mathcal{G}$ costruiamo una macchina di Turing non deterministica $\mathcal{M}$ simile a quella introdotta nella dimostrazione del teorema suddetto, e cioè una macchina a due nastri uno dei quali contiene la stringa in input e l'altro (nastro di lavoro) conterrà le forme sentenziali derivabili dalla grammatica che vengono via via generate dalla macchina. Questa sarà però modificata nel modo seguente. Innanzi tutto, sul nastro di lavoro, vengono inseriti due simboli speciali che delimitano lo spazio di lavoro effettivamente utilizzabile e che è pari alla lunghezza della stringa in input. Ogni volta che, applicando le produzioni della grammatica, si tenta di generare una forma sentenziale di lunghezza superiore allo spazio a disposizione, la computazione termina in uno stato non finale. Infatti, data la caratteristica delle grammatiche di tipo 1, da tale forma sentenziale non sarebbe possibile derivare la stringa in input. Inoltre, poiché si può verificare che la forma sentenziale generata rimanga sempre della stessa lunghezza a causa dell'applicazione ciclica di una sequenza di produzioni, sarà

necessario prevedere che anche in questo caso la computazione termini in uno stato non finale. In particolare, se $\mathcal{G}$ ha un alfabeto V di simboli terminali e non terminali, e se la stringa in input ha lunghezza n, la computazione dovrà terminare in uno stato non finale dopo che per $\mid V \mid^n$ passi consecutivi la lunghezza della forma sentenziale è rimasta costante. In tal modo la macchina $\mathcal{M}$ terminerà sempre le proprie computazioni, o in uno stato finale (accettazione) o in uno stato non finale (rifiuto).

La macchina $\mathcal{M}_L$ che riconosce il linguaggio L potrà essere ottenuta costruendo una macchina deterministica che simula la macchina $\mathcal{M}$ (cfr. Esercizio 5.14) e quindi riconosce una stringa x se e solo se essa è generata dalla grammatica $\mathcal{G}$. □

Come abbiamo visto, nella dimostrazione precedente abbiamo imposto che la macchina di Turing non deterministica $\mathcal{M}$ utilizzi su entrambi i nastri, se non si tiene conto dei simboli di delimitazione, un numero di celle al più pari alla lunghezza della stringa data in input. Vedremo ora che è possibile definire una macchina di Turing non detreministica con lo stesso potere computazionale che utilizza un solo nastro della stessa lunghezza.

Definizione 5.23 *Un* automa lineare *(*linear bounded automaton*) è una settupla* $\mathcal{M} = \langle \Gamma, \flat, \$, Q, q_0, F, \delta_{\mathrm{N}} \rangle$ *che corrisponde ad una macchina di Turing non deterministica ad un nastro con le seguenti varianti. Il simbolo* \$ *non appartiene a* Γ *ed è utilizzato per delimitare lo spazio di lavoro a disposizione della macchina. La macchina viene attivata nella configurazione iniziale* $\$q_0x\$$ *e il simbolo speciale* \$ *non può essere cancellato né scavalcato dalla testina della macchina.*

Teorema 5.13 *Se* $\mathcal{G}$ *è una grammatica di tipo* 1 *che genera il linguaggio* $L = L(\mathcal{G})$ *esiste un automa lineare* $\mathcal{B}_L$ *che riconosce* L*.*

Dimostrazione. Per dimostrare l'enunciato è sufficiente modificare la dimostrazione del teorema precedente adattandola nel modo seguente. Definiamo la macchina $\mathcal{B}_L$ in modo simile alla macchina $\mathcal{M}_L$, ma anziché prevedere due nastri facciamo uso di un unico nastro, diviso in quattro tracce. Come richiesto dalla definizione di automa lineare, lo spazio disponibile su tale nastro sarà delimitato dai simboli \$ e sarà pari alla lunghezza della stringa in input. Le quattro tracce del nastro di $\mathcal{B}_L$ sono utilizzate per rappresentare il contenuto dei due nastri di $\mathcal{M}_L$ (tracce pari) e la posizione delle rispettive testine (tracce dispari). Come si vede, la tecnica è la stessa della dimostrazione del Teorema 5.1. Il numero di celle usate dalla macchina $\mathcal{B}_L$ sarà pari a quello delle celle usate dalla macchina $\mathcal{M}_L$. □

Nel prossimo teorema vedremo che non solo gli automi lineari consentono il riconoscimento di tutti i linguaggi di tipo 1, ma che per ogni automa lineare che riconosce un dato linguaggio L esiste una grammatica di tipo 1 che genera L.

Teorema 5.14 *Se $\mathcal{B}$ è un automa lineare che accetta il linguaggio L allora esiste una grammatica $\mathcal{G}_L$ di tipo 1 tale che $L = L(\mathcal{G}_L)$.*

Dimostrazione. Anche questa dimostrazione si ispira alla analoga dimostrazione per i linguaggi di tipo 0 (Teorema 5.11). In particolare il nostro intendimento sarà di generare stringhe di caratteri e simulare il comportamento di un automa lineare su di esse, generando come sequenze di simboli terminali solo quelle stringhe che saranno accettate dall'automa lineare.

Esaminando le produzioni nella dimostrazione del Teorema 5.11, possiamo osservare che quattro di esse non sono di tipo 1 (9, 10, 11 e 12) e dunque rendono la grammatica corrispondente strettamente di tipo 0. In particolare, le produzioni 9 e 10 vengono utilizzate quando la computazione eseguita dalla macchina di Turing ha utilizzato una quantità di nastro superiore alla lunghezza della stringa in input. Poiché nel caso degli automi lineari ciò non potrà mai verificarsi tali produzioni non saranno necessarie nella grammatica che dobbiamo definire. Per lo stesso motivo, le produzioni 4 e 5, che generano un arbitrario numero di blank a destra e a sinistra della stringa in esame, non saranno più necessarie. Rimane pertanto da risolvere il problema costituito dalle produzioni 11 e 12, che eliminano i delimitatori A_1 e A_3 quando la computazione della macchina di Turing ha raggiunto lo stato finale. Per poter effettuare queste trasformazioni senza utilizzare produzioni strettamente di tipo 0 modificheremo la dimostrazione del Teorema 5.11 utilizzando caratteri organizzati su quattro piste anziché su due. Oltre alle due piste già presenti nella suddetta dimostrazione, che ora chiameremo piste 3 e 4, utilizzeremo una pista (pista 2) per rappresentare la posizione della testina della macchina e lo stato in cui essa si trova ed una pista (pista 1) per rappresentare i due simboli \$ che delimitano a destra e sinistra lo spazio di lavoro a disposizione dell'automa lineare. Inoltre la pista 2 sarà utilizzata anche per simboli di servizio necessari durante l'ultima fase della costruzione.

In definitiva, quindi, le produzioni della grammatica $\mathcal{G}_L$ saranno ottenute modificando opportunamente quelle della grammatica costruita nella dimostrazione del Teorema 5.11, per tener conto della nuova struttura dei caratteri.

Illustriamo ora le fasi attraverso cui avviene la generazione di una stringa del linguaggio L mediante la grammatica $\mathcal{G}_L$.

Fase 1 Per ogni stringa $x = a_{i_1} \dots a_{i_n} \in \Sigma^*$ la grammatica genera una rappresentazione della configurazione iniziale $\$q_0x\$$ di $\mathcal{B}$, così strutturata:

$$\begin{bmatrix} \$ \\ q_0 \end{bmatrix} a_{i_1} a_{i_1} \begin{bmatrix} \\ \end{bmatrix} a_{i_2} a_{i_2} \cdots \begin{bmatrix} \$ \\ \end{bmatrix} a_{i_n} a_{i_n}$$

Fase 2 In questa fase viene attuata la simulazione della macchina $\mathcal{B}$ applicando, sulla pista inferiore, le regole di transizione della macchina stessa.

Fase 3 Si passa dalla Fase 2 alla Fase 3 se la macchina $\mathcal{B}$ accetta la stringa x, cioè entra in una configurazione finale del tipo $\$b_1 \dots b_k q_\ell b_{k+1} \dots b_n\$$, con $b_i \in \bar{\Gamma}$, $i \in \{1, \dots, n\}$, $q_\ell \in F$. La forma sentenziale che corrisponde a tale configurazione finale sarà del seguente tipo:

$$\left[\begin{array}{c}\$\\ \\ \end{array}\right] a_{i_1} b_1 \cdots \left[\begin{array}{c}\\ q_\ell \\ \end{array}\right] a_{i_{k+1}} b_{k+1} \cdots \left[\begin{array}{c}\$\\ \\ \end{array}\right] a_{i_n} b_n$$

A questo punto, mediante un'opportuna serie di produzioni, dalla forma sentenziale viene generata la stringa x eliminando tutti i simboli non terminali.

La definizione delle produzioni che realizzano le tre fasi di generazione di una stringa del linguaggio L vengono lasciate al lettore (cfr. Esercizio 5.24). □

Esercizio 5.24 Definire le produzioni di tipo 1 della grammatica $\mathcal{G}_L$ introdotta nella dimostrazione del Teorema 5.14. Per quanto riguarda la Fase 3, si suggerisce di utilizzare la pista 1 per gestire la propagazione di opportune marche che svolgono un ruolo analogo a quello svolto dai non terminali A_4 ed A_5 nella dimostrazione del Teorema 5.11.

Capitolo 6

Modelli di Calcolo Imperativi e Funzionali

6.1 Introduzione

Il modello di calcolo creato da Turing per definire il concetto di calcolabilità è stato introdotto senza che vi fosse alcun riferimento alla struttura ed al funzionamento dei calcolatori elettronici che, come è noto, hanno visto la luce pochi anni dopo.

Come si è già detto (vedi Appendice A), nonostante l'appellativo di "macchine", i dispositivi astratti definiti da Turing corrispondono solo ad una formalizzazione dei procedimenti di calcolo umani e farebbe certamente sorridere l'ipotesi di realizzare un effettivo calcolatore basandosi su tale modello. La stessa considerazione vale anche per altri modelli di calcolo introdotti nello stesso periodo da altri matematici: lambda-calcolo, sistemi di riscrittura di Post, algoritmi di Markov.

Il fatto che tali modelli formali siano stati concepiti nell'ambito della logica matematica, indipendentemente dagli studi che stavano accompagnando la nascita dei primi calcolatori, non vuol dire però che, a posteriori, essi non si siano rivelati, per l'informatica, validi strumenti di formalizzazione di aspetti e proprietà del calcolo automatico. Ad esempio, i sistemi di riscrittura di Post hanno ispirato i modelli grammaticali di Chomsky, utili per lo studio delle proprietà sintattiche dei programmi e le stesse macchine di Turing continuano ancor oggi ad essere adottate come modello di riferimento per lo studio e la misura della complessità di risoluzione di problemi.

In questo capitolo prenderemo in esame invece due classi di modelli che, pur essendo anch'essi modelli astratti, hanno però un diretto riferimento ai calcolatori e ai linguaggi di programmazione e consentono di formalizzare i processi di esecuzione di programmi e di confrontare i relativi paradigmi di programmazione: i modelli di calcolo imperativi e i modelli di calcolo funzio-

nali. I primi saranno proposti attraverso l'introduzione di macchine astratte (le macchine a registri) basate sulla struttura classica degli elaboratori e programmabili con linguaggi simili ai linguaggi assemblativi, anche se molto più semplici; ai secondi giungeremo attraverso un approccio che, partendo da una definizione matematica delle funzioni calcolabili (funzioni ricorsive), ci permetterà di introdurre caratteristiche e proprietà di un semplice linguaggio di programmazione basato sul paradigma funzionale (formalismo di McCarthy).

6.2 Macchine a registri

I modelli di calcolo imperativi sono direttamente collegati all'architettura dei primi calcolatori elettronici, architettura che verso la metà degli anni '40 si è definitivamente consolidata secondo un modello che ha preso il nome di "modello di von Neumann", dal nome del matematico ungherese che ha avuto un ruolo fondamentale nella sua concezione.

Per il suo stretto legame con il modello di von Neumann, il paradigma imperativo caratterizza da sempre il linguaggio di macchina e i linguaggi assemblativi dei calcolatori reali (che ancora oggi sono fondamentalmente ispirati a tale modello) e anche la maggior parte dei linguaggi di programmazione ad alto livello, dai più antichi come il Fortran e il Cobol ai piú recenti come C e Java.

In un modello di calcolo imperativo un algoritmo viene descritto mediante una sequenza finita di istruzioni (*programma*), i dati da elaborare sono contenuti in un insieme finito di celle (*memoria*) e il calcolo viene realizzato da una *macchina* che esegue sequenzialmente le istruzioni, modificando via via, secondo quanto richiesto dalle stesse, il contenuto della memoria. Tipiche istruzioni di un linguaggio imperativo sono le istruzioni di *lettura* dei dati (da tastiera o da un supporto esterno), di *trasferimento* dei dati da una cella ad un'altra, di esecuzione di operazioni *aritmetiche* o *logiche* elementari (somme e prodotti aritmetici o logici etc.), di *visualizzazione* dei risultati (su schermo o su un supporto esterno).

In un tale modello di calcolo assume un ruolo fondamentale il concetto di *stato*, rappresentato in questo caso dal contenuto della memoria in un determinato istante. Ogni istruzione di un linguaggio di tipo imperativo comporta il passaggio da uno stato ad un altro e l'esecuzione di un programma imperativo si può rappresentare quindi come una sequenza di stati, analogamente a una computazione eseguita da una macchina di Turing che è anch'essa, come visto, riconducibile ad una sequenza di configurazioni istantanee.

Per discutere caratteristiche e proprietà computazionali del modello di calcolo imperativo introdurremo un tipo particolare di macchina astratta: la macchina a registri (detta anche RAM, Random Access Machine).

Una macchina a registri, è chiamata così perché la sua memoria consiste in una sequenza di un numero finito, ma arbitrario, di *registri*, celle di memoria ciascuna delle quali è in grado di contenere un intero grande a piacere.

I registri sono accessibili direttamente, in base al loro numero d'ordine. Un registro particolare, chiamato *accumulatore*, è destinato a contenere via via, uno degli operandi su cui agiscono le istruzioni della macchina. Un altro registro, chiamato contatore delle istruzioni (CI) contiene il numero d'ordine della prossima istruzione che dovrà essere eseguita. La macchina scambia informazioni con il mondo esterno mediante due *nastri di ingresso e uscita* consistenti di sequenze di celle, ciascuna delle quali ancora capace di contenere un intero grande a piacere. La macchina, infine, è dotata di una *unità centrale* capace di eseguire le istruzioni del linguaggio di programmazione. La Figura 6.1 fornisce una rappresentazione di una macchina a registri.

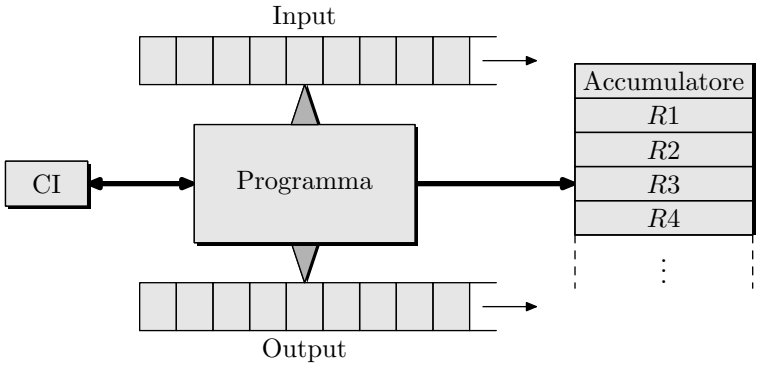


FIGURA 6.1 Rappresentazione fisica di una macchina a registri.

Un *programma* per RAM consiste in una sequenza di *istruzioni* di vario tipo (di trasferimento, aritmetiche, di controllo, di I/O). In genere un'istruzione agisce su *operandi* contenuti nei registri, che in tal caso vengono indicati semplicemente con il numero d'ordine del registro (ad esempio 3 indica il contenuto del registro 3). Quando un operando viene indirizzato in modo indiretto tramite il contenuto di un altro registro esso viene indicato con il numero del registro contenuto fra parentesi (ad esempio (3) indica il contenuto del registro indirizzato dal registro 3). Infine un operando può essere direttamente costituito dal dato su cui si vuole operare; in tal caso il dato viene preceduto da # (ad esempio #3 indica che l'operando è l'intero 3).

Di seguito forniamo una semplice descrizione in BNF del linguaggio delle macchine a registri. Da tale descrizione è possibile determinare, per ciascuna istruzione, il tipo di operandi che essa ammette.

```
<programma>   ::= <istruzione> {<istruzione>}
<istruzione>  ::= LOAD     [#]<operando> | (<operando>) |
                  STORE    <operando> | (<operando>) |
```

```
                    ADD       [#]<operando> | (<operando>) |
                    SUB       [#]<operando> | (<operando>) |
                    MULT      [#]<operando> | (<operando>) |
                    DIV       [#]<operando> | (<operando>) |
                    READ      <operando> | (<operando>) |
                    WRITE     [#]<operando> | (<operando>) |
                    JUMP      <etichetta> |
                    JGTZ      <etichetta> |
                    JZERO     <etichetta> |
                    HALT
<operando>    ::= <intero>
<etichetta>   ::= <intero>
```

Vediamo ora il significato delle varie istruzioni disponibili.

Le *istruzioni di trasferimento* (LOAD e STORE) permettono di trasferire il contenuto di un registro nell'accumulatore e viceversa.

Le *istruzioni aritmetiche* (ADD, somma, SUB, sottrazione, MULT, moltiplicazione, DIV, parte intera della divisione) permettono di eseguire le quattro operazioni tra il contenuto dell'accumulatore ed il contenuto di un registro. Il risultato rimane nell'accumulatore. Nota che l'istruzione SUB dà risultato 0 se si tenta di sottrarre un numero maggiore da un numero minore.

Le istruzioni vengono sempre eseguite sequenzialmente tranne nei casi in cui la sequenza di esecuzione non venga alterata da una *istruzione di controllo* e cioè una istruzione di HALT o una istruzione di salto. Quando si incontra l'istruzione HALT l'esecuzione del programma si arresta; in tal caso diremo che il calcolo è terminato.

Le istruzioni di salto sono salti incondizionati (JUMP) o condizionati in base al contenuto dell'accumulatore (JGTZ, JZERO cioè salta se l'accumulatore contiene un intero maggiore di zero, o rispettivamente, uguale a zero); l'operando di un'istruzione di salto è il numero d'ordine dell'istruzione alla quale eventualmente si effettua il salto. Nel caso in cui si effettui un salto ad una istruzione inesistente, vale a dire se il valore contenuto nel contatore delle istruzione è uguale a 0 o superiore alla lunghezza del programma, il programma si arresta e in tal caso diremo che il calcolo non è definito.

Infine le due *istruzioni di I/O* (READ e WRITE) consentono di leggere un numero intero sul nastro di ingresso e trasferirlo in un registro e, rispettivamente, di scrivere il contenuto di un registro sul nastro di uscita.

Come si vede il significato delle istruzioni è molto intuitivo e corrisponde a quello di analoghe istruzioni che fanno parte di tutti i principali linguaggi Assembler.[1] La principale differenza tra una RAM e un calcolatore reale con-

[1] Si noti che nei linguaggi Assembler reali si fa tipicamente uso di istruzioni a due operandi anziché ad uno solo, e ci si giova di un numero elevato di registri sui quali effettuare le operazioni.

siste nel fatto che il programma della RAM non è contenuto in memoria ma è, per così dire, "cablato". In altre parole la RAM non prevede le fasi di 'fetch' e di interpretazione delle istruzioni ma ogni istruzione viene direttamente eseguita secondo quello che potremmo definire un 'microprogramma' che specifica formalmente come essa agisce sullo stato della macchina (determinato dal contenuto di accumulatore, registri e contatore di istruzioni). Mostriamo a titolo di esempio come si può definire formalmente il comportamento di alcune istruzioni.

Indichiamo con R[i] il contenuto del registro i-esimo, con R[0] il contenuto dell'accumulatore e con CI il contenuto del contatore delle istruzioni e con $n \geq 0$ un generico intero non negativo.

LOAD #n	R[0] := n CI := CI + 1
ADD (n)	R[0] := R[0] + R[R[n]] CI := CI + 1
JZERO n	**if** R[0] = 0 **then** CI := n **else** CI := CI + 1

Il comportamento delle altre istruzioni si può definire in modo del tutto analogo.

Esercizio 6.1 Definire il "microprogramma" corrispondente alle istruzioni DIV n, MULT #n, JGTZ n.

Per descrivere compiutamente il comportamento di un programma per macchine a registri dobbiamo fare alcune ulteriori assunzioni. In particolare dobbiamo supporre che all'inizio le informazioni da elaborare siano fornite nell'opportuno ordine sul nastro di ingresso e che il contatore delle istruzioni sia inizializzato ad 1.

Mostriamo ora un semplice esempio di programma per RAM per il calcolo dell'esponenziale.

Esempio 6.1 Supponiamo che due interi $x > 0$ e $y \geq 0$ siano forniti sul nastro di ingresso. Il seguente programma dà in uscita il valore x^y.

```
      READ    1
      READ    2
      LOAD    #1
      STORE   3
5     LOAD    2
      JZERO   14
      LOAD    1
      MULT    3
      STORE   3
      LOAD    2
```

```
        SUB     #1
        STORE   2
        JUMP    5
14      WRITE   3
        HALT
```

Esercizio 6.2 Definire un programma per RAM per il calcolo della parte intera di $\log_2 x$, $x \geq 1$.

Esercizio 6.3 Definire un programma per RAM per il calcolo di $x!$, $x \geq 0$.

Per la loro affinità con la struttura e il funzionamento di un calcolatore reale, le RAM sono un modello estremamente interessante sia nell'ambito di studi di calcolabilità sia con riferimento agli aspetti di complessità computazionale. Dal primo punto di vista vedremo come la RAM ci consente di definire la classe delle funzioni intere calcolabili e di confrontarla con la classe delle funzioni calcolabili secondo Turing. Dal punto di vista della complessità vedremo che, sotto opportune ipotesi, le RAM sono fondamentali nella caratterizzazione della classe dei problemi computazionalmente trattabili.

6.2.1 Modelli per il costo di esecuzione di programmi RAM

Prima di poter correlare il comportamento delle RAM con altri modelli di calcolo come le macchine di Turing è necessario chiarire come si valuta il costo di esecuzione di un programma per RAM.

Un metodo semplice, ma già abbastanza significativo, per analizzare il costo di un programma per RAM, chiamato *modello a costi uniformi* consiste nell'attribuire un costo unitario a tutte le istruzioni e limitarsi a contare quante volte ogni istruzione viene eseguita in funzione dell'input del programma. Riprendiamo in esame l'esempio precedente; possiamo vederlo diviso in tre parti; lettura dell'input e inizializzazione, esecuzione del ciclo, terminazione. La prima e l'ultima parte hanno rispettivamente un costo pari a 4 e a 2, indipendentemente dal valore dell'input. Il corpo del programma, cioè il ciclo, consta di 9 istruzioni che vengono eseguite y volte. Inoltre le prime due istruzioni del ciclo vengono eseguite una volta in più, quando il valore y diventa uguale a zero e si effettua il salto all'istruzione 14. In definitiva il costo di esecuzione del programma è pari a $4 + 9y + 2 + 2 = 9y + 8$. Utilizzando la notazione asintotica il costo è dunque $O(y)$.

Chiaramente il modello a costi uniformi è un modello di analisi dei costi troppo idealizzato, perché in esso si fanno due ipotesi eccessivamente semplificative: che il costo di esecuzione di un'istruzione sia indipendente dalla taglia degli operandi e che l'accesso ad un registro abbia un costo unitario indipendentemente dal numero di registri accessibili.

Per ottenere un modello piu realistico, invece, è ragionevole pensare che il costo di esecuzione delle istruzioni dipenda dalla taglia degli operandi. Infatti i registri di una RAM possono contenere interi arbitrariamente grandi (a differenza delle celle di memoria di un calcolatore, che in genere contengono interi rappresentabili con 32 o 64 bit) e quindi se si opera su un intero n contenuto in un registro si assume di pagare un costo dell'ordine di $\log n$. Inoltre il costo di accesso alla memoria deve dipendere, in qualche modo, dalla quantità di memoria che si vuole indirizzare.

Sulla base di quanto detto, possiamo introdurre il *modello a costi logaritmici*, nel quale il costo di esecuzione di un'istruzione si può determinare nel seguente modo. Sia $l(n)$ definito come $l(n) =$ if $n = 0$ then 1 else $\lceil \log_2 n \rceil + 1$. Per valutare il costo di un'istruzione si mettono in conto i seguenti contributi:

1. l'elaborazione di un operando n ha costo $l(n)$,
2. ogni accesso ad un registro i ha costo $l(i)$.

I costi delle istruzioni sono dunque dati dalla seguente tabella:

LOAD	op	$c(\text{op})$
STORE	op	$c'(\text{op}) + l(R[0])$
ADD	op	$c(\text{op}) + l(R[0])$
SUB	op	$c(\text{op}) + l(R[0])$
MULT	op	$c(\text{op}) + l(R[0])$
DIV	op	$c(\text{op}) + l(R[0])$
READ	op	$c'(\text{op}) + l(\text{input})$
WRITE	op	$c(\text{op})$
JGTZ	et	$l(R[0])$
JZERO	et	$l(R[0])$
JUMP	et	1
HALT		1

dove $c(\text{op})$ e $c'(\text{op})$ dipendono dalla forma dell'operando op, e precisamente: $c(\#\text{n}) = l(n)$, $c(i) = l(i) + l(R[i])$, $c((i)) = l(i) + l(R[i]) + l(R[R[i]])$ e $c'(i) = l(i)$, $c'((i)) = l(i) + l(R[i])$. Inoltre, con $l(\text{input})$ si intende il valore della funzione l applicata all'intero contenuto nella cella del nastro di input correntemente in lettura.

Si noti che, per quanto riguarda la moltiplicazione, assumere un costo logaritmico è, in un certo senso, una forzatura; infatti ad oggi non è noto alcun metodo che consenta di eseguire la moltiplicazione di due interi di n bit con un costo computazionale $o(n \log n)$. Tuttavia adottare il costo logaritmico indicato nella tabella consente di ottenere una ragionevole approssimazione con una trattazione molto più semplice.

Vediamo ora quale risulta, da un punto di vista asintotico, il costo di esecuzione del programma dell'Esempio 6.1 secondo il modello a costi logaritmici,

istruzione per istruzione, tenendo conto del valore massimo allocato in ogni registro e di quante volte ogni istruzione viene eseguita (ultima colonna). (Si ricordi che nel registro 1 viene inserito l'intero x, nel registro 2 viene inserito l'intero y e nel registro 3 viene costruito il risultato x^y).

	READ	1	$l(1) + l(x)$	$O(\log x)$	1
	READ	2	$l(2) + l(y)$	$O(\log y)$	1
	LOAD	#1	$l(1)$	$O(1)$	1
	STORE	3	$l(3) + l(1)$	$O(1)$	1
5	LOAD	2	$l(2) + l(y)$	$O(\log y)$	$y + 1$
	JZERO	14	$l(y)$	$O(\log y)$	$y + 1$
	LOAD	1	$l(1) + l(x)$	$O(\log x)$	y
	MULT	3	$l(3) + l(x) + l(\mathrm{x^y})$	$O(\log x + y \log x)$	y
	STORE	3	$l(3) + l(\mathrm{x^y})$	$O(y \log x)$	y
	LOAD	2	$l(2) + l(y)$	$O(\log y)$	y
	SUB	#1	$l(1) + l(y)$	$O(\log y)$	y
	STORE	2	$l(2) + l(y)$	$O(\log y)$	y
	JUMP	5	1	$O(1)$	y
14	WRITE	3	$l(3) + l(\mathrm{x^y})$	$O(y \log x)$	1
	HALT		1	$O(1)$	1

In totale abbiamo che il costo di esecuzione del programma è:
$O(1) + O(\log x) + O(\log y) + O((y + 1) \log y) + O(y \log x) + O(y^2 \log x) + O(y \log y) + O(y) = O(y^2 \log x)$.

Esercizio 6.4 Determinare il costo di esecuzione del programma per il calcolo di $x!$ (Esercizio 6.3) sia con il modello a costi uniformi sia con il modello a costi logaritmici.

6.3 Macchine a registri e macchine di Turing

Analogamente a quanto si è fatto per le Macchine di Turing è possibile definire un concetto di funzione calcolabile con macchine a registri.

Definizione 6.1 *Una funzione $f : \mathbb{N}^n \to \mathbb{N}$ è* calcolabile con macchine a registri *se esiste un programma P per RAM tale che, se $f(x_1, \ldots, x_n) = y$ allora il programma P, inizializzato con $x_1, \ldots, x_n$ sul nastro di input, termina con y sul nastro di output e se $f(x_1, \ldots, x_n)$ non è definita allora il programma P non termina.*

Nel seguito vedremo che le RAM sono in grado di calcolare tutte e sole le funzioni calcolabili secondo Turing. Infatti, anche se il modello delle macchine a registri è orientato al calcolo di funzioni numeriche, non è difficile mostrare, mediante delle simulazioni, che le RAM hanno lo stesso potere computazionale delle Macchine di Turing.

Vediamo innanzitutto come una Macchina di Turing può essere simulata da una RAM. Per semplicità assumiamo che la macchina abbia un solo nastro semi-infinito e abbia come alfabeto i due soli simboli 0 e 1, ma si può facilmente generalizzare la dimostrazione al caso di Macchine di Turing a più nastri e con un alfabeto qualunque, purché finito.

Teorema 6.1 *Data una qualunque Macchina di Turing $\mathcal{M}$ con nastro semi-infinito e con alfabeto di nastro $\Gamma = \{0, 1\}$, esiste una RAM con relativo programma P tale che se $\mathcal{M}$ realizza la transizione dalla configurazione iniziale q_0x alla configurazione finale q_Fy e se la RAM è inizializzata con la stringa x nei registri $2, \ldots, \mid x \mid +1$, al termine della computazione la macchina a registri avrà nei registri $2, \ldots, \mid y \mid +1$ la stringa y. Inoltre se la macchina $\mathcal{M}$ opera in tempo T la RAM opera, nel modello a costi logaritmici, in tempo $O(T \log T)$.*

Dimostrazione. Per realizzare la simulazione consideriamo che alla cella i-esima del nastro di $\mathcal{M}$ corrisponda il registro i + 1 della RAM e che esso contenga 0 o 1 a seconda del contenuto della i-esima cella. Inoltre il registro 1 sarà utilizzato per indirizzare la cella correntemente letta dalla testina della macchina $\mathcal{M}$. Poiché la macchina $\mathcal{M}$ all'inizio è posizionata sulla prima cella, all'inizio il registro 1 indirizzerà il registro 2. Per ogni stato della macchina $\mathcal{M}$ il programma P conterrà una sequenza di istruzioni. Ad esempio, se in corrispondenza dello stato p la macchina prevede le seguenti due regole di transizione $\delta(p, 0) = (q, 1, d)$ e $\delta(p, 1) = (p, 1, s)$, il programma P dovrà prevedere il seguente frammento (che per facilitare l'intuizione supponiamo inizi con l'etichetta p):

```
        ...
p       LOAD    (1)
        JGTZ    p'
        LOAD    #1
        STORE   (1)
        LOAD    1
        ADD     #1
        STORE   1
        JUMP    q
p'      LOAD    1
        SUB     #1
        STORE   1
        JUMP    p
        ...
```

È facile vedere, dunque, che ad ogni transizione della macchina di Turing la RAM fa corrispondere l'esecuzione di al più 8 istruzioni ciascuna delle quali

ha un costo che è $O(\log i_{\mathrm{MAX}})$ dove i_{MAX} indica il massimo numero di celle usate da $\mathcal{M}$. Poiché, se la macchina $\mathcal{M}$ esegue T passi, abbiamo $i_{\mathrm{MAX}} \leq \mathrm{T}{+}1$ segue che il costo complessivo di esecuzione del programma P è $O(T \log T)$. □

Similmente possiamo mostrare che una macchina di Turing è in grado di simulare una RAM. In questo caso, per comodità e senza restrizione di generalità utilizziamo una macchina di Turing dotata di nastro di input, nastro di output e tre nastri di lavoro.

Teorema 6.2 *Data una macchina a registri con programma P che calcola la funzione f esiste una macchina di Turing $\mathcal{M}$ tale che se $f(x_1, \ldots, x_n) = y$ e sul nastro di input sono memorizzati in binario gli interi $x_1, \ldots, x_n$, la macchina $\mathcal{M}$ termina con la rappresentazione binaria di y sul nastro di output e se $f(x_1, \ldots, x_n)$ non è definita la macchina $\mathcal{M}$ non termina in una configurazione finale. Inoltre, se nel modello a costi logaritmici la RAM ha un costo T, $\mathcal{M}$ opera in tempo $O(T^2)$.*

Dimostrazione. I tre nastri di lavoro della macchina $\mathcal{M}$ sono usati nel seguente modo. Sul nastro T_1 sono memorizzati in binario tutti i contenuti dei registri effettivamente utilizzati dalla RAM, ciascuno preceduto dal numero d'ordine del registro stesso, sempre rappresentato in binario. Sul nastro T_2 è rappresentato in binario il contenuto dell'accumulatore. Infine il nastro T_3 è utilizzato come nastro di lavoro (Figura 6.2). Per ogni istruzione prevista dalla RAM la macchina $\mathcal{M}$ avrà alcuni stati che consentono di eseguire le operazioni corrispondenti. Se si tratta di istruzioni di salto si deve solo verificare se il contenuto dell'accumulatore soddisfa la condizione del salto ed eseguire le opportune transizioni. Se si tratta di operazioni che non alterano il contenuto dei registri (LOAD, WRITE, istruzioni aritmetiche) è sufficiente:

- cercare sul nastro T_1 il registro con il quale si deve operare,
- eseguire l'operazione prevista tra il contenuto del registro e l'accumulatore o il nastro di output, modificando di conseguenza il contenuto dell'accumulatore o del nastro di output.

Nel caso che si debba fare un'operazione di READ o di STORE e si renda quindi necessario modificare il contenuto di un registro, si deve:

- trascrivere dal nastro T_1 al nastro T_3 il contenuto del nastro T_1 a destra del registro che si deve modificare,
- eseguire l'operazione prevista, modificando il contenuto del registro cercato,
- ricopiare di nuovo dal nastro di lavoro sul nastro T_1 la parte del nastro che non è stata modificata.

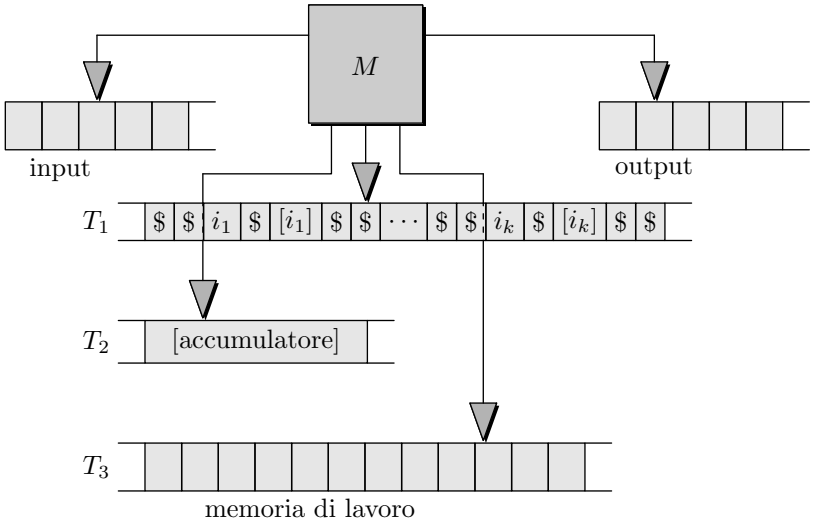


FIGURA 6.2 Macchina di Turing che simula una macchina a registri.

Le transizioni che la macchina $\mathcal{M}$ deve eseguire in corrispondenza di ogni istruzione della RAM nel caso peggiore costano un tempo pari alla massima lunghezza $\mathrm{t_{MAX}}$ raggiunta dal nastro T_1. D'altra parte, nel modello a costi logaritmici, se il nastro T_1 è lungo $\mathrm{t_{MAX}}$ vuol dire che la macchina a registri ha dovuto pagare un costo almeno uguale durante la sua computazione. Infatti, se sul nastro T_1 compare il numero d'ordine i di un registro, vuol dire che la RAM ha effettuato un accesso a tale registro ed ha quindi pagato almeno una volta un costo pari a $l(i)$. Analogamente, se compare sul nastro T_1 il contenuto di un registro $R[i]$ vuol dire che la RAM ha utilizzato come operando il contenuto di tale registro ed ha quindi pagato almeno una volta un costo pari a $l(R[i])$. Pertanto, in definitiva, se la macchina a registri opera con un costo T abbiamo che la Macchina di Turing opera in tempo $O(T \cdot \mathrm{t_{MAX}})$ e poiché $\mathrm{t_{MAX}} \leq T$ il costo totale è $O(\mathrm{T}^2)$. □

Grazie ai due teoremi precedenti possiamo affermare che qualunque funzione calcolabile secondo Turing è anche calcolabile mediante RAM e, viceversa, se sappiamo scrivere un programma RAM per una funzione essa è anche calcolabile secondo Turing. Questo risultato ci offre una ulteriore occasione di verificare la validità della tesi di Church-Turing.

Oltre a stabilire l'equivalenza del potere computazionale di macchine di Turing e RAM, un risultato di per sé molto significativo, i teoremi precedenti presentano un altro aspetto interessante, legato al costo con cui le simulazioni vengono effettuate. Infatti, poiché un problema (riconoscimento di un linguag-

gio, calcolo di una funzione etc.) viene generalmente considerato "trattabile", cioè risolubile in pratica, se esso è risolubile in tempo polinomiale, il fatto che macchine di Turing e RAM a costi logaritmici si possano simulare reciprocamente in tempo polinomiale significa che, al fine di determinare la trattabilità di un problema possiamo equivalentemente utilizzare macchine di Turing o RAM. A questo fine il fatto che si utilizzi un modello a costi logaritmici è fondamentale. È possibile infatti dimostrare il seguente risultato.

Teorema 6.3 *Nel modello a costi uniformi una RAM con l'operazione di moltiplicazione può calcolare in tempo polinomiale una funzione che richiede tempo esponenziale per essere calcolata da una Macchina di Turing.*

Esercizio 6.5 Dimostrare il teorema precedente.
[Suggerimento: Considerare i costi necessari a calcolare la funzione $f(n) = 2^{2^n}$ su una RAM a costi uniformi dotate di operatore di moltiplicazione e su Macchine di Turing].

Ciò significa, dunque, che le RAM a costi uniformi sono un modello di calcolo più potente delle Macchine di Turing nel senso che esse sono in grado di eseguire in tempo polinomiale calcoli che richiederebbero tempo esponenziale con una Macchina di Turing.

Esercizio 6.6 Mostrare che una RAM con tre soli registri può simulare una Macchina di Turing con un unico nastro di lavoro potenzialmente illimitato.
[Suggerimento: Assumere che il nastro di lavoro abbia un alfabeto binario e che il suo contenuto venga rappresentato con una coppia di interi in notazione binaria.]

6.4 Macchine a registri e linguaggi imperativi

Come si è detto precedentemente le macchine a registri sono un modello di calcolo particolarmente interessante perché il loro linguaggio di programmazione è basato sulle istruzioni fondamentali di un qualunque linguaggio di programmazione imperativo: istruzioni di input/output, trasferimenti tra registri, istruzioni aritmetiche elementari, etc. Di conseguenza i risultati e le proprietà che valgono per le RAM si possono facilmente estendere a macchine astratte programmate con linguaggi di programmazione più complessi ma sostanzialmente basati sulle stesse istruzioni.

Consideriamo, ad esempio, le tipiche istruzioni di un linguaggio assemblativo con istruzioni a due indirizzi (come ad esempio il linguaggio assemblativo della famiglia INTEL):

- istruzioni di trasferimento tra registri e memoria o tra registri di interfaccia di ingresso/uscita e memoria (ad esempio: MOV <locazione>

<registro>, MOV <registro> <locazione>, IN <porta> <registro> etc.);

- istruzioni aritmetiche (ad esempio ADD <locazione> <registro>, SUB <locazione> <registro>) eventualmente con operando immediato o con operando indiretto;

- istruzioni logiche (ad esempio AND <locazione> <registro>) o di rotazione e shift del contenuto di un registro (ad esempio ADL <registro>, che effettua lo shift circolare a sinistra del contenuto del registro);

- istruzioni di controllo (salti condizionati e non, arresto, reset etc.).

È evidente che, in modo del tutto analogo a come abbiamo definito la calcolabilità mediante RAM, possiamo definire le funzioni calcolabili con macchine programmate con questo tipo di linguaggio. È altrettanto chiaro però che, attraverso un procedimento di traduzione dei programmi così costruiti in programmi RAM, tutte le funzioni calcolabili con questo secondo modello sono calcolabili anche con le RAM.

Consideriamo ora un linguaggio ad alto livello, di tipo imperativo (Pascal, Fortran, C etc.). Ancora una volta potremo definire le funzioni calcolabili come quelle funzioni per le quali è possibile costruire un programma di calcolo in uno dei suddetti linguaggi. Tuttavia, anche in questo caso, l'esistenza di compilatori capaci di tradurre in linguaggio assemblativo i programmi scritti in tali linguaggi ci mostra immediatamente che le funzioni calcolabili con un linguaggio ad alto livello sono calcolabili con un programma assembler e, quindi, in definitiva, con un programma RAM.

Queste considerazioni forniscono un'altra conferma della tesi di Church-Turing. Infatti, dopo aver visto che, rispetto al calcolo di funzioni, le RAM e le macchine di Turing hanno lo stesso potere computazionale e dopo avere osservato che un qualunque programma in linguaggio imperativo può essere tradotto in un programma per RAM, possiamo dire a questo punto che un qualunque linguaggio di programmazione imperativo ha al più lo stesso potere delle macchine di Turing. Prima di concludere questa sezione desideriamo introdurre ancora un modello di calcolo basato su macchina a registri, ma questa volta si tratta di un modello molto elementare, in cui la semplificazione del linguaggio di programmmazione è spinta all'estremo limite: la *macchina a registri elementare* (MREL).

Una macchina a registri elementare, è dotata di un numero finito, ma arbitrario, di *registri*, celle di memoria che possono contenere un intero grande a piacere. I registri sono accessibili direttamente, in base al loro numero d'ordine. A differenza della RAM la macchina non prevede accumulatore. Invece, come nel caso della RAM, è previsto un contatore delle istruzioni (CI) che contiene il numero d'ordine della prossima istruzione che dovrà essere eseguita.

La macchina non ha nastri di ingresso e uscita ma per convenzione (e per semplicità) si assume che:

- i valori degli argomenti della funzione da calcolare $x_1, \ldots, x_n$ siano forniti all'inizio del calcolo nei registri R1, ..., Rn,
- tutti gli altri registri contengano il valore 0 all'inizio dell'esecuzione del programma,
- il valore della funzione $f(x_1, \ldots, x_n)$ calcolata dal programma, al termine dell'esecuzione del programma stesso si trovi in R1.

La Figura 6.3 fornisce una rappresentazione di una MREL.

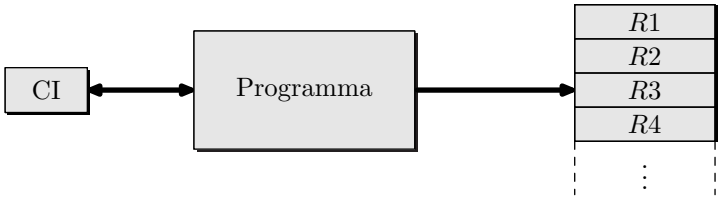


FIGURA 6.3 Rappresentazione fisica di una macchina a registri elementare.

Un *programma* per MREL consiste in una sequenza di *istruzioni* di tre soli tipi: incremento del contenuto di un registro, decremento del contenuto di un registro, salto condizionato in base al contenuto di un registro.

Di seguito forniamo una semplice descrizione in stile BNF del linguaggio delle MREL.

```
<programma>   ::= <istruzione> {<istruzione>}
<istruzione>  ::= R<intero positivo> := R<intero positivo> + 1 |
                  R<intero positivo> := R<intero positivo> − 1 |
                  IF R<intero positivo> = 0 THEN GOTO <etichetta>
<etichetta>   ::= <intero>
```

Si noti che, poiché il contenuto di un registro è un intero non negativo quando si decrementa il contenuto di un registro che è già pari a 0 il contenuto rimane 0.

Inoltre l'istruzione di salto condizionato IF Ri = 0 THEN GOTO <etichetta> viene interpretata nel seguente modo: se il contenuto del registro Ri è uguale a 0 allora si salta all'istruzione il cui numero d'ordine è dato dall'etichetta, altrimenti si prosegue con l'esecuzione dell'istruzione seguente.

Per convenzione il programma termina se si effettua un salto all'istruzione 0. In ogni altra circostanza (cioè quando il programma cicla o il programma salta ad un'istruzione non esistente ma diversa da 0) diciamo che il calcolo non è definito.

Esempio 6.2 Vediamo un esempio di programma MREL per il calcolo della somma di due interi $f(x, y) = x + y$:

IF R2 = 0 THEN GOTO 0
R2 := R2 − 1
R1 := R1 + 1
IF R3 = 0 THEN GOTO 1

Come si vede il fatto che il registro R3 sia inizializzato a zero ci consente di simulare un salto incondizionato.

Esercizio 6.7 Definire programmi per MREL corrispondenti ai seguenti programmi per RAM:

```
LOAD   1
ADD    2
STORE  1
```

```
LOAD   1
MULT   2
STORE  1
```

Con considerazioni analoghe a quelle fatte precedentemente non è difficile convincersi che se una funzione è calcolabile con una macchina a registri essa è calcolabile anche con una MREL. Infatti ogni passo di una RAM può essere sostituito da uno o più passi di una MREL. D'altra parte è interessante osservare che un linguaggio estremamente elementare come quello delle MREL continua ancora a possedere lo stesso potere computazionale di una Macchina di Turing o di un programma C.

Nella Sezione 6.6 utilizzeremo ancora le macchine a registri elementari per formalizzare in modo preciso i concetti di stato, computazione, etc. e fornire una sorta di "forma normale" per le funzioni calcolabili.

6.5 Funzioni ricorsive

La caratterizzazione delle funzioni calcolabili è stata data, fino a questo punto, attraverso la definizione delle classi di macchine (macchine di Turing, macchine a registri etc.) che vengono utilizzate per il loro calcolo. In questa sezione

introduciamo il formalismo delle funzioni ricorsive, che consente di dare una caratterizzazione matematica delle funzioni calcolabili attraverso l'individuazione di alcune funzioni base (funzioni calcolabili estremamente elementari) e la costruzione di nuove funzioni calcolabili, ottenute applicando opportuni operatori a funzioni definite precedentemente.

Le funzioni ricorsive sono state introdotte da Gödel e Kleene negli anni '30 (vedi Capitolo A), ma il loro studio si è sviluppato anche successivamente all'avvento dei primi calcolatori e dei primi linguaggi di programmazione ad alto livello finché, alla fine degli anni '50, esse hanno ispirato direttamente i primi linguaggi di programmazione funzionale.

Nel seguito del capitolo vedremo da un lato la relazione delle funzioni ricorsive con i modelli di calcolo basati sulle macchine a registri e con i relativi linguaggi di tipo imperativo e, dall'altro lato, come esse sono collegate a modelli e linguaggi di tipo funzionale, come il formalismo di McCarthy e il LISP.

In questa sezione, per descrivere in modo esatto la definizione di una funzione, si farà uso della *notazione lambda*, che permette di specificare l'insieme delle variabili sulle quali la funzione si intende definita. Per mezzo di tale formalismo, una funzione di n variabili $x_1, x_2, \ldots, x_n$, associata ad una espressione algebrica $f(x_1, \ldots, x_n)$ in forma chiusa, sarà descritta come $\lambda x_1 \lambda x_2 \ldots \lambda x_n . f(x_1, \ldots, x_n)$. Ad esempio, per indicare una corrispondenza che ad ogni coppia di valori x_1, x_2 associa la somma degli stessi, scriveremo $\lambda x_1 \lambda x_2 . x_1 + x_2$.

Consideriamo ora l'insieme di tutte le funzioni intere:

$$\mathcal{F} = \{f \mid f : \mathbb{N}^n \to \mathbb{N}, n \geq 0\}$$

In generale con la notazione $f^{(n)}$ intendiamo esplicitare che il numero di argomenti della funzione f è n. Se $n = 0$ la funzione è una costante. Per individuare all'interno dell'insieme $\mathcal{F}$ classi di funzioni che corrispondano ad un concetto intuitivo di calcolabilità procederemo nel seguente modo: definiremo anzitutto alcune *funzioni base* così elementari che apparirà del tutto naturale supporre di poterle calcolare, noti i valori degli argomenti. Successivamente definiremo degli *operatori*, ovvero, con termine più informatico, dei *costrutti*, che consentiranno di definire nuove funzioni a partire dalle funzioni già disponibili.

Definizione 6.2 *Definiamo come* funzioni base *le seguenti funzioni:*

i) $0^{(n)} = \lambda x_1 \ldots \lambda x_n . 0$, $n \geq 0$, funzioni zero,

ii) $S = \lambda x . x + 1$, funzione successore,

iii) $U_i^{(n)} = \lambda x_1 \ldots \lambda x_n . x_i$, funzioni identità o selettive.

La prima classe di funzioni che definiamo è la classe delle funzioni *ricorsive primitive*.

Definizione 6.3 *La classe delle* funzioni ricorsive primitive $\mathcal{P}$ *è la più piccola classe che contiene le funzioni base e inoltre:*

i) *se le funzioni* $h^{(m)}, g_1^{(n)}, \ldots, g_m^{(n)}$ *sono ricorsive primitive allora anche la funzione* $f^{(n)}$ *definita mediante* l'operatore di composizione*:*

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

è ricorsiva primitiva;

ii) *se le funzioni* $h^{(n+2)}, g^{(n)}$ *sono ricorsive primitive allora anche la funzione* $f^{(n+1)}$ *definita mediante* l'operatore di ricursione primitiva*:*

- $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$,
- $f(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$

è ricorsiva primitiva.

Esempio 6.3 Le seguenti funzioni sono ricorsive primitive:

1. *incr2* $= \lambda x \lambda y.x + 2$. Tale funzione può essere espressa utilizzando le funzioni base e la sola composizione come *incr2*$= S(S(U_1^{(2)})$.
2. *somma* $= \lambda x \lambda y.x + y$. Tale funzione può essere espressa, utilizzando la ricursione primitiva, nel modo seguente:
   - $somma(x, 0) = x$
   - $somma(x, y+1) = somma(x, y) + 1$

   In questo caso la funzione g della Definizione 6.3 ha un solo argomento e coincide con la funzione selettiva $U_1^{(1)}$; la funzione h ha tre argomenti ed è la funzione $\lambda x_1 \lambda x_2 \lambda x_3 . S(U_3^{(3)}(x_1, x_2, x_3))$, ottenuta per composizione della funzione successore e della funzione selettiva $U_3^{(3)}$.

Esercizio 6.8 Dimostrare che la funzione prodotto $\lambda x \lambda y.x * y$ è ricorsiva primitiva.

Esercizio 6.9 Dimostrare che la funzione predecessore $\lambda x.x - 1$, che si assume restituisca il valore 0 se $x = 0$, è ricorsiva primitiva.

Esercizio 6.10 Sia p una funzione ricorsiva primitiva con valori 0 e 1 e siano g ed h due funzioni ricorsive primitive, dimostrare che la funzione f definita per casi nel seguente modo:

$$f(x) = \begin{cases} \text{se } p(x) = 1 \text{ allora } g(x) \\ \text{altrimenti } h(x) \end{cases}$$

è ricorsiva primitiva.

Si noti che frequentemente la definizione di una funzione in termini ricorsivi ha una struttura più semplice, ed è basata sul seguente schema. Se le funzioni $h^{(2)}, g^{(1)}$ sono ricorsive primitive e c è una costante, allora anche la funzione $f^{(1)}$ definita come:

- $f(0) = c$
- $f(x+1) = h(g(x), f(x))$

è ricorsiva primitiva.

Esercizio 6.11 Dimostrare l'asserzione precedente, e cioé che una funzione definita secondo lo schema semplificato è in effetti ricorsiva primitiva.

Esercizio 6.12 Dimostrare, applicando lo schema semplificato, che la funzione fattoriale $\lambda x.x!$ è ricorsiva primitiva.

Esercizio 6.13 Dimostrare, applicando lo schema semplificato, che la funzione esponenziale $\lambda x.2^x$ è ricorsiva primitiva.

Come si vede, la definizione di ricursione primitiva è abbastanza comprensiva; tuttavia esistono funzioni per le quali è possibile dare una definizione di tipo ricorsivo (che permette quindi una univoca sequenza di passi di calcolo) ma che non rientrano nella classe $\mathcal{P}$. Un esempio è la funzione di tre argomenti così definita:

$$
\begin{aligned}
&f(x,y,0) = x+y\\
&f(x,y,1) = x\cdot y\\
&f(x,y,2) = x^y\\
&\text{e, per } n \geq 2,\\
&f(x,y,n+1) = \begin{cases} 1 \text{ se } y = 0 \\ \overbrace{f(x, f(x, \ldots, f(}^{y \text{ volte}} x,0,n), \ldots, n), n) \text{ altrimenti} \end{cases}
\end{aligned}
$$

Il metodo di definizione della funzione precedente è basato su una doppia ricursione: infatti, come si vede, per ogni valore di n la funzione $\lambda x \lambda y.f(x,y,n)$ è ricorsiva primitiva rispetto alla variabile y, ed inoltre la funzione $\lambda x \lambda y.f(x,y,n+1)$ è costruita utilizzando la funzione $\lambda x \lambda y.f(x,y,n)$.

Ad esempio, la funzione

$$
f(x,y,3) = \begin{cases} 1 & \text{se } y = 0 \\ \underbrace{x^{x^{\cdot^{\cdot^{x}}}}}_{y \text{ volte}} & \text{altrimenti} \end{cases}
$$

è una funzione ricorsiva primitiva ottenuta per iterazione dell'esponenziale (corrispondente a $\lambda x \lambda y.f(x, y, 2)$).

La definizione in termini finiti di f è, per ogni $x \geq 0$, $y \geq 0$, $n \geq 0$:

$$
\begin{aligned}
&f(x, 0, 0) = x \\
&f(x, y + 1, 0) = f(x, y, 0) + 1 \\
&f(x, 0, 1) = 0 \\
&f(x, 0, n + 2) = 1 \\
&f(x, y + 1, n + 1) = f(x, f(x, y, n + 1), n)
\end{aligned}
$$

È possibile dimostrare che la funzione f sopra definita, detta *funzione di Ackermann*, non è ricorsiva primitiva, pur essendo essa stessa definita in termini ricorsivi. Da ciò deriva che la ricursione primitiva non è sufficientemente potente da descrivere tutte le funzioni definibili in modo ricorsivo.

Introduciamo quindi ora un nuovo operatore che ci permetta di estendere la classe delle funzioni ricorsive primitive e di cogliere in modo più ampio il concetto di calcolabilità.

Definizione 6.4 *Data una funzione $g^{(n+1)}$, la funzione $f^{(n)}$ si dice definita mediante* l'operatore di minimalizzazione μ *se $f(x_1, \ldots, x_n)$ assume come valore il minimo t tale che $g(x_1, \ldots, x_n, t) = 0$; se $g(x_1, \ldots, x_n, t) \neq 0$ per tutti i valori di t, allora $f(x_1, \ldots, x_n)$ è indefinita.*

Utilizzeremo la notazione $f = \lambda x_1 \ldots \lambda x_n.\mu t(g(x_1, \ldots, x_n, t) = 0)$ per indicare che la funzione f è definita applicando l'operatore di minimalizzazione sulla funzione g.

Definizione 6.5 *La classe delle* funzioni ricorsive *(dette anche* ricorsive generali *o* ricorsive parziali*) $\mathcal{R}$ è la più piccola classe che contiene le funzioni base, ed è chiusa rispetto a* composizione*,* ricursione primitiva *e* minimalizzazione*.*

Esempio 6.4 La funzione $\lambda n.\sqrt{n}$ è ricorsiva in quanto può essere definita come $\sqrt{n} = \mu t((n + 1) - (t + 1) * (t + 1) = 0)$.

Si noti che anziché utilizzare il predicato $g(x_1, \ldots, x_n, t) = 0$ per determinare il valore fino al quale deve essere incrementata la variabile t, possiamo utilizzare un qualunque altro predicato $T(x_1, \ldots, x_n, t)$, purché esista una funzione ricorsiva g per cui $T(x_1, \ldots, x_n, t)$ è vero se e solo se $g(x_1, \ldots, x_n, t) = 0$.

Esempio 6.5 **i)** La funzione $\lambda n.\sqrt{n}$ può essere così definita:
$\sqrt{n} = \mu t((t + 1) * (t + 1) > n)$

ii) La funzione $\lambda n.md(n)$ che associa ad ogni intero n il minimo intero maggiore di 1 che divide n può essere così definita:
$md = \mu t((t > 1) \wedge resto(n, t) = 0)$

Esercizio 6.14 Utilizzando l'operatore μ, dimostrare che la seguente funzione è ricorsiva: $\lambda n.n$-*esimo numero primo*.

Esercizio 6.15 Sia $g(x,t)$ una funzione che, per ogni valore di x, si annulla per alcuni valori di $t > 0$ e sia $\bar{t}(x)$ il minimo di tali valori. Definire una funzione primitiva ricorsiva $g'(x,t)$ tale che $g'(x,t) = 0$ se $t < \bar{t}(x)$ e $g'(x,t) \geq 1$ altrimenti.

Esercizio 6.16 Sia $g(x,t)$ una funzione che, per ogni valore di x, si annulla per alcuni valori di $t > 0$ e sia $\bar{t}(x)$ il minimo di tali valori. Definire una funzione primitiva ricorsiva $g''(x,t)$ tale che $g''(x,t) = 1$ se $t = \bar{t}(x)$ e $g''(x,t) = 0$ altrimenti.

Esercizio 6.17 Dimostrare che se le funzioni g e h sono ricorsive primitive lo è anche la funzione f definita nel seguente modo (operatore μ *limitato*):
$f = \lambda x_1 \ldots \lambda x_n.\mu t \leq h(x_1, \ldots, x_n)(g(x_1, \ldots, x_n, t) = 0)$ cioè $f(x_1, \ldots, x_n)$ assume come valore il minimo valore di t tale che $g(x_1, \ldots, x_n, t) = 0$ se esiste un valore di $t \leq h(x_1, \ldots, x_n)$ tale che $g(x_1, \ldots, x_n, t) = 0$; se per nessun valore di $t \leq h(x_1, \ldots, x_n) g(x_1, \ldots, x_n, t) = 0$, allora $f(x_1, \ldots, x_n)$ assume un valore di default, ad esempio 0.
[Suggerimento: Utilizzare i due esercizi precedenti].

Le funzioni ricorsive sono dunque funzioni numeriche che hanno la caratteristica o di essere estremamente elementari (funzioni zero, successore, selettive) o di essere ottenibili mediante un numero finito di applicazioni degli operatori di composizione, di ricursione primitiva, di minimalizzazione a partire dalle funzioni base. Come si vede immediatamente la parzialità delle funzioni ricorsive deriva dall'utilizzazione dell'operatore μ, infatti mentre gli altri operatori portano da funzioni totali a funzioni totali, utilizzando l'operatore μ può accadere che il valore della funzione non sia definito per alcuni valori degli argomenti.

6.6 Funzioni ricorsive e linguaggi imperativi

Come si è visto, la definizione delle funzioni ricorsive si basa su costruzioni di tipo matematico che, almeno esplicitamente, non fanno riferimento a particolari modelli di macchina. D'altra parte, per chi abbia dimestichezza con la programmazione risulta subito evidente che le funzioni ricorsive sono calcolabili con le macchine a registri. Per verificare questa proprietà è sufficiente mostrare come si possono calcolare le funzioni ricorsive utilizzando un linguaggio ad alto livello (ad esempio C) e ricordare quanto detto precedentemente sulla possibilità di compilare il codice di un programma scritto in un linguaggio imperativo ad alto livello per una macchina a registri.

Teorema 6.4 *Data una qualunque funzione ricorsiva è possibile definire un programma C che la calcola.*

Esercizio 6.18 Dimostrare il precedente enunciato. Nota bene: al fine di rendere più significativo l'esercizio si raccomanda di utilizzare solo i costrutti fondamentali del C (if-else, while, do-while) e di utilizzare solo funzioni che non facciano uso della ricursione.

Quanto sopra ci dice dunque che le funzioni ricorsive possono essere calcolate con macchine a registri. Più complesso è dimostrare il risultato opposto e cioè che ogni funzione calcolata da una macchina a registri è ricorsiva. Quando avremo dimostrato quest'ultimo risultato avremo stabilito che effettivamente le funzioni calcolabili con macchine a registri e linguaggi di programmazione imperativi coincidono con le funzioni ricorsive.

Per giungere a questo obiettivo in modo più facilmente comprensibile, faremo riferimento, anziché a RAM, a macchine a registri elementari, e mostreremo come sia possibile codificare mediante interi gli stati assunti da una macchina a registri elementare, oltre che le computazioni da essa eseguite.

Introduciamo anzi tutto le seguenti funzioni di creazione di n-ple e di estrazione di elementi da una n-pla, che ci consentiranno di esprimere mediante numeri interi i suddetti concetti di stato e di computazione. In particolare indichiamo come funzioni di *creazione di n-ple* le funzioni P_k che associano biunivocamente un intero ad una n-pla di interi, e che sono così definite:

$$P_n = \lambda x_1 \ldots \lambda x_n [2^{x_1} \cdot \ldots \cdot {p_n}^{x_n}]$$

dove p_n è l'n-esimo numero primo ($p_1 = 2$).

Definiamo inoltre le funzioni di *estrazione di elementi da una n-pla* come

$$K_i = \lambda z[\text{esponente di } p_i \text{ nella decomposizione in fattori primi di } z]$$

dove, ancora, p_i è l'i-esimo numero primo ($p_1 = 2$).

Per tali funzioni useremo la notazione $\langle x_1, \ldots, x_n \rangle = P_n(x_1, \ldots, x_n)$ e $(z)_i = K_i(z)$, rispettivamente; inoltre introduciamo la notazione

$$M(t) = \text{if } t = 0 \text{ then } 0 \text{ else } \max\{n \mid p_n \text{ divide } t\}$$

per indicare l'indice del massimo numero primo che divide un intero dato.

Valgono le proprietà:

$$K_i(P_n(x_1, \ldots, x_n)) = x_i \quad \text{se } i \leq n, \text{ 0 altrimenti;}$$
$$P_n(K_1(z), \ldots, K_n(z)) = z \quad \text{se } n = M(z).$$

Esercizio 6.19 Dimostrare che P_n e K_i sono ricorsive primitive, per ogni n ed ogni i.

A questo punto, utilizzando la notazione introdotta, possiamo definire formalmente lo stato e la computazione effettuata da una macchina a registri elementare.

Definizione 6.6 *Uno* stato *di una macchina a registri elementare con programma π, il quale utilizza m registri ed è costituito da l istruzioni, è rappresentato da un intero:*

$$s = \langle i, \langle r_1, \ldots, r_m \rangle \rangle$$

dove i $(0 \leq i \leq l)$ è il numero d'ordine dell'istruzione da eseguire, e $r_1, \ldots, r_m \geq 0$ sono gli interi contenuti nei registri.

Esempio 6.6 Gli stati attraversati dalla computazione del programma della somma (Esempio 6.2) con input 3 e 2, sono codificati nel seguente modo:

$$\begin{array}{lllll}
s_1 & \langle 1, \langle 3,2,0 \rangle \rangle & = & 2 \cdot 3^{2^3 3^2} & = & 2 \cdot 3^{72} \\
s_2 & \langle 2, \langle 3,2,0 \rangle \rangle & = & 2^2 \cdot 3^{72} & & \\
s_3 & \langle 3, \langle 3,1,0 \rangle \rangle & = & 2^3 \cdot 3^{2^3 3} & = & 2^3 \cdot 3^{24} \\
s_4 & \langle 4, \langle 4,1,0 \rangle \rangle & = & 2^4 \cdot 3^{2^4 3} & = & 2^4 \cdot 3^{48} \\
s_5 & \langle 1, \langle 4,1,0 \rangle \rangle & = & 2 \cdot 3^{48} & & \\
s_6 & \langle 2, \langle 4,1,0 \rangle \rangle & = & 2^2 \cdot 3^{48} & & \\
s_7 & \langle 3, \langle 4,0,0 \rangle \rangle & = & 2^3 \cdot 3^{2^4} & = & 2^3 \cdot 3^{16} \\
s_8 & \langle 4, \langle 5,0,0 \rangle \rangle & = & 2^4 \cdot 3^{2^5} & = & 2^4 \cdot 3^{32} \\
s_9 & \langle 1, \langle 5,0,0 \rangle \rangle & = & 2 \cdot 3^{32} & & \\
s_{10} & \langle 0, \langle 5,0,0 \rangle \rangle & = & 2^{32} \quad . & &
\end{array}$$

Consideriamo ora una computazione eseguita da una macchina a registri elementare con m registri $r_1, \ldots, r_m$, programma π ed input $x_1, \ldots, x_n$ $(n \leq m)$, ed assumiamo che tale computazione sia data dalla sequenza di stati $s_1, \ldots, s_t$. Lo stato iniziale della computazione è $s_1 = \langle 1, \langle x_1, \ldots, x_n, 0, \ldots, 0 \rangle \rangle$ e gli stati successivi nella computazione devono essere correlati tra loro da una funzione R_π con valori $0, 1$, dipendente dal programma π, e così definita: $R_\pi(s_p, s_q) = 1$, dove $s_p = \langle i_p, \langle r_1^p, \ldots, r_m^p \rangle \rangle$ ed $s_q = \langle i_q, \langle r_1^q, \ldots, r_m^q \rangle \rangle$, se e solo se valgono le seguenti proprietà.

a) se i_p corrisponde all'istruzione $\mathrm{R}j = \mathrm{R}j + 1$

allora $i_q = i_p + 1$ ed $r_k^q = \begin{cases} r_k^p + 1 & \text{se } k = j \\ r_k^p & \text{altrimenti} \end{cases}$

b) se i_p corrisponde all'istruzione $\mathrm{R}j = \mathrm{R}j \dot{-} 1$

allora $i_q = i_p + 1$ ed $r_k^q = \begin{cases} r_k^p \dot{-} 1 & \text{se } k = j \\ r_k^p & \text{altrimenti} \end{cases}$

c) se i_s : IF $\mathrm{R}j = 0$ GOTO h

allora $i_q = \begin{cases} h & \text{se } r_j^p = 0 \\ i_p + 1 & \text{altrimenti} \end{cases}$ ed $r_k^q = r_k^p$ per ogni k.

d) Se i_p è un numero maggiore del numero di istruzione del programma allora $i_q = i_p$ ed $r_k^q = r_k^p$ per ogni k.
(si noti che questo ultimo caso comporta l'esecuzione di un ciclo infinito).

Chiaramente, una computazione è finita se e solo se il programma si arresta in modo legale, vale a dire se si verifica un salto all'istruzione 0. In tal caso si ha uno *stato finale*:

$$s_F = \langle 0, \langle r_1^F, \ldots, r_m^F \rangle \rangle$$

e il valore della funzione calcolata è r_1^F .

Definizione 6.7 *Sia $s_1, \ldots, s_t$ una computazione finita eseguita da un macchina a registri elementare con programma π. Tale computazione può essere codificata mediante un intero $c = 2^{s_1} 3^{s_2} \ldots p_t^{s_t}$.*

Esempio 6.7 La computazione del programma della somma (Esempio 6.2) con input 3 e 2, è codificata nel seguente modo:

$$2^{(2\cdot 3^{72})} \cdot 3^{(2^2\cdot 3^{72})} \cdot 5^{(2^3\cdot 3^{24})} \cdot 7^{(2^4\cdot 3^{48})} \cdot \ldots \cdot 29^{(2^{32})}$$

Dimostriamo ora il seguente Lemma.

Lemma 6.5 *Dato un programma π, la funzione T_π[2] così definita:*

$$T_\pi(x_1, \ldots, x_n, c) = \begin{cases} 1 & \text{se } c = \langle s_1, \ldots, s_F \rangle \text{ e } s_1, \ldots, s_F \text{ è la} \\ & \text{computazione di } \pi \text{ con input} \\ & x_1, \ldots, x_n \ ; \\ \\ 0 & \text{altrimenti} \end{cases}$$

è ricorsiva primitiva.

Dimostrazione. Definiamo anzitutto le seguenti funzioni con valori $0, 1$, che corrispondono rispettivamente alla verifica che la computazione c inizi in uno stato iniziale e termini in uno stato finale.

A) $I_\pi(x_1, \ldots, x_n, s) = 1$ se e solo se $s = \langle 1, \langle x_1, \ldots, x_n \rangle \rangle$.

B) $F_\pi(s) = 1$ se e solo se $(s)_1 = 0$, cioé se s è uno stato finale.

[2]Il predicato corrispondente alla funzione T_π è generalmente denominato *predicato di Kleene*.

Ricordando la definizione della funzione R_π introdotta precedentemente, possiamo a questo punto definire la funzione T_π nel seguente modo:

$$T_\pi(x_1,\ldots,x_n,c) = I_\pi(x_1,\ldots,x_n,(c)_1) \cdot \prod_{p=1}^{M(c)-1} R_\pi((c)_p,(c)_{p+1}) \cdot F_\pi((c)_{M(c)})$$

In altre parole, dato t si deve controllare che tutti gli esponenti della sua decomposizione in primi, $(c)_1,(c)_2,\ldots,(c)_p,(c)_{p+1},\ldots,(c)_{M(c)}$, costituiscano una sequenza di calcolo di π con input $x_1,\ldots,x_n$.

Essendo ricorsive primitive le funzioni: Π, M, P_n, K_i, segue che T_π è ricorsiva primitiva. □

Si noti che I_π dipende da π solo nel numero di registri di cui si deve verificare la corretta inizializzazione; F_π non dipende affatto da π. Al contrario, R_π dipende pesantemente dalle singole istruzioni di π.

Esercizio 6.20 Completare la dimostrazione verificando che tutte le funzioni considerate nella dimostrazione stessa sono ricorsive primitive.

Esempio 6.8 La funzione T_π associata al programma di somma π introdotto nell'Esempio 6.2 è ottenuta prendendo:

$$I_\pi(x_1,x_2,s) = \begin{cases} 1 & \text{se } (s)_1 = 1 \quad \& \quad ((s)_2)_1 = x_1 \\ & \qquad\qquad\ \ \& \quad ((s)_2)_2 = x_2 \\ 0 & \text{altrimenti} \end{cases}$$

$$F_\pi(a) = 1 \dot{-} (a)_1$$

$R_\pi(a,b) = 1$ se solo se $M(a) = M(b) = 2$, $M((a)_2) = M((b)_2) = 2$ e se:

$$\begin{aligned} (a)_1 = 1 \quad &\wedge \quad \{[((a)_2)_2) = 0 \ \&\ (b)_1 = 0] \vee [((a)_2)_2 \neq 0 \ \&\ (b)_1 = 2]\} \\ &\wedge \quad ((a)_2)_1 = ((b)_2)_1 \\ &\wedge \quad ((a)_2)_2 = ((b)_2)_2 \\ &\wedge \quad ((a)_2)_3 = ((b)_2)_3 \end{aligned}$$

oppure se:

$$\begin{aligned} (a)_1 = 2 \quad &\wedge \quad ((a)_2)_1 = ((b)_2)_1 \\ &\wedge \quad ((a)_2)_2 \dot{-} 1 = ((b)_2)_2 \\ &\wedge \quad ((a)_2)_3 = ((a)_2)_3 \\ &\wedge \quad (b)_1 = 3 \end{aligned}$$

oppure se:

$$\begin{aligned} (a)_1 = 3 \quad &\wedge \quad ((a)_2)_1 + 1 = ((b)_2)_1 \\ &\wedge \quad ((a)_2)_2 = ((b)_2)_2 \\ &\wedge \quad ((a)_2)_3 = ((b)_2)_3 \\ &\wedge \quad (b)_1 = 4 \end{aligned}$$

oppure se:

$$\begin{array}{lll} (a)_1 = 4 & \wedge & (((a)_2)_3 = 0 \ \& \ (b)_1 = 1) \vee (((a)_2)_3 \neq 0 \ \& \ (b)_1 = 5) \\ & \wedge & ((a)_2)_1 = ((b)_2)_1 \\ & \wedge & ((a)_2)_2 = ((b)_2)_2 \\ & \wedge & ((a)_2)_3 = ((b)_2)_3 \end{array}$$

Nell'esempio risulta chiaramente che I_π, R_π, F_π sono funzioni ricorsive primitive.

A questo punto possiamo formulare il teorema seguente.

Teorema 6.6 *Dato un programma π per una macchina a registri elementare, la funzione $\lambda x_1 \ldots \lambda x_n . f(x_1, \ldots, x_n)$ calcolata da π è una funzione ricorsiva parziale.*

Dimostrazione. Sia T_π la funzione definita nel Lemma precedente, in corrispondenza al programma π assegnato. Possiamo definire la funzione f nel seguente modo:

$$f(x_1, \ldots, x_n) = U(\mu c[T_\pi(x_1, \ldots, x_n, c) = 1])$$

dove $U(c) = (((c)_{M(c)})_2)_1$ è la funzione che estrae $s_F = (c)_{M(c)}$ da c ed estrae x_1^F da s_F. □

Il teorema precedente è particolarmente interessante per vari motivi. Innanzitutto consente di dimostrare che la classe delle funzioni ricorsive coincide con la classe delle funzioni calcolabili mediante RAM, e quindi con quella delle funzioni calcolabili secondo Turing, offrendo ancora una ulteriore conferma della testi di Church-Turing. Inoltre, esso offre una particolare caratterizzazione delle funzioni ricorsive (e quindi delle funzioni calcolabili in generale), mostrando come esse siano ottenibili combinando in modo opportuno due funzioni ricorsive primitive e utilizzano una sola volta l'operatore di minimalizzazione.

6.7 Funzioni ricorsive e linguaggi funzionali

La teoria delle funzioni ricorsive, oltre a costituire un significativo approccio al concetto di calcolabilità, alternativo a quelli basati su modelli di macchina, quali le Macchine di Turing e le Macchine a registri, ha avuto un impatto importante sullo sviluppo dei linguaggi di programmazione. In particolare, vari concetti impliciti nel modello delle funzioni ricorsive sono stati adottati nella definizione dei *linguaggi di programmazione funzionali*. Tali linguaggi, similmente a quanto avviene nel modello delle funzioni ricorsive, permettono di descrivere funzioni senza fare riferimento esplicito ai modelli di macchina che eseguono il calcolo. In tale contesto, il concetto di esecuzione di una computazione viene sostituito da quello di valutazione di una espressione, così

come avviene quando una funzione viene descritta mediante una espressione matematica in forma chiusa.

Nella presente sezione introduciamo un semplice linguaggio di programmazione funzionale il quale, come mostreremo, consente la definizione di tutte le funzioni ricorsive. Al fine di introdurre il linguaggio stesso, presentiamo un formalismo, detto *formalismo di McCarthy*, che descrive in modo particolarmente sintetico ed efficace la struttura sintattica che linguaggi di tipo funzionale possono avere, e che è alla base sia del linguaggio di programmazione funzionale che descriveremo in questa sezione, che del più noto di tali linguaggi, il linguaggio *LISP*.

Siano dati i seguenti insiemi di simboli:

$X = \{x_1, x_2, \ldots\}$: insieme, eventualmente infinito, di *simboli di variabile*

$B = \{b_1, b_2, \ldots\}$: insieme, eventualmente infinito, di *simboli di funzione base*

$F = \{F_1, F_2, \ldots\}$: insieme, eventualmente infinito, di *simboli di variabile funzione*

$a : \mathbb{N}^+ \rightarrow \mathbb{N}$: arità delle funzioni in B

$A : \mathbb{N}^+ \rightarrow \mathbb{N}^+$: arità delle funzioni in F.

Utilizzando gli insiemi suddetti e le arità delle funzioni, possiamo introdurre la seguente definizione.

Definizione 6.8 *Un* termine *su* $< X, B, F >$ *può essere definito induttivamente nel modo seguente:*

1. *se* $x \in X$ *allora* x *è un termine*
2. *se* $b_j \in B$, *con arità* $a(j) = n \geq 0$, *e* $t_1, \ldots, t_n$ *sono termini, allora* $b(t_1, \ldots, t_n)$ *è un termine*
3. *se* $F_i \in F$, *con arità* $A(i) = m > 0$, *e* $t_1, \ldots, t_m$ *sono termini*
4. *allora* $F_i(t_1, \ldots, t_m)$ *è un termine.*

Se la arità di una funzione base $b_k()$ *è* $a(k) = 0$, *allora tale funzione viene detta* costante *e viene rappresentata semplicemente come* b_k.

A questo punto possiamo definire il concetto di *schema di programma* su X, B, F.

Definizione 6.9 *Dati tre insiemi* X, B, F, *uno* schema di programma *sulla terna* $< X, B, F >$ *è costituito da una sequenza finita di coppie di termini del tipo*

$$F_i(x_1, \ldots, x_n) \Leftarrow t_j$$

dove $F_i \in F$ *e* t_j *è un generico termine su* $< X, B, F >$*, seguita da un termine del tipo*

$$F_h(b_1, \ldots, b_m)$$

dove $F_h \in F$*,* $A(h) = m$ *e* $b_1, \ldots, b_m \in B$ *sono costanti.*

Nell'interpretazione più naturale di uno schema di programma la sequenza di coppie può essere vista come una sequenza di dichiarazioni di funzioni, mentre il termine contenente la costante b può essere visto come la richiesta di determinare il valore della funzione F_h sugli argomenti $b_1, \ldots, b_m$.

Esempio 6.9 Siano b_1, b_2, b_3, b_4 rispettivamente di arità $3, 1, 1, 0$, ed F_1, F_2 di arità $2, 1$. Un esempio di schema di programma su $X = \{X_1, X_2\}$, $B = \{b_1, b_2, b_3, b_4\}$ ed $F = \{F_1, F_2\}$ è il seguente:

$$\begin{array}{lcl} F_1(x_1, x_2) & \Leftarrow & b_1(x_1, F_2(b_2(x_2)), x_1) \\ F_2(x_1) & \Leftarrow & b_2(F_1(x_1, b_3(x_1))) \\ F_2(b_4) & & \end{array}$$

Per passare dal formalismo astratto di McCarthy a un linguaggio specifico è sufficiente definire l'insieme delle funzioni base che deve essere messo in corrispondenza con i simboli dell'insieme B. In tal modo, possiamo definire un semplice linguaggio di tipo funzionale, denominato *SLF*, per la definizione di funzioni intere, scegliendo le funzioni base come segue.

Sia $B = \mathcal{N} \cup \{S, P, \textit{if-then-else}\}$, in cui $\mathcal{N}$ rappresenta tutti i simboli di costante corrispondenti ai naturali, e indichiamo con S la funzione successore (di arità 1), con P la funzione predecessore (di arità 1) e con *if-then-else* l'usuale costrutto di alternativa (di arità 3), definito nel modo seguente

if-then-else $(t_1, t_2, t_3) = t_2$ se il valore di t_1 è 0, t_3 altrimenti.

Esempio 6.10 Un programma per il calcolo della sottrazione tra due interi non negativi, ad esempio gli interi 4 e 3, può essere formulato, nel linguaggio SLF, nel modo seguente:

$$\begin{array}{lcl} \textsc{sub}(x, y) & \Leftarrow & \textit{if-then-else}(y, x, P(\textsc{sub}(x, P(y)))) \\ \textsc{sub}(4, 3) & & \end{array}$$

Come mostrato dal teorema seguente, il potere computazionale del linguaggio *SLF* è equivalente a quello di qualunque altro formalismo per il calcolo delle funzioni ricorsive (macchina di Turing, RAM, linguaggi di programmazione imperativi, etc.).

Teorema 6.7 *Il linguaggio SLF consente la definizione di tutte le funzioni ricorsive.*

Dimostrazione. Le funzioni base possono essere espresse in *SLF*; infatti, il successore fa parte delle funzioni base di *SLF*, mentre le funzioni *zero*, $0_i^{(n)}$, e *selettive*, $U_i^{(n)}$, possono essere definite rispettivamente come segue:

$$F(x_1, \ldots, x_n) \Leftarrow 0$$

$$F(x_1, \ldots, x_n) \Leftarrow x_i$$

Se la funzione F è definita per composizione delle funzioni $h, g_1, \ldots, g_m$, essa può essere definita in *SLF* come segue:

$$F(x_1, \ldots, x_n) \Leftarrow h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

Se la funzione F è definita per ricursione primitiva a partire dalle funzioni g ed h, essa può essere definita in *SLF* come segue:

$$F(x_1, \ldots, x_n, y) \Leftarrow$$

$$\Leftarrow \textit{if-then-else}(y, g(x_1, \ldots, x_n), h(x_1, \ldots, x_n, y, F(x_1, \ldots, x_n, P(y))))$$

Se la funzione F è definita per minimalizzazione a partire dalla funzione g, essa può essere definita in *SLF* come segue:

$$F(x_1, \ldots, x_n) \Leftarrow G(x_1, \ldots, x_n, 0)$$

$$G(x_1, \ldots, x_n, y) \Leftarrow \textit{if-then-else}(g(x_1, \ldots, x_n, t), t, G(x_1, \ldots, x_n, S(t)))$$

□

Esercizio 6.21 Fornire un programma *SLF* per il calcolo di 4!.

Il formalismo di McCarthy e il linguaggio *SLF* è l'anello di raccordo tra il formalismo delle funzioni ricorsive e i reali linguaggi di programmazione funzionali che ne sono stati derivati. In particolare, il linguaggio di programmazione *LISP* può essere ottenuto a partire dal formalismo di McCarthy assumendo come funzioni base le operazioni di manipolazione di liste adottando una diversa interpretazione dell'insieme dei simboli delle funzioni base, e cioè assumendo le funzioni base di manipolazione di liste.

Capitolo 7

Teoria generale della calcolabilità

7.1 Tesi di Church-Turing

Quanto si è visto nel capitolo precedente ha due aspetti interessanti: il primo è che differenti formulazioni del concetto di calcolabilità, corrispondenti a diverse nozioni di algoritmo e di dispositivo di calcolo, portano a definire tutte la stessa classe di funzioni; il secondo è che le tecniche di simulazione di una macchina da parte di un'altra permettono di introdurre concetti fondamentali per una teoria della programmazione, come quelli di configurazione, di sequenza di calcolo e di aritmetizzazione. Vedremo, nelle prossime sezioni, come il concetto di aritmetizzazione ci permette di formulare una teoria astratta della calcolabilità, uniforme rispetto alla classe di macchine che esegue il calcolo. In questa sezione faremo invece alcune considerazioni sulla equivalenza delle varie nozioni di calcolabilità.

Durante gli anni '30 e all'inizio degli anni '40 hanno visto la luce tutte le più importanti caratterizzazioni delle funzioni calcolabili: quella di Turing, basata sulle macchine omonime, quella di Kleene, basata sulle equazioni funzionali, quella di Church, basata sul λ-calcolo, quella di Markov, basata sugli algoritmi normali e quella di Post, basata sui sistemi combinatori (si vedano le note storiche al termine del capitolo). L'estrema semplicità delle assunzioni, operazioni base e meccanismi di calcolo dotati di struttura estremamente elementare, hanno portato i vari autori a formulare l'ipotesi che la loro nozione di calcolabilità fosse la più generale possibile e che non potessero esistere procedimenti algoritmici di calcolo non esprimibili nel loro formalismo. In particolare, i nomi di Church e di Turing sono stati legati a questo punto di vista: infatti, con il nome di *Tesi di Church-Turing* ci si riferisce alla congettura che ogni funzione calcolabile rispetto ad una qualunque intuitiva nozione di algoritmo sia calcolabile secondo Turing. Dato che, come abbiamo visto, la classe delle funzioni ricorsive coincide con la classe delle funzioni calcolabili secondo Turing, ne consegue che, in generale e nel seguito di questo testo, il termine

"funzioni ricorsive" viene utilizzato come sinonimo di funzioni calcolabili.

La constatazione dell'equivalenza dei vari meccanismi algoritmici non ha fatto altro che corroborare l'idea che le funzioni ricorsive siano la più generale classe di funzioni per le quali si possa comunque dare un metodo di calcolo intuitivamente algoritmico. La tesi di Church-Turing è diventata così il principio fondante della ricorsività ed è ora indiscussamente accettata, anche se la sua indimostrabilità è peraltro evidente.

Una volta ammessa la validità della tesi di Church-Turing è quindi possibile utilizzare descrizioni informali (purché rigorose) di algoritmi al fine di dimostrare la calcolabilità di certe funzioni senza dovere esibire una macchina di Turing (o un programma per macchina a registri) che le calcola. L'uso della tesi di Church-Turing consente di esprimere rapidamente e suggestivamente risultati che un approccio più formale permetterebbe di raggiungere più faticosamente.

Ad esempio, usando la tesi di Church-Turing possiamo agevolmente definire la funzione:

$$f_\pi(x,z) = \begin{cases} y & \text{se il programma } \pi \text{ con input } x \text{ restituisce il} \\ & \text{valore } y \text{ con un calcolo lungo al più } z \text{ passi;} \\ 0 & \text{altrimenti} \end{cases}$$

e dire che essa è ricorsiva anche senza fornire esplicitamente l'algoritmo che la calcola (il quale richiederebbe, ad esempio, una definizione formale del modello di calcolo adottato, nonché del relativo concetto di numero di passi).

7.2 Enumerazione di funzioni ricorsive

In questa sezione mostreremo come per le macchine a registri elementari e per le macchine di Turing si possa stabilire una corrispondenza biunivoca con l'insieme dei numeri naturali: tale corrispondenza viene chiamata *aritmetizzazione*.[1] Vedremo poi come ciò consenta di ottenere una enumerazione delle funzioni ricorsive e di formulare una teoria generale della calcolabilità, indipendente dal particolare modello di calcolo assunto.

7.2.1 Enumerazione delle macchine a registri elementari

Nella Sezione 6.6 abbiamo visto come ogni funzione calcolata mediante un programma elementare per macchine a registri sia una funzione ricorsiva. Ciò è stato dimostrato introducendo il concetto di codifica di una computazione, e realizzando tale codifica mediante l'associazione di numeri naturali agli stati della macchina e alle sequenze finite di stati. In questa sezione intendiamo

[1] Un termine usato alternativamente per tale corrispondenza è quello di *gödelizzazione*, in quanto tale approccio fu introdotto da K. Gödel per realizzare una corrispondenza biunivoca tra formule logiche e numeri naturali.

mostrare che anche i programmi per macchine a registri elementari possono essere codificati con numeri naturali.

Ricordiamo anzitutto che ogni programma elementare è costituito da un numero finito di istruzioni del tipo:

$$\begin{array}{ll} Ri = Ri + 1 & i \geq 1,\ n \geq 0 \\ Ri = Ri \dot{-} 1 & \\ \text{IF } Ri = 0 \text{ GOTO } n & \end{array}$$

L'informazione di cui abbiamo bisogno per caratterizzare ogni istruzione è, dunque, il suo posto nella sequenza, il suo tipo e i naturali i ed n che essa contiene. Un modo per codificare queste informazioni è il seguente:

$$\begin{array}{rcl} Ri = Ri + 1 & \longrightarrow & 3 \cdot (i \dot{-} 1) + 1 \\ Ri = Ri \dot{-} 1 & \longrightarrow & 3 \cdot (i \dot{-} 1) + 2 \\ \text{IF } Ri = 0 \text{ GOTO } n & \longrightarrow & 3 \cdot J(i \dot{-} 1, n) \end{array}$$

dove J è la biiezione $\mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}^+$:

$$J(x, y) = \frac{(x + y + 1)(x + y)}{2} + x + 1. \tag{7.1}$$

simile alla funzione coppia di Cantor già definita nell'Esempio 1.11.

Il codice dell'istruzione m-esima viene poi usato come esponente per il numero primo p_m ($p_1 = 2$), e quindi se il programma consta di m istruzioni il suo codice sarà, coerentemente con la notazione introdotta nella Sezione 6.6, $\langle y_1, \ldots, y_m \rangle$ dove $y_1, \ldots, y_m$ sono i codici delle rispettive istruzioni.

Esempio 7.1 Il seguente programma di somma viene così codificato:

$$\begin{array}{ll} \text{IF } R2 = 0 \text{ GOTO } 0 & 3 \cdot J(1, 0) = 9 \\ R2 = R2 \dot{-} 1 & 3 \cdot 1 + 2 = 5 \\ R1 = R1 + 1 & 3 \cdot 0 + 1 = 1 \\ \text{IF } R3 = 0 \text{ GOTO } 1 & 3 \cdot J(2, 1) = 27 \end{array}$$

e ad esso corrisponde il codice $2^9 \cdot 3^5 \cdot 5^1 \cdot 7^{27}$.

Poiché sia la codificazione delle istruzioni che la codificazione del programma sono biunivoche (in virtù dell'unicità della decomposizione in primi e del quoziente e resto della divisione per tre), come ad ogni programma corrisponde un intero, maggiore o uguale a 2, così ad ogni intero:

$$x = 2^{y_1} \ldots {p_m}^{y_m} \qquad \text{dove} \qquad y_i \begin{cases} \geq 0 & \text{se } 1 \leq i \leq m \\ > 0 & \text{se } i = m \end{cases}$$

corrisponde un programma costituito da m istruzioni in cui la j-esima istruzione è ottenuta decodificando y_j, vale a dire dividendo y_j per 3 ed individuando resto e quoziente.

Esempio 7.2 Al numero 600 corrisponde la decomposizione $2^3 \cdot 3 \cdot 5^2$ e quindi il programma:

$$\begin{array}{l} \text{IF } R1 = 0 \text{ GOTO } 0 \\ R1 = R1 + 1 \\ R1 = R1 \dot{-} 1. \end{array}$$

Tale programma calcola la funzione:

$$f(x) = \begin{cases} \text{se } x = 0 \text{ allora } 0 \\ \text{indefinita altrimenti} \end{cases}$$

infatti il programma si arresta regolarmente se e solo se il contenuto del registro $R1$ è 0.

Si noti anche che, mentre la codificazione di un programma in un naturale produce solo naturali x tali che, se $2^{y_1} \dots {p_m}^{y_m}$ è la decomposizione in fattori primi di x, allora $y_1, \dots, y_m > 0$, in generale decodificando un numero naturale per ricavare il programma ci imbatteremo in situazioni in cui all'istruzione j-esima corrisponde $y_j = 0$. Come si può verificare, allo 0 non è associata alcuna istruzione. Introdurremo perciò una istruzione NOPE (con codice 0) di non-operazione che ha l'unico effetto di passare il controllo all'istruzione successiva.

Così al numero naturale $2^9 \cdot 5^5 \cdot 7^1 \cdot 11^{27}$ corrisponde una sequenza uguale a quella del programma di somma visto precedentemente, ma con una NOPE inserita fra la prima e la seconda istruzione.

Avendo stabilito una corrispondenza biunivoca tra naturali e macchine a registri elementari, possiamo pensare che esista una macchina a registri elementare che riceve in input il numero naturale che codifica un'altra macchina e ne simula il comportamento.

In altre parole, analogamente a quanto fatto nella Sezione 5.8 per le macchine di Turing, possiamo introdurre il concetto di macchina a registri universale.

Esercizio 7.1 Definire una macchina a registri (non elementare) U che, dati in input due naturali x e y, determina l'output che la macchina a registri elementare M_x con codifica x ottiene su input y.
[Suggerimento: La macchina U deve effettuare la decomposizione di x in fattori primi, decodificare gli esponenti della decomposizione per individuare a che istruzione corrispondano, ed eseguire via via tali istruzioni. Si suggerisce di estendere la definizione di macchina a registri includendo istruzioni complesse quali quella che calcola l'esponente del numero primo i-esimo nella decomposizione del naturale j, o quelle che consentono di determinare x e y dato $J(x, y)$.]

7.2.2 Enumerazione delle macchine di Turing

Che le macchine di Turing siano un'infinità numerabile è già stato osservato, ed è un'ovvia conseguenza del fatto che esse possono essere descritte mediante

stringhe di lunghezza finita su di un opportuno alfabeto finito. Nel seguito vedremo un particolare modo con cui si può realizzare una corrispondenza biunivoca tra le macchine di Turing ed i numeri naturali.

Supponiamo inizialmente che siano dati un alfabeto Γ ed un insieme di stati Q, e mostriamo come sia possibile enumerare l'insieme delle macchine di Turing con alfabeto di nastro Γ ed insieme degli stati Q.

Si ricorda, dalla Sezione 5.1, che la funzione di transizione di una macchina di Turing deterministica $\mathcal{M} = \langle \Gamma, \text{b}, Q, q_0, F, \delta \rangle$ è definita come una funzione parziale $\delta : (Q - F) \times (\Gamma \cup \{\text{b}\}) \mapsto Q \times (\Gamma \cup \{\text{b}\}) \times \{d, s, i\}$.

Il metodo che utilizzeremo per codificare una macchina di Turing consiste nell'elencare in modo lessicografico le quintuple che ne costituiscono la funzione di transizione: a tal fine, ogni coppia (q, a) per la quale la funzione di transizione non è definita sarà codificata con la quintupla $(q, a, \perp, \perp, \perp)$, dove il simbolo $\perp$ non appartiene a Γ.

Per poter effettuare un'enumerazione lessicografica è necessario introdurre un ordinamento arbitrario sugli insiemi $Q \cup \{\perp\}$, $\Gamma \cup \{\text{b}, \perp\}$, $\{s, d, i, \perp\}$: tale ordinamento induce un ordinamento tra le quintuple, il quale, a sua volta, induce un ordinamento lessicografico tra sequenze di quintuple, e quindi tra codifiche di macchine di Turing.

Si noti che l'ordinamento sopra definito si applica ad insiemi di macchine di Turing definite su una prefissata coppia di insiemi Γ, Q. Al fine di ottenere l'enumerazione di tutte le macchine di Turing, definite quindi su qualunque alfabeto di nastro e qualunque insieme degli stati, è necessario estendere opportunamente il procedimento.

Esercizio 7.2 Completare al caso generale la definizione dell'enumerazione delle macchine di Turing sopra introdotta.
[Suggerimento: Assumere che esista un alfabeto Σ tale che, per ogni Q, Γ i nomi degli stati in $Q \cup \{\perp\}$ e dei simboli in $\Gamma \cup \{\text{b}, \perp\}$ siano rappresentati mediante stringhe finite in Σ^*. Dimostrare quindi che l'insieme di tutte le coppie (Γ, Q) è numerabile. Infine, utilizzare il procedimento introdotto nel Teorema 1.4 per completare la dimostrazione]

Come si può facilmente comprendere, il metodo di enumerazione visto non è l'unico possibile. Ad esempio, utilizzando la descrizione linearizzata delle macchine di Turing, è possibile derivare una diversa enumerazione delle macchine stesse.

Esercizio 7.3 Definire una corrispondenza biunivoca tra i naturali e le macchine di Turing basata sulla descrizione linearizzata.

7.2.3 Enumerazione delle funzioni ricorsive

I metodi di enumerazione delle macchine introdotti nelle sottosezioni precedenti, oltre a definire delle biiezioni fra i numeri naturali e le macchine stesse,

determinano anche una corrispondenza tra numeri naturali e funzioni ricorsive.

Sia ad esempio $\mathcal{E} : \mathbb{N} \longrightarrow \{\mathcal{M}\}$ una biiezione tra $\mathbb{N}$ e le descrizioni delle macchine di Turing. Con la notazione $\mathcal{M}_x$ indichiamo la macchina $\mathcal{E}(x)$, cioè la $(x+1)$-esima nell'enumerazione. Data una qualunque codifica $\mathcal{C}$ dei naturali su un opportuno alfabeto, consideriamo ora la funzione ricorsiva parziale così definita:

$$\varphi_x(y) = \begin{cases} z \in \mathbb{N} & \text{se inizializzata nella configurazione iniziale} \\ & q_0\mathcal{C}(y) \text{ la macchina } \mathcal{M}_x \text{ si arresta nella} \\ & \text{configurazione finale } \mathcal{C}(y)\text{b}q_F\mathcal{C}(z) \\ \text{indefinita} & \text{altrimenti.} \end{cases}$$

In tal modo, a fianco della enumerazione delle macchine, otteniamo una enumerazione delle funzioni ricorsive ad un argomento e, in particolare, la funzione precedentemente definita è la $x + 1$-esima di tale enumerazione.

Si noti che mentre la funzione $\mathcal{E}$ è una biiezione, la corrispondenza tra naturali e funzioni ricorsive ad un argomento non lo è, a causa del fatto che ogni funzione è calcolata da più macchine di Turing. L'estensione al caso generale di funzioni n-arie di quanto detto con riferimento alle funzioni ricorsive ad un argomento, può essere semplicemente ottenuta ricordando che una funzione n-aria può essere posta in corrispondenza con una funzione unaria, il cui unico argomento codifica gli n argomenti della funzione data.

Le stesse considerazioni valgono per qualunque altra enumerazione di qualunque altro tipo di macchine, ed in particolare per l'enumerazione delle macchine a registri elementari introdotta nella Sottosezione 7.2.1.

Il valore x associato alla funzione φ_x viene detto indice *indice della funzione* (calcolata dalla macchina $\mathcal{M}_x$). Dato che x è il numero naturale che codifica la macchina $\mathcal{M}_x$, viene anche chiamato, con un piccolo abuso di terminologia, il programma che calcola la funzione φ_x.

7.3 Proprietà di enumerazioni di funzioni ricorsive

L'introduzione del concetto di enumerazione delle funzioni ricorsive ci consente di formulare i risultati fondamentali della teoria della ricorsività in modo "machine independent". In altri termini, tutta l'informazione relativa ad una particolare classe di macchine o di programmi (e ad un particolare modo di codificare input e output) rimane racchiusa negli indici delle funzioni e dà luogo a una enumerazione che potrà anche essere diversa da quelle introdotte precedentemente, ma che ne condividerà le proprietà generali. In tal modo, una dimostrazione fornita assumendo una enumerazione corrispondente ad un generico modello di calcolo, automaticamente diviene una dimostrazione valida per qualunque enumerazione associata ad una altro modello di calcolo (e ad una qualunque altra codificazione dell'input e dell'output).

Nel seguito, illustriamo alcune proprietà fondamentali che valgono per ogni enumerazione delle funzioni ricorsive.

La prima di tali proprietà consiste nella esistenza di una funzione universale corrispondente alla funzione calcolata dalla macchina universale. Il teorema seguente consiste nella affermazione della esistenza della funzione universale nella sua forma più generale.

Teorema 7.1 (Esistenza della funzione universale) *Sia data una qualunque enumerazione delle funzioni ricorsive. Per ogni $k \in \mathbb{N}$ esiste z tale che, per ogni $x_1, \ldots, x_k$:*

$$\varphi_z(x_1, \ldots, x_k, y) = \begin{cases} \varphi_y(x_1, \ldots, x_k) & \text{se essa è definita} \\ \textit{indefinita}, & \text{altrimenti.} \end{cases}$$

Dimostrazione. La dimostrazione è soltanto accennata. Anzitutto, non è difficile verificare che una funzione universale con le caratteristiche suddette esiste nel caso delle macchine a registri: in tal caso, infatti, è sufficiente realizzare la codifica dei k argomenti in un unico numero naturale e fare ricorso alla macchina universale vista nell'Esercizio 7.1. Al fine di estendere tale proprietà ad enumerazioni associate ad altri modelli di calcolo, osserviamo che, dati due modelli di calcolo $\mathcal{MC}_1, \mathcal{MC}_2$ dotati di potere computazionale equivalente alle macchine di Turing, esiste una funzione calcolabile che associa al codice di una macchina in $\mathcal{MC}_1$ il codice di una macchina in $\mathcal{MC}_2$. Quindi, l'esistenza di una funzione universale in $\mathcal{MC}_2$ implica l'esistenza di una funzione universale in $\mathcal{MC}_1$. □

Esercizio 7.4 Dimostrare l'ultima affermazione contenuta nella dimostrazione del teorema precedente, vale a dire che, nelle condizioni ivi indicate, l'esistenza di una funzione universale in $\mathcal{MC}_2$ implica l'esistenza di una funzione universale in $\mathcal{MC}_1$.

Il secondo risultato, detto Teorema s-m-n, consente di effettuare in modo algoritmico la trasformazione da un programma che ammette come input i dati $x_1, \ldots, x_n$ ed $y_1, \ldots, y_m$, ad un insieme di programmi che inglobano in sé i dati $y_1, \ldots, y_m$ come parametri.

Teorema 7.2 (s-m-n) *Sia data una qualunque enumerazione delle funzioni ricorsive. Per ogni $m, n \geq 1$ esiste una funzione ricorsiva s tale che per ogni $x, y_1, \ldots, y_m, z_1, \ldots, z_n$:*

$$\varphi_x(y_1, \ldots, y_m, z_1, \ldots, z_n) = \varphi_{s(x,y_1,\ldots,y_m)}(z_1 \ldots z_n) \quad .$$

Dimostrazione. Assumiamo per semplicità $m = n = 1$ e dimostriamo che, nel caso della enumerazione associata alle macchine a registri, esiste s tale che $\varphi_x(y, z) = \varphi_{s(x,y)}(z)$. Dati x e y, un modo per calcolare il valore $s(x, y)$ è il seguente: generiamo la macchina a registri di indice x e la trasformiamo in

una nuova macchina la quale, dopo avere, con la sua prima istruzione, inserito il valore y nel primo registro, si comporta esattamente come la precedente. Infine, determiniamo il codice di tale nuova macchina $x' = s(x, y)$.

Si comprende facilmente che l'estensione della dimostrazione a valori di m ed n qualunque è immediata. Per quanto riguarda l'estensione del risultato a tutte le enumerazioni, ricordiamo quanto detto nella dimostrazione del Teorema 7.1. □

Esercizio 7.5 Completare la dimostrazione precedente, mostrando in particolare come l'esistenza della corrispondenza tra modelli di calcolo equivalenti alle macchine di Turing consenta di estendere il teorema s-m-n a tutte le enumerazioni di funzioni ricorsive.

Esempi di applicazione del teorema s-m-n saranno forniti nelle sezioni successive.

Un ultimo risultato interessante basato sulla enumerazione delle funzioni ricorsive è il teorema seguente.

Teorema 7.3 (Forma Normale di Kleene) *Data una qualunque enumerazione delle funzioni ricorsive, esistono una funzione ricorsiva U e un insieme di funzioni ricorsive T_k tali che, per ogni $k \in \mathbb{N}$, tutte le funzioni ricorsive di k argomenti sono esprimibili nel seguente modo:*

$$\varphi_i(x_1, \ldots, x_k) = U(\mu t[T_k(i, x_1, \ldots, x_k, t) = 0]).$$

Dimostrazione. Dimostriamo inizialmente il risultato nel caso in cui l'enumerazione considerata sia quella derivante dall'enumerazione delle macchine a registri elementari (vedi Sezione 7.2.1). Nel Teorema 6.6 abbiamo dimostrato che dato ogni programma π siamo in grado di fornire una funzione ricorsiva T_π tale che se f è la funzione calcolata mediante π, allora:

$$f(x_1, \ldots, x_k) = U(\mu t[T_\pi(x_1, \ldots, x_k, t) = 0]).$$

Sia ora π_i il programma per macchine a registri elementari che calcola la funzione φ_i. In base alla definizione della funzione T_π non è difficile comprendere che tale funzione può essere costruita a partire dal codice del programma π, e che tale costruzione può essere effettuata in modo algoritmico. In altre parole, dato k, esiste una funzione ricorsiva T_k che, assumendo in input il codice i di un programma π_i, determina tale programma e, dati i valori $x_1, \ldots, x_k, t$, effettua i controlli previsti nel Lemma 6.5 e calcola il valore $T_{\pi_i}(x_1, \ldots, x_k, t)$. La generalizzazione del risultato ad una qualunque enumerazione di funzioni ricorsive è lasciata al lettore. □

Esercizio 7.6 Completare la dimostrazione precedente, mostrando che il risultato vale per qualunque enumerazione di funzioni ricorsive.

Il Teorema della Forma Normale di Kleene mostra che il calcolo di una qualunque funzione ricorsiva può essere ricondotto al calcolo di due funzioni elementari: la prima (predicato di Kleene) consistente nel verificare che una computazione soddisfi dei requisiti di ammissibilità e la seconda che si limita a decodificare il codice della computazione per estrarre il risultato del calcolo.

7.4 Funzioni non calcolabili

Nel Capitolo 5 abbiamo introdotto le nozioni di funzione calcolabile, di funzione non calcolabile, di insieme decidibile, di insieme semidecidibile ecc. Avendo ora introdotto il concetto di enumerazione di funzioni ricorsive, possiamo riprendere in modo indipendente dal modello di calcolo quanto visto con riferimento specifico alle macchine di Turing, e fornire ulteriori esempi di funzioni non calcolabili.

Mostriamo innanzi tutto come si possa riformulare, in modo "macchine independent," la dimostrazione della non decidibilità del problema della terminazione.

Teorema 7.4 *Sia data una qualunque enumerazione delle funzioni ricorsive. Non esiste allora alcuna funzione ricorsiva g tale che per ogni x e y:*

$$g(x,y) = \begin{cases} 1 & \text{se } \varphi_x(y) \text{ è definita} \\ 0 & \text{altrimenti} \end{cases}$$

Dimostrazione. Non è difficile verificare che, se la funzione g fosse ricorsiva, lo sarebbe anche la funzione:

$$f(x) = \begin{cases} 1 & \text{se } g(x,x) = 0 \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Si noti che la funzione f è costruita appositamente per fornire un controesempio, come già la macchina di Turing $\mathcal{H}''$ nella dimostrazione del Teorema 5.8: infatti se ora assumiamo che e sia un indice per la funzione f e applichiamo la f al proprio indice, abbiamo:

$$f(e) = \varphi_e(e) = 1 \qquad \text{se e solo se} \qquad g(e,e) = 0$$

ma contemporaneamente, per definizione della g :

$$g(e,e) = 0 \qquad \text{se e solo se} \qquad \varphi_e(e) \text{ non è definita.}$$

Ad una analoga contraddizione si perviene nel caso in cui

$$f(e) = \varphi_e(e) = 0$$

In conclusione, si può desumere che la funzione g non può essere ricorsiva. □

Si noti che, utilizzando la stessa tecnica di dimostrazione, otteniamo il seguente risultato.

Teorema 7.5 *Non esiste una funzione ricorsiva g' tale che per ogni x :*

$$g'(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ è definita} \\ 0 & \text{altrimenti} \end{cases}$$

Una conseguenza immediata di tali risultati è rappresentata dal seguente corollario.

Corollario 7.6 *Data una qualunque enumerazione delle funzioni ricorsive, gli insiemi*

$$K = \{x \mid \varphi_x(x) \text{ è definita}\}$$

e

$$K' = \{\langle x, y\rangle \mid \varphi_x(y) \text{ è definita}\}$$

non sono decidibili.

Quanto visto implica che il problema di decidere se un generico programma si arresta su un input generico non è più "difficile" del caso particolare in cui ci si chiede se un generico programma termina quando gli viene fornito in input il proprio codice. Si può anzi osservare, come enunciato nel teorema seguente, che anche nel caso in cui l'input fornito ad un programma è prefissato (ad esempio viene fornito in input il valore costante 0), il problema della terminazione rimane indecidibile.

Prima di fornire l'enunciato del teorema seguente, osserviamo che la dimostrazione del teorema stesso utilizzerà la tecnica già vista nella Sezione 4.8.1 e cioè si baserà sulla riduzione del problema della terminazione al problema in esame. Inoltre in essa si farà uso del teorema *s-m-n* (Teorema 7.2) precedentemente enunciato.

Teorema 7.7 *Data una qualunque enumerazione delle funzioni ricorsive, non esiste una funzione ricorsiva g tale che per ogni x :*

$$g(x) = \begin{cases} 1 & \text{se } \varphi_x(0) \text{ è definita} \\ 0 & \text{altrimenti} \end{cases}$$

Dimostrazione. Supponiamo per assurdo che la funzione g fosse ricorsiva: in tal caso potremmo definire una funzione ricorsiva g' capace di decidere per ogni x se $\varphi_x(x)$ è definita o no. Sia:

$$f(x, y) = \begin{cases} 0 & \text{se } \varphi_x(x) \text{ è definita} \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Chiaramente, la funzione f è ricorsiva e, in base al teorema *s-m-n* (Teorema 7.2) esiste s tale che: $\varphi_{s(x)}(y) = f(x, y)$. In base alla definizione della funzione f, si può osservare che le funzioni $\varphi_{s(x)}$ costituiscono, al variare di $x \in \mathbb{N}$ una sequenza infinita di funzioni ognuna delle quali o è identicamente nulla, o è sempre indefinita. In particolare, se $\varphi_x(x)$ è definita, allora $\varphi_{s(x)}$ è identicamente nulla, mentre se $\varphi_x(x)$ è indefinita, allora è sempre indefinita la funzione $\varphi_{s(x)}$.

Se ora consideriamo $g' = g \cdot s$ abbiamo che:

$$g'(x) = g(s(x)) = \begin{cases} 1 & \text{se } \varphi_{s(x)}(0) \text{ è definita} \\ 0 & \text{altrimenti} \end{cases}$$

e quindi:

$$g'(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ è definita} \\ 0 & \text{altrimenti} \end{cases}$$

Poiché tale g' non può essere ricorsiva, se ne deve dedurre che non è ricorsiva neanche la funzione g. □

Lasciamo per esercizio la dimostrazione del seguente teorema, che specializza il teorema precedente al caso delle macchine di Turing.

Teorema 7.8 *Data una qualunque enumerazione delle macchine di Turing, la funzione*

$$b(x) = \begin{cases} 1 & \textit{se } \mathcal{M}_x \textit{ termina su nastro } \text{ƀ} \\ 0 & \textit{altrimenti} \end{cases} \tag{7.2}$$

non è calcolabile.

Esercizio 7.7 Dimostrare il Teorema 7.8.

Applicando la tecnica utilizzata nella dimostrazione del Teorema 7.7, è possibile derivare i seguenti risultati.

Teorema 7.9 *Data una qualunque enumerazione delle funzioni ricorsive, non esiste una funzione ricorsiva g tale che, per ogni x, si ha che:*

$$g(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è costante} \\ 0 & \text{altrimenti} \end{cases}$$

Dimostrazione. Come nella dimostrazione del Teorema 7.7, consideriamo le funzioni

$$f(x, y) = \begin{cases} 0 & \text{se } \varphi_x(x) \text{ è definita} \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

e $\varphi_{s(x)}(y) = f(x,y)$. Anche in questo caso, la funzione

$$g'(x) = g(s(x)) = \begin{cases} 1 & \text{se } \varphi_{s(x)} \text{ è costante} \\ 0 & \text{altrimenti} \end{cases}$$

risolverebbe il problema della fermata: infatti è chiaro che $\varphi_{s(x)}$ è costante (ed uguale a 0) se e solo se $\varphi_x(x)$ è definita. □

Teorema 7.10 *Data una qualunque enumerazione delle funzioni ricorsive, non esiste una funzione ricorsiva g tale che, per ogni x, y, z:*

$$g(x,y,z) = \begin{cases} 1 & \text{se } \varphi_x(y) = z \\ 0 & \text{altrimenti} \end{cases}$$

Dimostrazione. Si avrebbe, in caso contrario:

$$g'(x) = g(s(x),0,0) = \begin{cases} 1 & \text{se } \varphi_{s(x)}(0) = 0 \\ 0 & \text{altrimenti} \end{cases}$$

e g' risolverebbe il problema della fermata. □

Teorema 7.11 *Data una qualunque enumerazione delle funzioni ricorsive, non esiste una funzione ricorsiva g tale che, per ogni x e y :*

$$g(x,y) = \begin{cases} 1 & \text{se } \varphi_x = \varphi_y \\ 0 & \text{altrimenti .} \end{cases}$$

Dimostrazione. Sia e un indice per la funzione $\lambda x[0]$. Avremmo ancora che:

$$g'(x) = g(s(x),e) = \begin{cases} 1 & \text{se } \varphi_{s(x)} = \varphi_e \\ 0 & \text{altrimenti} \end{cases}$$

risolverebbe il problema della terminazione. □

Esercizio 7.8 Mostrare che non esiste una funzione ricorsiva g tale che, per ogni x,

$$g(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è totale} \\ 0 & \text{altrimenti} \end{cases}$$

Esercizio 7.9 Mostrare che non esiste una funzione ricorsiva g tale che, per ogni x,

$$g(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è crescente} \\ 0 & \text{altrimenti} \end{cases}$$

In realtà, come vedremo nella Sezione 7.6, è possibile dimostrare che non solo sono indecidibili le proprietà suddette, ma ogni altra proprietà "non banale" relativa alla funzione calcolata mediante un programma x.

Il metodo introdotto sopra, in cui ci si riconduce alla indecidibilità del problema della terminazione, non è in effetti l'unico utilizzabile per dimostrare risultati di indecidibilità. Un ulteriore metodo consiste nell'utilizzare direttamente una diagonalizzazione, così come si è fatto proprio nella dimostrazione dell'indecidibilità del problema della terminazione (Teorema 7.4).

Come esempio di applicazione di questo ulteriore metodo, forniamo una nuova dimostrazione dell'indecidibilità del problema di verificare se la funzione calcolata da un programma è totale.

Teorema 7.12 *Data una qualunque enumerazione delle funzioni ricorsive, non esiste una funzione ricorsiva g tale che, per ogni x,*

$$g(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è totale} \\ 0 & \text{altrimenti} \end{cases}$$

Dimostrazione. Supponiamo per assurdo che la funzione g sia calcolabile. In tal caso sarebbe calcolabile anche la funzione

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{se } g(x) = 1, \text{ cioè se } \varphi_x \text{ è totale} \\ 0 & \text{altrimenti} \end{cases}$$

Chiaramente, la funzione f sarebbe totale. Se e fosse un suo indice si avrebbe allora la contraddizione seguente:

$$\varphi_e(e) = f(e) = \varphi_e(e) + 1$$

In conclusione, la funzione g non può essere calcolabile. □

7.5 Indecidibilità in matematica ed informatica

Oltre ai problemi indecidibili già illustrati nelle sezioni precedenti esistono numerosi altri esempi di problemi indecidibili che si incontrano in diversi settori della matematica e dell'informatica.

Nel campo della teoria dei linguaggi formali, ad esempio, abbiamo già visto (vedi Sezione 4.8.1) che il problema della ambiguità di una grammatica context free è indecidibile. Altri esempi di problemi indecidibili nella teoria dei linguaggi formali sono i seguenti.

- Date due grammatiche context free $\mathcal{G}_1, \mathcal{G}_2$, $L(\mathcal{G}_1) = L(\mathcal{G}_2)$?
- Date due grammatiche context free $\mathcal{G}_1, \mathcal{G}_2$, $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) = \emptyset$?

- Data una grammatica contestuale $\mathcal{G}$, il linguaggio $L(\mathcal{G})$ è vuoto?

Nell'ambito della teoria della programmazione, i risultati di indecidibilità già incontrati nelle sezioni precedenti hanno implicazioni molto significative per la teoria stessa. Infatti, facendo riferimento ad uno specifico linguaggio di programmazione, anziché ad un generico modello di calcolo, tali risultati possono essere interpretati in termini di indecidibilità dei seguenti problemi.

- Un programma contenente la dichiarazione di una particolare procedura chiamerà, nel corso della sua esecuzione, la procedura stessa?
- Una variabile definita all'interno di un programma assumerà un particolare valore, durante l'esecuzione del programma stesso? In particolare, un predicato utilizzato come test di uscita da un ciclo assumerà il valore VERO?
- Un programma, in corrispondenza di un particolare input, fornisce un determinato output?
- Due programmi sono equivalenti, ovvero calcolano la medesima funzione?

Oltre ai problemi precedenti, che riguardano l'impossibilità di decidere in merito al comportamento di un programma, sono anche indecidibili, in conseguenza del teorema di Rice (vedi Teorema 7.15), i problemi seguenti, ed in generale qualunque proprietà non banale[2] relativa alla funzione calcolata da un programma.

- Un programma calcola una funzione costante, ovvero il suo output è indipendente dall'input?
- Un programma calcola una funzione totale, ovvero termina per ogni input?
- Un programma calcola una funzione crescente?

Nel campo della matematica, e più precisamente della logica, hanno particolare importanza anche da un punto di vista storico, in quanto all'origine degli studi sulla calcolabilità, i seguenti problemi.

- Data una formula del calcolo dei predicati, tale formula è un teorema?
- Data una formula dell'aritmetica, tale formula è un teorema della teoria?

Un altro celebre problema che è stato mostrato indecidibile è il problema delle equazioni diofantee, che può essere così formulato.

[2]Una proprieà è non banale se è valida per un insieme non vuoto di istanze, ma non per l'intero universo delle istanze stesse.

- Dato un sistema di equazioni lineari a coefficienti ed esponenti interi, come ad esempio l'equazione $x^2 + y^2 = z^2$, il sistema ammette soluzioni intere?

Un ulteriore problema indecidibile, di natura combinatoria, è il seguente problema di pavimentazione (o *tiling*).

- Dato un insieme di mattonelle quadrate con il bordo colorato, ciascuna suddivisa in quattro triangoli come in Figura 7.1, e disponibili in numeri arbitrario, è possibile pavimentare un pavimento di qualsiasi forma?[3]

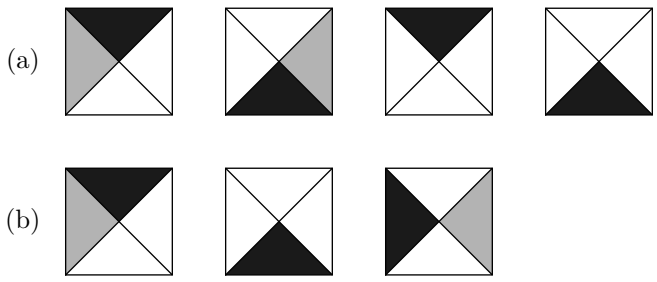


FIGURA 7.1 Due insiemi di base di mattonelle. (a) Caso in cui è possibile pavimentare. (b) Caso in cui non è possibile.

Infine, osserviamo come esistano anche circostanze in cui un una funzione è calcolabile, ma non siamo in grado di conoscere il programma che la calcola. Un esempio di funzione di tale tipo è il seguente.

- Consideriamo la funzione $F(x)$ definita come:

$$F(x) = \begin{cases} 1 \text{ se il problema del continuo ha soluzione positiva} \\ 0 \text{ altrimenti} \end{cases}$$

  dove si ricorda che il problema del continuo è stato definito nella Sezione 1.1.8.

La funzione $F(x)$ coincide o con la funzione $0(x) = 0$ o con la $1(x) = 1$; in entrambi i casi si tratta di funzioni costanti e quindi chiaramente calcolabili,

[3]Con "pavimentazione" si intende una disposizione delle mattonelle in cui due mattonelle vengono accostate, verticalmente o orizzontalmente, solo se i loro bordi hanno lo stesso colore.

ma noi non saremo in grado di determinare quale sia l'algoritmo corretto per calcolare la F fino a che non si saprà se il problema del continuo ha soluzione positiva o negativa.

Analogamente la funzione

$$g(x) = \begin{cases} 1 \text{ se nell'espansione di } \pi \text{ esistono almeno} \\ \quad x \text{ 5 consecutivi} \\ \\ 0 \text{ altrimenti} \end{cases}$$

è sicuramente calcolabile. Infatti si presentano due casi:

1. esiste un k tale che per $x > k$ non esistono x 5 consecutivi;
2. per ogni x esistono almeno x cinque consecutivi.

In ogni caso, la funzione è calcolabile; tuttavia non sappiamo quale sia l'algoritmo corretto per calcolarla.

Infine, ricordiamo che esistono tuttora funzioni di cui non è noto se siano calcolabili o non calcolabili. Consideriamo ad esempio la seguente funzione.

$$g'(x) = \begin{cases} 1 \text{ se nell'espansione di } \pi \text{ esistono esatta-} \\ \quad \text{mente } x \text{ 5 consecutivi} \\ \\ 0 \text{ altrimenti} \end{cases}$$

In questo caso la funzione g' può essere calcolabile ma nulla esclude che si possa dimostrare che essa non è calcolabile.

7.6 Teoremi di Kleene e di Rice

In questa sezione presentiamo due risultati di notevole importanza per la conoscenza delle proprietà delle funzioni calcolabili.

Il primo è noto come *teorema di ricursione* (o teorema di *Kleene*).

Teorema 7.13 (Ricursione) *Sia data una enumerazione delle funzioni ricorsive. Se t è una funzione calcolabile totale, allora esiste $e \in \mathbb{N}$ tale che*

$$\varphi_e = \varphi_{t(e)}$$

Prima di procedere con la dimostrazione del teorema osserviamo che il teorema stesso è anche chiamato teorema del *punto fisso*. In generale, dati un insieme S ed una trasformazione $\tau : S \mapsto S$, si definisce *punto fisso* di τ un valore $s \in S$ tale che $\tau(s) = s$. Il teorema di ricursione mostra che, data una funzione ricorsiva totale t, la trasformazione $\tau : R \mapsto R$ (R insieme delle funzioni ricorsive) definita come $\tau(\varphi_i) = \varphi_{t(i)}$, ammette un punto fisso $\varphi_e = \varphi_{t(e)} = \tau(\varphi_e)$.

Dimostrazione. Senza perdita di generalità, dimostriamo il teorema per una qualunque enumerazione delle macchine di Turing: la sua estensione al caso di enumerazioni qualunque di funzioni è immediata. Consideriamo una enumerazione delle macchine di Turing e definiamo $M[u]$ una famiglia di macchine con parametro u avente il seguente comportamento:
"*con input x la macchina $M[u]$ calcola $z = \varphi_u(u)$ e se $\varphi_u(u)$ è definita calcola $\varphi_z(x)$*".

In altre parole $M[u]$ calcola la funzione parziale

$$\varphi_{g(u)}(x) = \begin{cases} \varphi_{\varphi_u(u)}(x) & \text{se } \varphi_u(u) \text{ è definita} \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Sia ora $v \in \mathbb{N}$ un valore tale che $\varphi_v = \lambda x.t(g(x))$. In altre parole, per individuare v, generiamo la macchina $\mathcal{M}$ che calcola la funzione $\lambda x.t(g(x))$ e determiniamo l'indice di $\mathcal{M}$. Abbiamo pertanto che, per ogni $x \in \mathbb{N}$,

$$\varphi_{g(v)}(x) = \varphi_{\varphi_v(v)}(x) = \varphi_{t(g(v))}(x)$$

Ponendo $e = g(v)$ abbiamo $\varphi_e = \varphi_{t(e)}$. □

Il teorema di ricursione è uno strumento molto importante ai fini della dimostrazione di risultati che riguardano le funzioni calcolabili. Mostriamo ad esempio un interessante risultato, la cui dimostrazione si basa sul risultato precedente, e che può essere interpretato come segue: in ogni modello di calcolo esiste un programma che fornisce costantemente in output il proprio codice.

Corollario 7.14 *Data una enumerazione delle funzioni ricorsive, esiste un indice i tale che, per ogni $x \in \mathbb{N}$, $\varphi_i(x) = i$.*

Dimostrazione. Consideriamo la funzione di proiezione $f(x, y) = x$. Poiché in base al Teorema *s-m-n* esiste una funzione ricorsiva totale s tale che $\varphi_{s(x)}(y) = f(x, y) = x$, utilizzando il teorema di ricursione possiamo affermare che esiste un valore i per cui, per ogni y, $\varphi_i(y) = \varphi_{s(i)}(y) = i$. □

Esercizio 7.10 Dimostrare che per ogni enumerazione dei programmi di un linguaggio di programmazione esiste un valore $i \in \mathbb{N}$ tale che l'i-esimo programma e l'$(i + 1)$-esimo programma calcolano la medesima funzione.

Tra le applicazione del teorema di ricursione la più rilevante è il teorema di Rice, che asserisce che qualunque proprietà non banale delle funzioni calcolabili è indecidibile.

Teorema 7.15 (Rice) *Sia data una qualunque enumerazione delle funzioni ricorsive e sia F un insieme di funzioni calcolabili. L'insieme $S = \{x \mid \varphi_x \in F\}$ è decidibile se e solo se $F = \emptyset$ oppure F coincide con l'intera classe delle funzioni calcolabili.*

Dimostrazione. Senza perdita di generalità, dimostriamo il teorema nel caso di una enumerazione delle macchine di Turing. In tal caso l'insieme S contiene dunque tutti gli indici delle macchine che calcolano funzioni appartenenti ad F. Supponiamo che $F \neq \emptyset$ ed F, al tempo stesso, non coincidente con l'intera classe delle funzioni calcolabili, e che, per assurdo, S sia decidibile. Siano i e j rispettivamente il primo indice appartenente ad S ed il primo indice non appartenente ad S. Consideriamo la seguente funzione:

$$C(x) = \begin{cases} i & \text{se } x \notin S \\ j & \text{altrimenti} \end{cases}$$

Poiché la funzione C è totale, per il teorema di ricursione esiste e tale che $\varphi_{C(e)} = \varphi_e$. Per definizione, se $C(e) = i$ allora $e \notin S$ e quindi $\varphi_e \notin F$, ma poiché per ipotesi $i \in S$ e $\varphi_i \in F$ abbiamo anche $\varphi_{C(e)} = \varphi_i = \varphi_e \in F$ e ciò determina una contraddizione. D'altra parte è immediato verificare che anche nel caso $C(e) = j$ si ottiene una contraddizione. □

Nella Sezione 7.4 abbiamo mostrato diversi esempi di funzioni non calcolabili che possono essere interpretati come esempi di indecidibilità di corrispondenti insiemi di programmi. Ad esempio, dimostrare che la funzione

$$g(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è costante} \\ 0 & \text{altrimenti} \end{cases}$$

non è calcolabile, come fatto in quella sezione, equivale a dimostrare la non decidibilità dell'insieme $S = \{x \mid \varphi_x \in F\}$, dove F è l'insieme delle funzioni calcolabili totali con valore costante. Come si può facilmente osservare, questo risultato non è che una semplice applicazione del Teorema di Rice. Tale teorema consente in effetti di dimostrare in termini generali risultati di non decidibilità per ciascuno dei quali si renderebbero altrimenti necessarie tecniche ad hoc.

L'interpretazione del Teorema di Rice in termini di teoria della programmazione è particolarmente significativa. Infatti, in tale contesto, dal teorema consegue l'impossibilità di provare specifiche proprietà (costanza, crescenza ecc.) delle funzioni calcolate da programmi. In particolare, un'interessante applicazione, la cui verifica viene lasciata al lettore, è quella che permette di stabilire l'indecidibilità della correttezza di un programma.

Esercizio 7.11 Data una enumerazione delle funzioni ricorsive e data una funzione ricorsiva f, dimostrare che la funzione

$$p(x) = \begin{cases} 1 & \text{se } \varphi_x = f \\ 0 & \text{altrimenti} \end{cases}$$

non è calcolabile.

7.7 Insiemi decidibili e semidecidibili

In questa ultima sezione esamineremo alcune importanti proprietà degli insiemi decidibili e semidecidibili. I concetti di decidibilità e semidecidibilità sono stati introdotti nella Sezione 5.3 con riferimento alla T-calcolabilità. Vedremo ora come tali nozioni possono essere riformulate astraendo rispetto al modello di calcolo adottato. In questo ambito prenderemo in considerazione fondamentalmente insiemi di interi ma, chiaramente, tutti i concetti formulati potranno essere applicati anche ad altre tipologie di insiemi, come ad esempio i linguaggi definiti su opportuni alfabeti.

Innanzi tutto presentiamo due nuove definizioni che ci consentono di caratterizzare gli insiemi decidibili e semidecidibili mediante le proprietà di opportune funzioni ad essi associate.

Definizione 7.1 *Un insieme $A \subseteq \mathbb{N}$ viene detto* ricorsivo *se la sua funzione caratteristica $\mathcal{C}_A$:*

$$\mathcal{C}_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{altrimenti} \end{cases}$$

è ricorsiva totale.

Definizione 7.2 *Un insieme $A \subseteq \mathbb{N}$ viene detto* ricorsivamente enumerabile *(r.e.) se $A = \emptyset$ o se esiste una funzione ricorsiva totale $f : \mathbb{N} \mapsto \mathbb{N}$ tale che $A = \text{imm}(f)$. In tal caso diciamo che la funzione f* enumera *l'insieme A.*

Nel seguito mostreremo che le classi degli insiemi ricorsivi e degli insiemi ricorsivamente enumerabili coincidono rispettivamente con le classi degli insiemi decidibili e semidecidibili e, di conseguenza, la classe degli insiemi ricorsivi è strettamente contenuta nella classe degli insiemi ricorsivamente enumerabili. Prima di dimostrare questi risultati discutiamo brevemente alcuni esempi ed alcune elementari proprietà di tali classi di insiemi.

Esempio 7.3 L'insieme dei numeri pari è ricorsivo, poichè la funzione che ci consente di decidere se un numero è pari o meno è ricorsiva totale.

Esempio 7.4 L'insieme $\{x \mid x \text{ è un numero primo}\}$ è ricorsivamente enumerabile. Infatti si può facilmente mostrare che la funzione

$$p(x) = \begin{cases} 2 & \text{se } x = 0 \\ p_{x+1} & \text{altrimenti} \end{cases}$$

dove p_i rappresenta l'i-esimo numero primo, è una funzione ricorsiva totale che enumera l'insieme stesso.

Esempio 7.5 L'insieme $\{\langle y, t\rangle \mid \varphi_y(y) \text{ si arresta in meno di } t \text{ passi}\}$ è ricorsivo (basta eseguire t passi di $\varphi_y(y)$ e verificare).

Esercizio 7.12 Dimostrare che l'insieme $\{\langle i, x, t\rangle \mid t$ è il codice della computazione eseguita dalla MT di indice i con input $x\}$ è ricorsivo.

L'esempio che segue illustra invece casi di insiemi non ricorsivi.

Esempio 7.6 Consideriamo i seguenti insiemi.

(a) L'insieme $K = \{y \mid \varphi_y(y)$ è definita$\}$ non è ricorsivo (Corollario 7.6).

(b) L'insieme $Z = \{\langle x, y, z\rangle \mid \varphi_x(y) = z\}$ non è ricorsivo (Corollario 7.10).

(c) L'insieme $T = \{x \mid \varphi_x$ è totale$\}$ non è ricorsivo (Teorema 7.12).

Con riferimento agli esempi precedenti, possiamo osservare che nel caso dell'insieme dei numeri primi, l'insieme stesso è anche ricorsivo. Al contrario, gli insiemi K e Z costituiscono esempi di insiemi non ricorsivi ma ricorsivamente enumerabili. Per poter dimostrare tali asserzioni sarà tuttavia necessario utilizzare tecniche più raffinate che illustreremo successivamente. Infine, vedremo che l'insieme T non solo non è ricorsivo, ma non è neanche ricorsivamente enumerabile.

Consideriamo ora alcune proprietà degli insiemi ricorsivi. In particolare, la prima proprietà che dimostriamo è l'equivalenza del concetto di decidibilità e di ricorsività di un insieme.

Teorema 7.16 *Un insieme $S \subseteq \mathbb{N}$ è ricorsivo se e solo se esso è decidibile.*

Dimostrazione. È sufficiente osservare che la funzione caratteristica $\mathcal{C}_A$ di un insieme A è ricorsiva e totale se e solo se esiste una MT che accetta ogni input x per cui la funzione $\mathcal{C}_A(x) = 1$ e rifiuta ogni altro input. □

Teorema 7.17 *Se un insieme A è ricorsivo allora l'insieme complemento $\bar{A} = \mathbb{N} - A$ è anch'esso ricorsivo.*

Dimostrazione. $\mathcal{C}_{\bar{A}}(x) = 1 - \mathcal{C}_A(x)$. □

Teorema 7.18 *Se gli insiemi A e B sono ricorsivi allora anche gli insiemi $A \cup B$ e $A \cap B$ sono ricorsivi.*

Dimostrazione. Si osservi infatti che:

$$\mathcal{C}_{A\cup B}(x) = \mathcal{C}_A(x) \oplus \mathcal{C}_B(x)$$
$$\mathcal{C}_{A\cap B}(x) = \mathcal{C}_A(x) \cdot \mathcal{C}_B(x)$$

dove $\oplus$ denota la somma logica, definita come $x \oplus y = 1$ se $x = 1$ o $y = 1$, e $x \oplus y = 0$ altrimenti. □

Esercizio 7.13 Verificare che l'insieme degli insiemi ricorsivi è un'algebra di Boole.

Per quanto riguarda gli insiemi ricorsivamente enumerabili, possiamo innanzi tutto mostrare, con il teorema seguente, quale sia la loro relazione con gli insiemi semidecidibili.

Teorema 7.19 *Sia dato un insieme $S \subseteq \mathbb{N}$: le seguenti affermazioni sono equivalenti.*

1. *S è ricorsivamente enumerabile;*
2. *S è semidecidibile;*
3. *S è il dominio di definizione di una funzione g_S parziale calcolabile;*
4. *S è l'immagine di una funzione h_S parziale calcolabile.*

Dimostrazione. La dimostrazione sarà fornita mostrando che ogni affermazione è implicata dalla precedente e che l'affermazione 4 implica a sua volta la 1.

1. $\longrightarrow$ 2. Per ipotesi, o $S = \emptyset$ oppure esiste una funzione $f_S : \mathbb{N} \mapsto \mathbb{N}$ che lo enumera. Nel primo caso, l'insieme è ovviamente semidecidibile, mentre, nel secondo caso, possiamo definire una macchina di Turing $\mathcal{M}$ che opera nel seguente modo: dato un input x la macchina $\mathcal{M}$ enumera l'insieme S calcolando via via la funzione f_S sugli argomenti $0, 1, 2, \ldots$ e verificando se x viene generato dall'applicazione della funzione f_S ad un qualche argomento $y \in \mathbb{N}$. In tal caso, e solo in tal caso, $\mathcal{M}$ accetta x.

2. $\longrightarrow$ 3. Sia $\mathcal{M}$ una macchina di Turing che accetta S. Possiamo allora definire la funzione g_S nel seguente modo:

$$g_S(x) = \begin{cases} 1 \text{ se } \mathcal{M} \text{ accetta } x \\ \text{indefinito altrimenti.} \end{cases}$$

   Chiaramente, la funzione parziale g_S è calcolabile e il suo dominio di definizione coincide con S.

3. $\longrightarrow$ 4. Sia g_S una funzione parziale calcolabile con dominio di definizione S. Possiamo allora definire la funzione h_S nel seguente modo:

$$h_S(x) = \begin{cases} x \text{ se } g_s(x) = 1 \\ \text{indefinito altrimenti.} \end{cases}$$

   Chiaramente, la funzione parziale h_S è anch'essa calcolabile e la sua immagine coincide con S.

4. $\longrightarrow$ 1. Supponiamo che S contenga almeno un elemento; in caso contrario l'insieme è ricorsivamente enumerabile per definizione. Sia $\mathcal{M}_S$ una macchina di Turing che calcola la funzione parziale h_S avente per immagine l'insieme S. Definiamo ora una macchina di Turing $\mathcal{M}$ che opera, per fasi, nel seguente modo: in corrispondenza alla fase i-esima ($i > 0$), $\mathcal{M}$ simula i passi delle computazioni effettuate da $\mathcal{M}_S$ sugli input $0, \ldots, i-1$. Per ogni computazione che termina (in al più i passi), restituendo quindi un valore $y \in \text{imm}(h_S)$, la macchina di Turing $\mathcal{M}$ restituisce lo stesso valore y.

   Possiamo allora definire la funzione f_S che enumera S nel modo seguente:

   $$f_S(x) = y \text{ se } y \text{ è l'}(x+1)\text{-esimo valore restituito da } \mathcal{M}$$

   Chiaramente la funzione così definita è una funzione totale calcolabile e la sua immagine coincide, per costruzione, con l'insieme S.

□

Una immediata conseguenza del risultato precedente è che i linguaggi di tipo 0 sono ricorsivamente enumerabili. Infatti, come mostrato nel Corollario 5.10, essi sono semidecidibili. Una ulteriore conseguenza è che, dato che gli insiemi ricorsivi sono decibili (vedi Teorema 7.16) e la classe degli insiemi decidibili è strettamente inclusa nella classe degli insiemi semidecidibili, allora ogni insieme ricorsivo è anche ricorsivamente enumerabile ed esistono insiemi ricorsivamente enumerabili ma non ricorsivi.

Esempio 7.7 L'insieme $K = \{x \mid \varphi_x(x) \text{ è definita}\}$ è ricorsivamente enumerabile, ma non ricorsivo. Infatti, l'insieme K non è ricorsivo, come deriva dal Corollario 7.6, mentre, per mostrare che esso è ricorsivamente enumerabile è sufficiente osservare che $K = \text{dom}(\psi)$ se si definisce:

$$\psi(x) = \begin{cases} 1 \text{ se } \varphi_x(x) \text{ è definita} \\ \text{indefinita, altrimenti.} \end{cases}$$

Esercizio 7.14 Mostrare che un insieme infinito $A \subseteq \mathbb{N}$ è enumerabile in ordine crescente se e solo se esso è ricorsivo.

Osserviamo ora che esistono insiemi non ricorsivamente enumerabili.

Teorema 7.20 *L'insieme* $T = \{x \mid \varphi_x \text{ è totale}\}$ *non è ricorsivamente enumerabile.*

Dimostrazione. Supponiamo, per assurdo, che l'insieme sia ricorsivamente enumerabile e che f sia la sua funzione enumeratrice. Potremmo allora costruire una funzione:

$$h(x) = \varphi_{f(x)}(x) + 1.$$

Sia $\overline{y}$ un suo indice. Poiché h è totale deve esistere $\overline{x}$ tale che $f(\overline{x}) = \overline{y}$. Allora si ha nuovamente la contraddizione:

$$h(\overline{x}) = \varphi_{\overline{y}}(\overline{x}) \qquad e \qquad h(\overline{x}) = \varphi_{\overline{y}}(\overline{x}) + 1.$$

□

Un elementare ragionamento basato sulla cardinalità ci può convincere del fatto che la classe degli insiemi non ricorsivamente enumerabili corrisponde alla "quasi totalità" della classe degli insiemi di interi. Infatti, la classe di tutti gli insiemi di interi ha cardinalità $2^{\mathbb{N}}$ mentre l'insieme delle funzioni ricorsive parziali ha cardinalità $\mathbb{N}$ e quindi le funzioni ricorsive totali, che sono un sottoinsieme di quelle ricorsive parziali, avranno anch'esse al più cardinalità $\mathbb{N}$.

Prima di mostrare ulteriori esempi di insiemi non ricorsivamente enumerabili, esaminiamo rispetto a quali operazioni insiemistiche la classe degli insiemi ricorsivamente enumerabili risulti chiusa.

Teorema 7.21 *Se gli insiemi $A \subseteq \mathbb{N}$ e $B \subseteq \mathbb{N}$ sono ricorsivamente enumerabili allora anche gli insiemi $A \cup B$ e $A \cap B$ sono ricorsivamente enumerabili.*

Dimostrazione. È sufficiente osservare che, se $A = \mathrm{dom}(\psi_A)$, $B = \mathrm{dom}(\psi_B)$, allora basta definire:

$$g_{A\cup B}(x) = \begin{cases} 1 \text{ se } \psi_A(x) \text{ è definito o } \psi_B(x) \text{ è definito} \\ \textit{indefinita}, \text{ altrimenti} \end{cases}$$

$$g_{A\cap B}(x) = \begin{cases} 1 \text{ se } \psi_A(x) \text{ è definito e } \psi_B(x) \text{ è definito} \\ \textit{indefinita}, \text{ altrimenti} \end{cases}$$

per avere $A \cup B = \mathrm{dom}(g_{A\cup B})$ e $A \cap B = \mathrm{dom}(g_{A\cap B})$. □

La proprietà di chiusura, contrariamente al caso degli insiemi ricorsivi, non vale invece per l'operazione di complementazione. Questo fatto deriva dal risultato seguente.

Teorema 7.22 *Sia $A \subseteq \mathbb{N}$ un insieme ricorsivamente enumerabile e sia ricorsivamente enumerabile anche il suo complemento $\bar{A}$; allora A è ricorsivo.*

Dimostrazione. Se $A = \mathrm{dom}(\psi_A)$ ed $\bar{A} = \mathrm{dom}(\psi_{\bar{A}})$ possiamo definire la funzione caratteristica di A :

$$f_A(x) = \begin{cases} 1 & \text{se } \psi_A(x) \text{ è definita} \\ 0 & \text{se } \psi_{\bar{A}}(x) \text{ è definita} \end{cases}$$

e poiché una delle due alternative si verifica sicuramente, f_A è ricorsiva totale. □

Come conseguenza immediata del teorema precedente, se un insieme è ricorsivamente enumerabile, ma non ricorsivo, allora il suo complemento non può essere ricorsivamente enumerabile.

Esempio 7.8 L'insieme

$$\overline{K} = \{x \mid \varphi_x(x) \text{ non è definita}\}$$

non è ricorsivamente enumerabile. Infatti sappiamo che K è semidecidibile e quindi ricorsivamente enumerabile. D'altra parte esso non è ricorsivo e quindi il suo complemento non può essere ricorsivamente enumerabile.

Il Teorema 7.22 mette sostanzialmente in luce che il complemento di un insieme ricorsivamente enumerabile, ma non ricorsivo, ha delle proprietà di indecidibilità più elevate dell'insieme stesso.

A conferma di questo fatto possiamo verificare che, ad esempio, per rispondere alla domanda "x appartiene a K?" dobbiamo verificare se esiste y tale che $\varphi_x(x)$ si arresta in meno di y passi, per rispondere alla domanda "x appartiene a $\overline{K}$?" dobbiamo verificare se *per ogni* y, $\varphi_x(x)$ richiede più di y passi: è chiaro che nel primo caso una risposta affermativa può essere data in un tempo finito, mentre nel secondo caso ciò non può avvenire.

È interessante osservare, peraltro, che in tutti e due i casi il predicato di cui si richiede la verifica, cioè i predicati "$\varphi_x(x)$ richiede meno di y passi" e "$\varphi_x(x)$ richiede più di y passi", sono decidibili, mentre il predicato "esiste y tale che $\varphi_x(x)$ richiede meno di y passi" è ricorsivamente enumerabile, e il predicato "per ogni y $\varphi_x(x)$ richiede più di y passi" è addirittura non ricorsivamente enumerabile.

In effetti, si può verificare che tale fenomeno vale in generale, come mostrato dai seguenti teoremi.

Teorema 7.23 *Sia $A \subseteq \mathbb{N}$ un insieme non vuoto. A è allora ricorsivamente enumerabile se e solo se esiste un insieme ricorsivo $B \subseteq \mathbb{N}^2$ tale che $x \in A$ se e solo se $\exists y[(x, y) \in B]$.*

Dimostrazione. Sia f la funzione che enumera A; definiamo $B = \{x, y \mid f(y) = x\}$. B è chiaramente ricorsivo poiché f è totale. Ciò prova che il fatto che A è ricorsivamente enumerabile è condizione sufficiente per l'esistenza dell'insieme ricorsivo B. Per dimostrare che la condizione è anche necessaria, supponiamo che sia dato un qualunque insieme ricorsivo $B \subseteq \mathbb{N}^2$ e che f_B sia la sua funzione caratteristica. Abbiamo allora che se definiamo la funzione g_A nel seguente modo:

$$g_A(x) = \begin{cases} 1 \text{ se esiste } y \text{ tale che } f_B(x, y) = 1 \\ \text{indefinita, altrimenti.} \end{cases}$$

allora $A = \text{dom}(g_A)$. □

Teorema 7.24 *Sia $A \subseteq \mathbb{N}$. A è allora il complemento di un insieme ricorsivamente enumerabile se e solo se esiste un insieme ricorsivo $B \subseteq \mathbb{N}^2$ tale che $x \in A$ se e solo se $\forall y[(x, y) \in B]$.*

Dimostrazione. La dimostrazione viene lasciata come esercizio (vedi Esercizio 7.15). □

Esercizio 7.15 Dimostrare il Teorema 7.24.

Quanto mostrato nei Teoremi 7.23 e 7.24 può essere esteso al caso in cui vengano utilizzati più quantificatori esistenziali ed universali alternati, secondo le seguenti definizioni.

Definizione 7.3 *Per ogni insieme $A \subseteq \mathbb{N}$, $A \in \Sigma_k$ se e solo se esiste un predicato ricorsivo $P(x, y_1, \ldots, y_k)$ tale che $x \in A$ se e solo se $\exists y_1 \forall y_2 \ldots Q y_k\ P(x, y_1, \ldots, y_k)$, dove $Q = \exists$ per k dispari, $Q = \forall$ per k pari.*

Definizione 7.4 *Per ogni insieme $A \subseteq \mathbb{N}$, $A \in \Pi_k$ se e solo se esiste un predicato ricorsivo $P(x, y_1, \ldots, y_k)$ tale che $x \in A$ se e solo se $\forall y_1 \exists y_2 \ldots Q y_k\ P(x, y_1, \ldots, y_k)$, dove $Q = \forall$ per k dispari, $Q = \exists$ per k pari.*

Definizione 7.5 *Per ogni insieme $A \subseteq \mathbb{N}$, $A \in \Delta_k$ se e solo se $A \in \Sigma_k$ e $A \in \Pi_k$.*

Dalle definizione precedenti deriva immediatamente che, dato un insieme $A \subseteq \mathbb{N}$, $A \in \Sigma_n$ se e solo se $\overline{A} \in \Pi_n$.

L'insieme delle classi Σ_k e Π_k viene chiamato *gerarchia aritmetica* (o *gerarchia di Kleene*).

Chiaramente la classe Σ_0 e Π_0 della gerarchia aritmetica coincidono con la classe degli insiemi ricorsivi, mentre la classe Σ_1 coincide con la classe degli insiemi ricorsivamente enumerabili e la classe Π_1 coincide con la classe degli insiemi che sono complemento di insiemi ricorsivamente enumerabili.

Due proprietà fondamentali della gerarchia aritmetica sono le seguenti:

i) $\forall i\ [\Sigma_i \subsetneq \Sigma_{i+1}]$ e $\forall i\ [\Pi_i \subsetneq \Pi_{i+1}]$

ii) $\forall i\ [\Sigma_i \cup \Pi_i \subset \Sigma_{i+1} \cap \Pi_{i+1}]$.

Possiamo utilizzare la gerarchia aritmetica per caratterizzare il livello di indecidibilità cui appartiene un insieme. Ad esempio, l'insieme $T = \{x \mid \varphi_x$ è totale$\}$ coincide con l'insieme $\{x \mid \forall y \exists k[\varphi_x(y)$ richiede meno di k passi$]\}$, e quindi l'insieme delle funzioni ricorsive totali appartiene a Π_2. Una illustrazione dei primi livelli della gerarchia aritmetica è data in Figura 7.2.

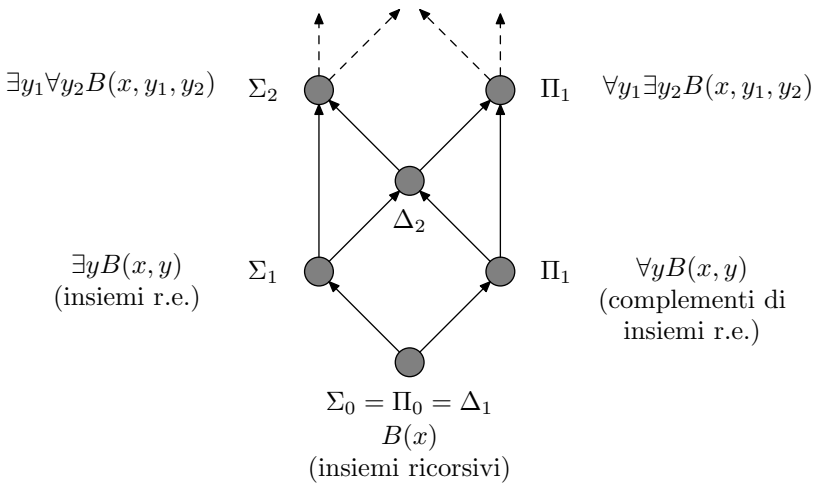


FIGURA 7.2 Gerarchia di Kleene.

Capitolo 8

Teoria della Complessità

Come si è visto nei capitoli precedenti, la Teoria della Calcolabilità si interessa dell'esistenza o meno di algoritmi per la soluzione di specifici problemi, oltre che della relazione (in termini di possibilità o meno di soluzione dei problemi stessi) tra i diversi modelli di calcolo. Una volta che un problema è stato riconosciuto come risolubile da un determinato modello di calcolo - tipicamente da macchine di Turing - la Teoria della Calcolabilità non è interessata a determinare il *costo* di soluzione del problema in termini di *risorse di calcolo* utilizzate. Ad esempio, un determinato problema può risultare risolubile ma richiedere, almeno su certe istanze, un tempo intollerabilmente lungo per il calcolo della soluzione, risultando così non risolubile in pratica, se non su istanze particolarmente semplici.

In questo senso, la Teoria della Calcolabilità si limita a considerare aspetti *qualitativi* della risolubilità di problemi, limitandosi, nell'ambito di un dato modello di calcolo, a distinguere ciò che è risolubile da ciò che non lo è.

La Teoria della Complessità, al contrario, si occupa della caratterizzazione e della classificazione dei problemi (risolubili, per ipotesi) dal punto di vista della quantità di risorse di calcolo necessarie per risolverli. In particolare, la Teoria della Complessità investiga la quantità di risorse necessarie (*lower bound*) e/o sufficienti (*upper bound*) per risolvere i vari problemi.

Le più naturali risorse di calcolo considerate nell'ambito della Teoria della Complessità sono il tempo e la memoria utilizzati da un algoritmo per la risoluzione di un problema.

Naturalmente, la quantità precisa di risorse utilizzate in pratica nella risoluzione dipenderà dal modello di calcolo considerato e dalle caratteristiche strutturali e tecnologiche delle macchine utilizzate per l'esecuzione di un algoritmo.

Al fine di poter disporre di una classificazione dei problemi basata sulla loro intrinseca difficoltà, sarà quindi necessario fare riferimento a modelli

astratti, che permettano di prescindere da caratteristiche realizzative (come ad esempio la RAM che, come abbiamo visto, è una ragionevole astrazione di un usuale calcolatore). In questo modo potremo classificare i problemi in termini di risorse di calcolo richieste per la loro soluzione su un modello di calcolo particolare. Potremo fare ad esempio riferimento ai problemi risolubili con macchine di Turing non deterministiche in spazio lineare (si ricordi che tale classe include il riconoscimento di linguaggi di tipo 1) o ai problemi risolubili in tempo cubico con una RAM.

Tra i risultati più importanti che la Teoria della Complessità intende perseguire c'è in particolare la caratterizzazione dei problemi computazionalmente trattabili: problemi cioè che possono essere efficientemente risolti con algoritmi che richiedono un tempo di calcolo polinomiale. Come vedremo, un problema può essere classificato come trattabile senza fare riferimento ad un particolare modello di calcolo.

Ciò è reso possibile dal fatto che molti modelli generali di computazione danno luogo, in larga misura e per la maggior parte dei casi più interessanti, alla medesima classificazione. Per tale motivo nel seguito si farà uso, come modello di calcolo di riferimento, della Macchina di Turing a più nastri introdotta nella Sezione 5.4, sfruttando, a tal fine, una versione particolare della Tesi di Church-Turing, che asserisce che ogni dispositivo di calcolo naturale può essere simulato da una macchina di Turing a più nastri con un costo per la simulazione di ogni operazione al più polinomiale nella dimensione del problema. Un esempio di tale relazione è stato dato in Sezione 6.3, dove si è illustrato come macchine di Turing deterministiche e RAM siano simulabili tra loro in tempo polinomiale.

8.1 Valutazioni di complessità

L'obiettivo principale della Teoria della Complessità è dunque la caratterizzazione di insiemi di problemi in funzione delle risorse di calcolo (spazio, tempo) necessarie alla loro risoluzione: questi insiemi prendono il nome di *classi di complessità*.

Indichiamo con $\hat{t}_A(x)$ il tempo di esecuzione dell'algoritmo A su input x, inteso come numero di passi elementari effettuati . Il *tempo di esecuzione nel caso peggiore* (worst case) di A sarà allora il tempo utilizzato da A sull'istanza più difficile e sarà espresso come $t_A(n) = \max\{\hat{t}_A(x) \mid \forall x : |x| \leq n\}$, dove $|x|$ indica la dimensione di x, intesa come lunghezza della descrizione dell'istanza stessa. Quindi $t_A(n)$ rappresenta il tempo di esecuzione di A sull'istanza di lunghezza al più n (secondo una qualche codifica) per il quale tale tempo di esecuzione è massimo.

Corrispondentemente, sia $\hat{s}_A(x)$ lo spazio di memoria utilizzato dall'algoritmo A su input x. Lo *spazio di esecuzione nel caso peggiore* di A, definito come lo spazio utilizzato da A sull'istanza peggiore, è dato da $s_A(n) = \max\{\hat{s}_A(x) \mid \forall x : |x| \leq n\}$.

Nel caso di complessità spaziale si farà sempre riferimento allo spazio di lavoro *addizionale*, cioè alla quantità di celle di nastro (in una macchina di Turing) o di locazioni di memoria utilizzate per eseguire la computazione, non considerando lo spazio richiesto per la descrizione iniziale dell'istanza in questione. A tal fine, assumiamo che la descrizione dell'istanza sia rappresentata su una memoria di sola lettura (read-only) la cui dimensione non viene considerata nella valutazione della complessità spaziale, e a cui sia possibile accedere in modo *one-way*, muovendosi cioè dall'inizio alla fine, in modo tale che la descrizione dell'input possa essere letta una sola volta, per mezzo di un'unica scansione.

Data una istanza di un problema, la lunghezza cui si fa riferimento in Teoria della Complessità è definita come il numero di caratteri necessari per descrivere tale istanza nell'ambito di un qualche metodo di codifica predefinito. In tal modo, naturalmente, sia la lunghezza associata ad una istanza che la complessità di un algoritmo operante sull'istanza stessa saranno dipendenti dal metodo di codifica adottato.

Definizione 8.1 *Dato un problema $\mathcal{P}$ con insieme $I_{\mathcal{P}}$ delle possibili istanze, e dato un alfabeto Σ, definiamo come* schema di codifica *di $\mathcal{P}$ una funzione $e : I_{\mathcal{P}} \mapsto \Sigma^*$ che mappa ogni istanza di $\mathcal{P}$ in una corrispondente stringa di simboli in Σ^*.*

Diciamo che uno schema di codifica è *ragionevole* se non vi sono istanze la cui descrizione è artificiosamente lunga. Ciò significa, ad esempio:

- considerare codifiche in base $k \geq 2$ dei valori numerici interessati;
- rappresentare insiemi mediante enumerazione delle codifiche dei loro elementi;
- rappresentare relazioni e funzioni mediante enumerazione delle codifiche dei relativi elementi (coppie e n-ple);
- rappresentare grafi come coppie di insiemi (di nodi ed archi).

In generale, tra tutti gli schemi di codifica ragionevoli vale la seguente condizione di correlazione polinomiale.

Definizione 8.2 *Due schemi di codifica e, e' per un problema $\mathcal{P}$ sono* polinomialmente correlati *se esistono due polinomi p, p' tali che, per ogni $x \in I_{\mathcal{P}}$:*

- $|e(x)| \leq |p'(e'(x))|$
- $|e'(x)| \leq |p(e(x))|$

Essenzialmente, quindi, e ed e' sono polinomialmente correlati se per ogni istanza le lunghezze delle relative codifiche nei due schemi sono "non troppo diverse". Si può comprendere, già a livello intuitivo, come un algoritmo che utilizza tempo (o spazio) polinomiale per risolvere un problema $\mathcal{P}$ le cui istanze siano codificate per mezzo di e utilizzi tempo (o spazio) polinomiale per risolvere $\mathcal{P}$ anche nel caso in cui le istanze siano codificate per mezzo di e'.

Esempio 8.1 L'uso di schemi di codifica "non ragionevoli" può portare a valutazioni di complessità paradossali.

Ad esempio, si consideri il problema di decidere, dato un naturale x, se esso è primo. Il semplice algoritmo consistente nel tentare di dividere x per tutti i naturali $i < x$ (e dedurre che x è primo se per tutti questi valori la divisione fornisce resto non nullo) esegue $O(x)$ passi (assumendo, per semplicità, che una divisione sia eseguita in tempo costante rispetto al valore dei relativi operandi) e risulta quindi esponenziale per una codifica ragionevole, quale ad esempio quella più naturale consistente nel rappresentare x per mezzo di $\lceil \log_2(x+1) \rceil$ bit.

Consideriamo, al contrario, l'uso di una codifica non ragionevole, quale ad esempio quella unaria, che utilizza un alfabeto di un solo carattere "|" e rappresenta l'intero n per mezzo di una stringa di n caratteri "|". In tal caso, l'algoritmo precedente risulta avere complessità polinomiale (lineare, addirittura).

Si osservi che le due codifiche non sono polinomialmente correlate.

Dalla proprietà di correlazione polinomiale valida tra tutti gli schemi di codifica ragionevoli deriva che la complessità di un algoritmo risulta invariante (a meno di un polinomio) al variare dello schema utilizzato. Per tutti i problemi che saranno considerati in seguito, tutti gli schemi di codifica più naturali risultano ragionevoli, per cui, in generale, si assumerà implicitamente di far uso di uno di tali schemi e non si farà riferimento ad aspetti relativi alla codifica di istanze.

Infine, osserviamo che in generale l'analisi della complessità di un dato problema ha luogo mediante valutazioni separate delle quantità di risorse necessarie (lower bound) e sufficienti (upper bound) per la risoluzione del problema stesso. Soltanto nel momento in cui, eventualmente, queste misure dovessero coincidere avremmo una valutazione esatta della complessità stessa.

Inoltre, per semplicità, l'analisi viene effettuata *asintoticamente*, vale a dire che quel che interessa caratterizzare è come la complessità di risoluzione del problema cresca al tendere all'infinito della dimensione delle relative istanze.

Da tali considerazioni deriva la seguente definizione.

Definizione 8.3 *Dato un problema $\mathcal{P}$ diciamo che:*

1. *$\mathcal{P}$ ha* delimitazione superiore di complessità temporale *(upper bound) $O(g(n))$ se esiste un algoritmo A che risolve $\mathcal{P}$ tale che $\hat{t}_A(n) = O(g(n))$;*
2. *$\mathcal{P}$ ha* delimitazione inferiore di complessità temporale *(lower bound) $\Omega(g(n))$ se per ogni algoritmo A che risolve $\mathcal{P}$ si ha $\hat{t}_A(n) = \Omega(g(n))$;*
3. *$\mathcal{P}$ ha* complessità temporale (esatta) *$\Theta(g(n))$ se ha upper bound $O(g(n))$ e lower bound $\Omega(g(n))$.*

Definizioni simili possono essere introdotte per lo spazio utilizzato.

Esempio 8.2 Se consideriamo ad esempio il classico problema dell'ordinamento di una sequenza di n interi, abbiamo che esso presenta, come noto, un upper bound sulla complessità temporale $O(n \log n)$ e lower bound sulla stessa complessità $\Omega(n \log n)$, per cui la complessità temporale esatta di tale problema è $\Theta(n \log n)$.

8.2 Tipi di problemi

Come detto poco sopra, la Teoria della Complessità si interessa in generale della difficoltà di soluzione di problemi o, alternativamente, di calcolo di funzioni. Ai fini di un miglior trattamento formale, risulta però conveniente introdurre una prima classificazione di tali problemi che tenga conto del tipo di questione posta da ogni specifico problema.

Si noti, ad esempio, che per ogni funzione (eventualmente multivalore) $f : \mathcal{I} \mapsto 2^{\mathcal{O}}$, si possono porre almeno quattro specifici problemi:

- dati $x \in \mathcal{I}$, si vuole determinare se esiste $y \in f(x)$
- dato $x \in \mathcal{I}$, si vuole un qualche $y \in f(x)$.
- dato $x \in \mathcal{I}$, si vuole conoscere $|\, f(x) \,|$.
- dato $x \in \mathcal{I}$, si vuole la $y \in f(x)$ "migliore" secondo una qualche misura data.

Da tali considerazioni deriva una caratterizzazione dei problemi in termini di problemi di decisione, ricerca, enumerazione ed ottimizzazione, rispettivamente.

I problemi di decisione rappresentano la tipologia di problemi più "semplici", nel senso che la loro caratterizzazione richiede il minor numero di concetti introdotti.

Definiamo un problema di decisione $\mathcal{P}_D$ per mezzo di:

1. un insieme di istanze $I_{\mathcal{P}_D}$;
2. un predicato $\pi : I_{\mathcal{P}_D} \mapsto \{\text{VERO}, \text{FALSO}\}$ che determina una partizione $I_{\mathcal{P}_D} = Y_{\mathcal{P}_D} \cup N_{\mathcal{P}_D}$, $Y_{\mathcal{P}_D} \cap N_{\mathcal{P}_D} = \emptyset$, in un insieme $Y_{\mathcal{P}_D}$ di istanze *positive*, tali cioè che $x \in Y_{\mathcal{P}_D}$ se e solo se $\pi(x) = \text{VERO}$, ed in un insieme $N_{\mathcal{P}_D}$ di istanze *negative*, tali che $x \in N_{\mathcal{P}_D}$ se e solo se $\pi(x) = \text{FALSO}$.

Si noti come tutti i problemi consistenti nel riconoscimento di un linguaggio L studiati in precedenza siano problemi di decisione, dove $I_{\mathcal{P}_D} = \Sigma^*$ e $Y_{\mathcal{P}_D} = L$.

Se consideriamo inoltre una qualsiasi codifica e delle istanze di $\mathcal{P}_D$ mediante stringhe su un qualche alfabeto Σ, possiamo poi associare a $\mathcal{P}_D$ il problema equivalente di riconoscere il linguaggio $L = \{a \in \Sigma^* \mid \exists x \in Y_{\mathcal{P}_D}, a = e(x)\}$ costituito da tutte le stringhe che descrivono istanze positive di $\mathcal{P}_D$.

Si noti come l'insieme di stringhe $\Sigma^* - L$ comprenda sia le descrizioni di istanze negative di $\mathcal{P}_D$ che stringhe che non rappresentano descrizioni di istanze $x \in I_{\mathcal{P}_D}$. In generale, comunque, queste ultime stringhe possono essere assunte come facilmente riconoscibili: possiamo assumere cioè che il problema di decidere, data una stringa $a \in \Sigma^*$, se essa sia descrizione di una istanza di $\mathcal{P}_D$ mediante lo schema di codifica e sia un problema risolubile in modo efficiente (in un numero di passi polinomiale nella lunghezza della stringa).

Un algoritmo per un problema di decisione $\mathcal{P}_D$ calcola il predicato π prendendo in input una istanza $x \in I_{\mathcal{P}_D}$, o più esattamente la sua descrizione secondo uno schema di codifica prefissato, e restituendo in output un valore VERO, FALSO corrispondente a $\pi(x) = T$ o $\pi(x) = F$, rispettivamente.

Esempio 8.3 Il problema di decisione CRICCA è definito come segue[1]:

CRICCA
ISTANZA: Grafo $G = (V, E)$, intero $K > 0$.
PREDICATO: Esiste $V' \subseteq V$ con $\mid V' \mid \geq K$ e tale che per ogni coppia $u, v \in V'$ $(u, v) \in E$?

Dato un grafo G ed un intero positivo K, un algoritmo per CRICCA restituirà quindi il valore VERO se esistono almeno K nodi in G tutti mutuamente collegati tra loro, altrimenti l'algoritmo restituirà il valore FALSO.

Si può osservare come il predicato associato ad un problema di decisione in generale sia relativo all'esistenza o meno di una qualche struttura, dipendente dall'istanza in questione, che gode di determinate proprietà. Indichiamo tale struttura come la *soluzione* dell'istanza.

Esempio 8.4 Nel caso di CRICCA, la struttura in questione è rappresentata da un sottoinsieme $V' \subseteq V$ che soddisfa il predicato associato al problema.

Si noti come, mentre in un problema di decisione ci si chiede se esiste o meno una soluzione per una istanza x, è possibile anche porsi l'obiettivo di determinare tale soluzione, se essa esiste (se cioè $x \in Y_{\mathcal{P}_D}$).

Tale situazione è formalizzata per mezzo della definizione della classe dei problemi di ricerca.

Definiamo un problema di ricerca $\mathcal{P}_R$ per mezzo di:

1. un insieme di istanze $I_{\mathcal{P}_R}$;
2. un insieme di soluzioni $S_{\mathcal{P}_R}$;
3. una *proprietà di ammissibilità*, che deve valere per tutte le soluzioni di una istanza: tale proprietà induce una relazione $R \subseteq I_{\mathcal{P}_R} \times S_{\mathcal{P}_R}$,

[1] Nel seguito, presenteremo i vari problemi omettendo di specificare che, in effetti, le relative istanze sono da considerare codificate secondo un qualche schema di codifica ragionevole: parleremo cioé ad esempio di "grafo" e non di "codifica di grafo".

detta *relazione di ammissibilità*. Data una istanza $x \in I_{\mathcal{P}_R}$, l'insieme $\mathrm{Sol}(x) = \{y \in S_{\mathcal{P}_R} \mid \langle x, y\rangle \in R\}$ viene detto insieme delle soluzioni *ammissibili* di x.

Il problema di ricerca $\mathcal{P}_R$ ha un problema di decisione associato $\mathcal{P}_D$ con $I_{\mathcal{P}_D} = I_{\mathcal{P}_R}$ e con il predicato π definito come "Esiste $y \in S_{\mathcal{P}_R}$ tale che $\langle x, y\rangle \in R$?"

Un algoritmo per un problema di ricerca $\mathcal{P}_R$ calcola una funzione (parziale) $\phi : I_{\mathcal{P}_R} \mapsto S_{\mathcal{P}_R}$ tale che $\phi(x)$ è definita se e solo se esiste almeno una soluzione $y = \phi(x)$ per la quale $\langle x, y\rangle \in R$. L'algoritmo prende quindi in input una istanza $x \in I_{\mathcal{P}_R}$, o più esattamente la sua descrizione secondo uno schema di codifica prefissato, e restituisce in output la codifica di una soluzione $y \in \mathrm{Sol}(x)$ (se una soluzione esiste).

Esempio 8.5 Il problema di ricerca RICERCA CRICCA è definito come segue:

RICERCA CRICCA
ISTANZA: Grafo $G = (V, E)$, intero $K > 0$
SOLUZIONE: Insieme di nodi V'
AMMISSIBILITÀ: $V' \subseteq V$, $\lvert V'\rvert \geq K$, $\forall\, u, v \in V'$ $(u, v) \in E$

Dato un grafo G ed un intero positivo K, un algoritmo per RICERCA CRICCA restituirà quindi, se esiste, un sottoinsieme V' di V che soddisfa la proprietà di ammissibilità, per il quale cioè vale il predicato definito in CRICCA.

I problemi di enumerazione sono fortemente correlati ai problemi di ricerca, in quanto fanno riferimento ad una medesima definizione. Infatti anche un problema di enumerazione $\mathcal{P}_E$ può essere definito mediante:

1. un insieme di istanze $I_{\mathcal{P}_E}$;
2. un insieme di soluzioni $S_{\mathcal{P}_E}$;
3. una proprietà di ammissibilità, che induce la relazione di ammissibilità $R \subseteq I_{\mathcal{P}_E} \times S_{\mathcal{P}_E}$.

Un algoritmo per un problema di enumerazione $\mathcal{P}_E$ calcola una funzione $\psi : I_{\mathcal{P}_E} \mapsto \mathbb{N}$ tale che per ogni $x \in I_{\mathcal{P}_E}$, $\psi(x) = \mid \mathrm{Sol}(x) \mid$. Il valore restituito dall'algoritmo su input x è quindi il numero di soluzioni ammissibili dell'istanza x.

Esempio 8.6 Il problema di enumerazione ENUMERA CRICCHE viene definito allo stesso modo di RICERCA CRICCA, ma quel che si vuole in questo caso è restituire un valore intero n pari al numero di soluzioni ammissibili distinte, pari cioè al numero di sottoinsiemi distinti V' che soddisfano il predicato definito in CRICCA.

Un problema di ottimizzazione $\mathcal{P}_O$ richiede per ogni istanza $x \in I_{\mathcal{P}_O}$ la restituzione della soluzione migliore associata ad x, rispetto ad una qualche misura di qualità definita sull'insieme delle coppie istanza–soluzione.

Più formalmente, definiamo un problema di ricerca $\mathcal{P}_O$ per mezzo di:

1. un insieme di istanze $I_{\mathcal{P}_O}$;
2. un insieme di soluzioni $S_{\mathcal{P}_O}$;
3. una proprietà di ammissibilità, rappresentata dalla relazione di ammissibilità $R \subseteq I_{\mathcal{P}_O} \times S_{\mathcal{P}_O}$;
4. una funzione di misura $\mu : I_{\mathcal{P}_O} \times S_{\mathcal{P}_O} \mapsto \mathbb{N}$ tale che per ogni $x \in I_{\mathcal{P}_O}$, $y \in S_{\mathcal{P}_O}$, $\mu(x, y)$ è definita se e solo se $y \in \text{Sol}(x)$;
5. un criterio di scelta $c \in \{\text{MIN}, \text{MAX}\}$.

Il problema richiede, per ogni istanza x, la soluzione $y \in \text{Sol}(x)$ tale che per ogni $z \in \text{Sol}(x)$, $\mu(x, y) \leq \mu(x, z)$ se $c = \text{MIN}$ e $\mu(x, y) \geq \mu(x, z)$ se $c = \text{MAX}$.

Esempio 8.7 Il problema di ottimizzazione MASSIMA CRICCA è definito come.

MASSIMA CRICCA
ISTANZA: Grafo $G = (V, E)$
SOLUZIONE: Insieme di nodi V'
AMMISSIBILITÀ: $V' \subseteq V$, $\forall\, u, v \in V'\ (u, v) \in E$
MISURA: $\lvert V' \rvert$
CRITERIO: MAX

8.3 Complessità temporale e spaziale

Come vedremo, buona parte della Teoria della Complessità sarà introdotta facendo riferimento ai problemi di decisione ed ai linguaggi ad essi associati. Ciò è dovuto a diversi motivi:

- Ogni problema di decisione può essere visto, indipendentemente dalla struttura delle sue istanze (numeri, stringhe, grafi, etc.), come il problema di discriminare tra due insiemi di stringhe: da una parte le codifiche di istanze in $Y_{\mathcal{P}}$ e, dall'altra, sia le stringhe che rappresentano istanze in $N_{\mathcal{P}}$ che le stringhe le quali non rappresentano istanze di $\mathcal{P}$. Come osservato sopra, dato un qualche criterio di codifica (ad esempio, numeri in rappresentazione binaria, insiemi come sequenze di identificatori associati ai relativi elementi, etc.), verificare che una data stringa rappresenti una istanza di un problema $\mathcal{P}$ nell'ambito di quel criterio di codifica è un compito semplice, che può essere risolto da un algoritmo efficiente. Questa caratteristica permette un trattamento formale unico di tutti i problemi di decisione in termini di problemi di riconoscimento di linguaggi.

- Dato che un problema di decisione ha un solo bit (VERO o FALSO) come output, nell'analisi di complessità non c'è bisogno di prestare attenzione al costo di restituzione del risultato. Tutti i costi hanno una origine completamente computazionale e non hanno nulla a che fare con la quantità di informazione in output e con il tempo necessario a restituirla.

Possiamo quindi ora definire formalmente i concetti di tempo e di spazio di computazione, facendo riferimento al riconoscimento di linguaggi da parte di macchine di Turing.

Definiamo dapprima l'accettazione di una stringa da parte di una data macchina di Turing (deterministica o non deterministica), in un numero limitato di passi.

Definizione 8.4 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica) $\mathcal{M} = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) ed un intero $\tau > 0$, una stringa $x \in \Sigma^*$ è accettata da $\mathcal{M}$ in τ passi se esiste una computazione $q_0 x \overset{}{\underset{\mathcal{M}}{\vdash\!\!\!-\!\!\!-}} c$ composta da r passi, con c configurazione finale e $r \leq \tau$.*

Possiamo quindi definire, per una macchina di Turing deterministica, la condizione di rifiuto di una stringa.

Definizione 8.5 *Dato un alfabeto Σ, una macchina di Turing deterministica $\mathcal{M}_D = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) ed un intero $\tau > 0$, una stringa $x \in \Sigma^*$ è rifiutata da $\mathcal{M}_D$ in τ passi se esiste una computazione $q_0 x \overset{}{\underset{\mathcal{M}_D}{\vdash\!\!\!-\!\!\!-}} c$ composta da r passi, con c configurazione massimale e non finale, e $r \leq \tau$.*

A partire dalle definizioni precedenti, introduciamo quindi la proprietà di *accettabilità* di un linguaggio in tempo limitato da parte di una macchina di Turing (deterministica o non deterministica).

Definizione 8.6 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica) $\mathcal{M} = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) e una funzione $t : \mathbb{N} \mapsto \mathbb{N}$, diciamo che $\mathcal{M}$* accetta *$L \subseteq \Sigma^*$ in tempo $t(n)$ se, per ogni $x \in \Sigma^*$, $\mathcal{M}$ accetta x in tempo al più $t(|\,x\,|)$ se $x \in L$, mentre, se $x \notin L$, allora $\mathcal{M}$ non la accetta.*

Infine, introduciamo la proprietà di decidibilità in un numero di passi limitato di un linguaggio da parte di una macchina di Turing deterministica.

Definizione 8.7 *Dato un alfabeto Σ, una macchina di Turing deterministica $\mathcal{M}_D = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) e una funzione $t : \mathbb{N} \mapsto \mathbb{N}$, diciamo che $\mathcal{M}_D$* riconosce *$L \subseteq \Sigma^*$ in tempo $t(n)$ se, per ogni $x \in \Sigma^*$, $\mathcal{M}_D$ accetta x in tempo al più $t(|\,x\,|)$ se $x \in L$ e rifiuta x, ancora in tempo al più $t(|\,x\,|)$, se $x \notin L$.*

Diciamo quindi che un linguaggio L è *accettabile in tempo deterministico* $t(n)$ se esiste una macchina di Turing deterministica $\mathcal{M}_D$ che accetta L in tempo $t(n)$. Similmente, diciamo che un linguaggio L è *accettabile in tempo non deterministico* $t(n)$ se esiste una macchina di Turing non deterministica $\mathcal{M}_{ND}$ che accetta L in tempo $t(n)$

Infine, diciamo che un linguaggio L è *decidibile in tempo* $t(n)$ se esiste una macchina di Turing deterministica $\mathcal{M}_D$ che decide L in tempo $t(n)$.

Per quanto riguarda la definizione della complessità spaziale, faremo riferimento al numero di celle di nastro (o nastri, se più di uno) di lavoro accedute nel corso di una computazione. Assumeremo quindi che la stringa di input sia contenuta in un nastro aggiuntivo *one way*, vale a dire un nastro di sola lettura e su cui sia possibile muovere la testina soltanto verso destra, e non terremo conto delle celle utilizzate su tale nastro per la rappresentazione dell'input, mentre conteremo le celle utilizzate sugli altri nastri. In tal modo, introduciamo le seguenti definizioni, corrispondenti a quelle date sopra per la complessità temporale.

Definizione 8.8 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica) $\mathcal{M} = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) ed un intero $\sigma > 0$, una stringa $x \in \Sigma^*$ è accettata da $\mathcal{M}$ in spazio σ se esiste una computazione $q_0 \# x \underset{\mathcal{M}}{\vdash\!\!\!\!-\!\!\!\!-} c$, nel corso della quale vengono accedute v celle di nastro, con c configurazione finale e $v \leq \sigma$.*

Definizione 8.9 *Dato un alfabeto Σ, una macchina di Turing deterministica $\mathcal{M}_D = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) ed un intero $\sigma > 0$, una stringa $x \in \Sigma^*$ è rifiutata da $\mathcal{M}_D$ in spazio σ se esiste una computazione $q_0 \# x \underset{\mathcal{M}_D}{\vdash\!\!\!\!-\!\!\!\!-} c$, nel corso della quale vengono accedute v celle, con c configurazione massimale e non finale e $v \leq \sigma$.*

Definizione 8.10 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica) $\mathcal{M} = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) e una funzione $s : \mathbb{N} \mapsto \mathbb{N}$, diciamo che $\mathcal{M}$* accetta *$L \subseteq \Sigma^*$ in spazio $s(n)$ se, per ogni $x \in \Sigma^*$, $\mathcal{M}$ accetta x in spazio $s(|\,x\,|)$ se $x \in L$, mentre, se $x \notin L$, allora $\mathcal{M}$ non la accetta..*

Definizione 8.11 *Dato un alfabeto Σ, una macchina di Turing deterministica $\mathcal{M}_D = (\Gamma, \flat, Q, q_0, F, \delta)$ (con $\Sigma \subseteq \Gamma$) e una funzione $s : \mathbb{N} \mapsto \mathbb{N}$, diciamo che $\mathcal{M}_D$* riconosce *$L \subseteq \Sigma^*$ in spazio $s(n)$ se, per ogni $x \in \Sigma^*$, $\mathcal{M}_D$ accetta x in spazio $s(|\,x\,|)$ se $x \in L$ e rifiuta x (ancora in spazio $s(|\,x\,|)$) se $x \notin L$.*

Diciamo quindi che un linguaggio L è *accettabile in spazio deterministico* $s(n)$ se esiste una macchina di Turing deterministica $\mathcal{M}_D$ che accetta L in spazio $s(n)$, e che è *accettabile in spazio non deterministico* $s(n)$ se esiste una macchina di Turing non deterministica $\mathcal{M}_{ND}$ che accetta L in spazio $s(n)$.

Diciamo infine che un linguaggio L è *decidibile in spazio* $s(n)$ se esiste una macchina di Turing deterministica $\mathcal{M}_D$ che decide L in spazio $s(n)$.

In generale, per semplicità e per ragioni che saranno più chiare dopo avere introdotto i teoremi di compressione (nella Sezione 8.4), la complessità viene definita in termini asintotici, non facendo riferimento a costanti moltiplicative e termini di grado inferiore.

L'obiettivo fondamentale della Teoria della Complessità è essenzialmente quello di catalogare l'insieme di tutti i problemi (in particolare, dei problemi di riconoscimento) in funzione della quantità di risorse di calcolo necessarie per la relativa risoluzione.

A tale scopo, risulta di primaria rilevanza il concetto di *classe di complessità* intesa come l'insieme dei problemi (funzioni) risolubili in un determinato limite di risorse di calcolo (tipicamente spazio o tempo), nel contesto di uno specifico modello di calcolo. Come vedremo, la definizione di molte ed importanti classi di complessità fa riferimento a problemi di decisione e, quindi, tali classi sono tipicamente definite in termini di riconoscimento di linguaggi.

Ad esempio, data una funzione $f : \mathbb{N} \mapsto \mathbb{N}$, è possibile definire immediatamente le classi seguenti:

- DTIME($f(n)$) è l'insieme dei linguaggi decisi da una macchina di Turing deterministica ad 1 nastro in tempo al più $f(n)$;

- DSPACE($f(n)$) la classe dei linguaggi decisi da una macchina di Turing deterministica ad 1 nastro (più un nastro one way di input) in spazio al più $f(n)$;

- NTIME($f(n)$) è l'insieme dei linguaggi accettati da una macchina di Turing non deterministica ad 1 nastro in tempo al più $f(n)$;

- NSPACE($f(n)$) è l'insieme dei linguaggi accettati da una macchina di Turing non deterministica ad 1 nastro (più un nastro *one way* di input) in spazio al più $f(n)$.

Come abbiamo detto in precedenza, nell'esposizione dei concetti della Teoria della Complessità e nella definizione delle classi di complessità faremo generalmente uso, come modello di calcolo di riferimento, delle macchine di Turing. Tuttavia, come vedremo in seguito, tale scelta non è limitativa, in quanto molti importanti risultati sono invarianti rispetto al modello di calcolo considerato (almeno per quanto riguarda i modelli di calcolo più usuali).

Ad esempio, una importante classe di complessità è rappresentata, come vedremo, dalla classe P, l'insieme dei linguaggi decidibili da una macchina di Turing deterministica ad 1 nastro in tempo polinomiale. Se definiamo ora la classe P_k come l'insieme dei linguaggi decidibili da una macchina di Turing deterministica a k nastri in tempo polinomiale, possiamo osservare che ogni macchina di Turing a k nastri $\mathcal{M}_k$ operante in tempo polinomiale $p(n)$ può

essere simulata da una macchina di Turing ad 1 nastro $\mathcal{M}$ operante in tempo $O((p(n))^2)$ (vedi Sezione 5.4.3).

Da ciò, tenendo conto della banale osservazione che un linguaggio deciso da una macchina di Turing ad 1 nastro in tempo $t(n)$ è chiaramente decidibile da una macchina di Turing a k nastri nello stesso tempo, deriva che $\mathrm{P} = \mathrm{P}_k$: possiamo quindi identificare con P l'insieme dei problemi decidibili in tempo polinomiale da una macchina di Turing deterministica, indipendentemente dal numero di nastri utilizzati.

Una relazione dello stesso tipo rispetto a P può essere mostrata anche per quanto riguarda la decidibilità da parte di macchine a registri, in quanto, come visto nella Sezione 6.3, una macchina di Turing esegue $O((p(n))^2)$ passi per simulare una macchina a registri operante in tempo $p(n)$, mentre una macchina di Turing operante in tempo $p(n)$ può essere simulata da una macchina a registri mediante l'esecuzione di $O(p(n)\log(p(n)))$ passi.

Relazioni simili possono essere mostrate anche, in molti casi, per quanto riguarda classi di complessità spaziale.

8.4 Teoremi di compressione ed accelerazione

I cosiddetti teoremi di compressione e di accelerazione, considerati in questa sezione, mostrano che da qualunque algoritmo con una complessità (di tempo o spazio) $\phi(n)$ è possibile derivare un algoritmo di complessità $c\phi(n)$ per ogni costante $c < 1$ predefinita.

In conseguenza di tali risultati, sarà nel seguito possibile e giustificato caratterizzare la complessità di algoritmi e problemi indipendentemente da costanti moltiplicative.[2] Il primo teorema che introduciamo è relativo alla possibilità di comprimere lo spazio utilizzato nel corso di una computazione, adottando un alfabeto di nastro di dimensione sufficientemente grande.

Teorema 8.1 (Compressione lineare) *Sia $\mathcal{M}$ una macchina di Turing a 1 nastro (deterministica o non deterministica) che accetta un linguaggio L in spazio $s(n)$ e sia $c > 0$ una costante. È possibile derivare una macchina di Turing $\mathcal{M}'$ ad 1 nastro che accetta L in spazio $s'(n) \leq cs(n)$.*

Dimostrazione. Sia $k = \lceil \frac{1}{c} \rceil$. Mostriamo come ogni configurazione di lunghezza m del nastro di $\mathcal{M}$ possa essere associata ad una configurazione "compressa", di lunghezza $\lceil \frac{m}{k} \rceil$ del nastro di $\mathcal{M}'$.

Ogni cella sul nastro di $\mathcal{M}'$ corrisponderà a k celle adiacenti sul nastro di $\mathcal{M}$. Ciò viene ottenuto utilizzando per $\mathcal{M}'$ un alfabeto di nastro Γ' tale che $|\,\bar{\Gamma}'\,| = |\,\bar{\Gamma}\,|^k$. Si osservi che, dato che una posizione della testina di $\mathcal{M}'$ corrisponde a k posizioni consecutive della testina di $\mathcal{M}$, sarà necessario per

[2] Nel dimostrare i teoremi di compressione faremo riferimento a macchine di Turing ad 1 nastro. I risultati possono essere comunque facilmente estesi a macchine di Turing multinastro.

$\mathcal{M}'$ rappresentare, all'interno dello stato, l'informazione su quale di tali k posizioni sia quella su cui si trova la testina di $\mathcal{M}$.

Ciò viene reso possibile considerando gli stati di $\mathcal{M}'$ come coppie (q, j), con $q \in Q, 1 \leq j \leq k$. Ad ogni istante, $\mathcal{M}'$ si trova nello stato (q, j) se e solo se $\mathcal{M}$ si trova nello stato q e la sua testina è posta sulla j-ma tra le k celle corrispondenti alla posizione su cui si trova la testina di $\mathcal{M}'$.

La simulazione di $\mathcal{M}$ da parte di $\mathcal{M}'$ risulta quindi piuttosto semplice, almeno dal punto di vista concettuale. □

Come si può notare, il risparmio di spazio (così come, successivamente, di tempo) deriva esclusivamente da una opportuna modifica del modello di calcolo, e non da miglioramenti nella struttura concettuale dell'algoritmo, rappresentata dalla funzione di transizione.

Esercizio 8.1 Determinare la struttura di una macchina di Turing $\mathcal{M}'$ operante in spazio pari a $1/2$ di quanto utilizzato dalla macchina di Turing $\mathcal{M}$, la cui funzione di transizione è rappresentata in Tabella 5.1,

Il Teorema di Accelerazione lineare, introdotto qui di seguito, mostra poi come un opportuno incremento della dimensione dell'alfabeto di nastro consenta, sfruttando in modo adeguato la conseguente compressione di spazio, di ridurre anche il numero di passi eseguiti.

Teorema 8.2 (Accelerazione lineare) *Sia $\mathcal{M}$ una macchina di Turing (deterministica o non deterministica) che accetta una linguaggio L in tempo $t(n)$ e sia $\varepsilon > 0$ una costante. È possibile allora derivare una macchina di Turing $\mathcal{M}'$ con 2 nastri che accetta L in tempo $t'(n) \leq \lceil \varepsilon t(n) \rceil + O(n)$.*

Dimostrazione. Indichiamo con T'_0, T'_1 i due nastri di $\mathcal{M}'$ e con T il nastro di $\mathcal{M}$. Se la stringa in input w è definita sull'alfabeto Σ e se Γ è l'alfabeto di nastro di T, definiamo i due alfabeti di nastro di T'_0 e di T'_1 come $\bar{\Gamma}'_0 = \bar{\Gamma}$ e $\bar{\Gamma}'_1 = \bar{\Gamma}^m$ per un intero $m > 0$ opportuno dipendente da ε. $\mathcal{M}'$ simula $\mathcal{M}$ in 2 fasi.

1. Nella prima fase $\mathcal{M}'$ scandisce la stringa di input (che assumiamo contenuta nel nastro T'_0) e la copia, in forma compressa, sul nastro T'_1.

2. Nella seconda fase, $\mathcal{M}'$ utilizza il solo nastro T'_1 per simulare l'operato di $\mathcal{M}$ sul nastro T. Il contenuto di T'_1 viene mantenuto, durante la simulazione, allineato al contenuto di T, nel senso che esso contiene (ad istanti opportuni) la codifica compressa del contenuto di T. La simulazione opera in modo tale da accettare l'input in forma compressa su T'_1 se e solo se $\mathcal{M}$ accetta l'input originale su T.

La macchina di Turing $\mathcal{M}'$ può quindi essere descritta come la concatenazione di due macchine di Turing $\mathcal{M}_1, \mathcal{M}_2$, che eseguono le operazioni previste per le due fasi della computazione.

1. $\mathcal{M}_1$ legge iterativamente su T_0', per un valore m dipendente da ε e che verrà determinato in seguito, m caratteri della stringa di input w e li codifica sul nastro T_1' per mezzo di un unico carattere dell'alfabeto Γ_0'. Al termine della codifica, la testina di T_1' viene spostata all'inizio della stringa compressa.

   È facile verificare che il numero di passi di calcolo eseguiti durante questa fase sarà al più pari a $|\,w\,| + 1 + \lceil \frac{|w|}{m} \rceil$ (Esercizio 8.2). Indichiamo con $\chi(w)$ la stringa contenuta in T_1' al termine di questa fase.

2. $\mathcal{M}_2$ utilizza ora solo T_1' per simulare su $\chi(w)$ la computazione eseguita da $\mathcal{M}$ su w: nel far ciò simula ogni sequenza di m passi eseguiti da $\mathcal{M}$ in 9 passi, in modo tale che, per ogni intero $s \geq 0$, il nastro T contenga, dopo $9s$ passi di computazione di $\mathcal{M}_2$, la codifica $\chi(x)$ del contenuto x del nastro di $\mathcal{M}$ dopo ms passi di $\mathcal{M}$.

   Possiamo considerare lo stato di $\mathcal{M}_2$ dopo $9s$ passi come una coppia (q, j), dove $q \in Q$ è lo stato di $\mathcal{M}$ dopo ms passi e l'intero j $(1 \leq j \leq m)$ indica quale simbolo della sequenza di m simboli di $\bar{\Gamma}$ corrispondente al simbolo di $\bar{\Gamma}_1'$ attualmente scandito su T_1' è scandito da $\mathcal{M}$ sul proprio nastro T (vedi Figura 8.1). Se ad esempio assumiamo $m = 3$, allora,

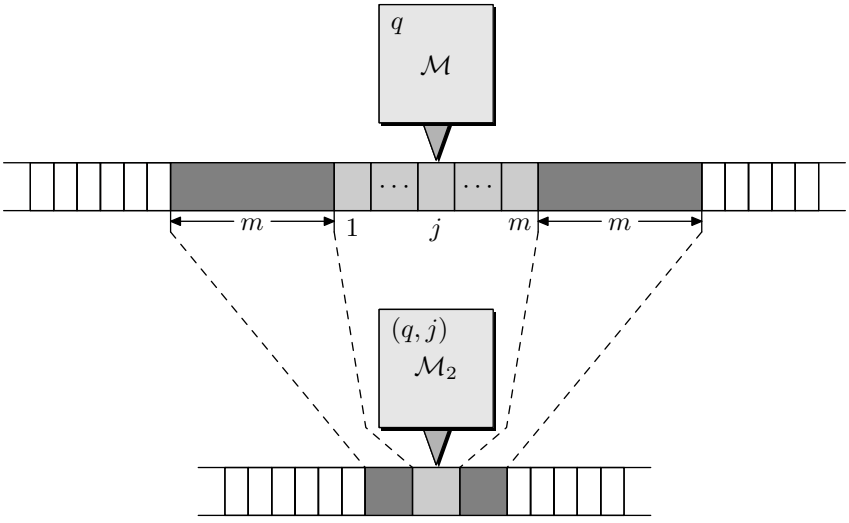


FIGURA 8.1 Simulazione di $\mathcal{M}$ da parte di $\mathcal{M}_2$.

---

se dopo $6s$ passi $\mathcal{M}$ si trova nella configurazione $abbaqbabb$, con $q \in Q$, $\mathcal{M}_2$ dopo $18s$ passi si troverà nella configurazione $(abb)(q, 2)(aba)(bb\underline{b})$,

dove ogni terna di caratteri da $\bar{\Gamma} = \{a, b, \bar{b}\}$ è in effetti rappresentata da un singolo carattere su un alfabeto (incluso il simbolo di cella vuota) di dimensione 27.

La simulazione di una sequenza di 9 passi avviene nel modo seguente:

(a) La testina su T_1' scandisce prima la cella precedente e poi quella seguente a quella su cui è posizionata (in 4 passi, muovendosi a sinistra di una cella, a destra di due e a sinistra di una): in tal modo $\mathcal{M}_2$ viene a conoscere, e a codificare nel proprio stato, il valore di una sequenza di $3m$ simboli sul nastro di $\mathcal{M}$ (che denominiamo *stringa attiva*), di cui almeno m precedenti ed m seguenti il simbolo attualmente scandito da $\mathcal{M}$. Si noti che la computazione di $\mathcal{M}$, per i prossimi m passi, potrà essere determinata, oltre che dallo stato attuale, soltanto dai simboli che compaiono nella stringa attiva: infatti, in m passi di computazione, la testina di $\mathcal{M}$ non potrà comunque esaminare simboli contenuti in celle più lontane di m posizioni da quella attualmente scandita.

(b) A questo punto, $\mathcal{M}_2$ è in grado di predire esattamente i prossimi m passi eseguiti da $\mathcal{M}$ e simularne l'effetto in un solo passo, per mezzo di una funzione di transizione rappresentata da una tabella di dimensione finita e pari a $\mid Q \mid\mid \bar{\Gamma} \mid^{3m}$, costruita a partire dalla funzione di transizione di $\mathcal{M}$.

(c) Dato che la simulazione di m passi di $\mathcal{M}$ può al massimo modificare i simboli contenuti nella cella attualmente scandita su T_1', in quella precedente ed in quella successiva, $\mathcal{M}_2$ aggiorna tali valori per mezzo di 5 passi aggiuntivi, posizionando quindi la testina di T_1' sul simbolo che codifica la m-pla che sul nastro di $\mathcal{M}$ contiene il simbolo attualmente scandito su T.

Più formalmente, definiamo l'insieme degli stati di $\mathcal{M}_2$ come

$$Q_2 = (Q \times \{1, \ldots, m\}) \cup (Q \times \{1, \ldots, m\} \times \bar{\Gamma}^{3m} \times \{1, \ldots, 8\}),$$

dove l'interpretazione di uno stato $(q, j) \in Q \times \{1, 2, \ldots, m\}$ è stata fornita sopra, mentre uno stato $(q, j, x, p) \in Q \times \{1, 2, \ldots, m\} \times \bar{\Gamma}^{3m} \times \{1, 2, \ldots, 8\}$ rappresenta lo stato raggiunto da $\mathcal{M}_2$ dopo una sequenza di $p < 9$ passi eseguiti a partire dallo stato (q, j) e con stringa attiva x.

Le 9 transizioni effettuate da $\mathcal{M}_2$ saranno allora, nell'ipotesi che y sia il simbolo scandito, x il simbolo precedente, z il simbolo successivo e (q, j) sia lo stato iniziale della sequenza:

1. $\delta_2((q, j), y) = ((q, j, \bar{b}^{3m}, 1), y, s)$
2. $\delta_2((q, j, \bar{b}^{3m}, 1), x) = ((q, j, x\bar{b}^{2m}, 2), x, d)$
3. $\delta_2((q, j, \bar{b}^{2m}, 2), y) = ((q, j, xy\bar{b}^{m}, 3), y, d)$
4. $\delta_2((q, j, \bar{b}^{m}, 3), z) = ((q, j, xyz, 4), z, s)$

Sia ora $xyz = uav$, dove $a \in \Sigma$ è il simbolo scandito da $\mathcal{M}$ (e quindi $|\,u\,| = m + j - 1$ e $|\,v\,| = 2m - j$) ed assumiamo che $uqav \overset{}{\underset{\mathcal{M}}{\longmapsto}} u'q'a'v'$ in m passi e che $u'a'v' = x'y'z'$ (con $|\,x'\,| = |\,y'\,| = |\,z'\,| = m$). Allora i passi successivi saranno:

5. $\delta_2((q, j', xyz, 4), y) = ((q, j', xyz, 5), y, s)$
6. $\delta_2((q, j', xyz, 5), x) = ((q, j', xyz, 6), x', d)$
7. $\delta_2((q, j', xyz, 6), y) = ((q, j', xyz, 7), y', d)$
8. $\delta_2((q, j', xyz, 7), z) = ((q, j', xyz, 8), z', s)$

Dove $j' = |\,u'\,| + 1$.

Infine, per quanto riguarda l'ultimo passo,

9. $$\delta_2((q, j', xyz, 9), y') = \begin{cases} ((q', j'), y', s) & j' \leq m \\ ((q', j' - m), y', i) & m < j' \leq 2m \\ ((q', j' - 2m), y', d) & 2m < j' \end{cases}$$

Da quanto detto, abbiamo che, se $\mathcal{M}$ accetta w (di lunghezza n) in t passi, $\mathcal{M}'$ accetta la stessa stringa in $9\lceil \frac{t}{m} \rceil + n + 1 + \lceil \frac{n}{m} \rceil$ passi. Mostriamo allora che, per t sufficientemente grande, basta porre $m \geq 9\frac{1+\varepsilon}{\varepsilon}$ per dimostrare l'enunciato. Infatti, se $m \geq 9\frac{1+\varepsilon}{\varepsilon}$ abbiamo, assumendo che t sia sufficientemente grande da far sì che $\varepsilon t > m$, che $\varepsilon t \geq 9\frac{t+m}{m}$ e quindi $\lceil \varepsilon t \rceil \geq 9\lceil \frac{t}{m} \rceil$, dal che il teorema deriva immediatamente. □

Come detto in precedenza, la dimostrazione si può facilmente estendere al caso di macchine di Turing multinastro (basta considerare codifiche compresse del contenuto di tutti i nastri). Si osservi però che una macchina di Turing con $k > 1$ nastri può essere simulata da una macchina di Turing avente anch'essa k nastri, in quanto non è necessario un nastro aggiuntivo per eseguire le operazioni della fase 1 della simulazione.

Essenzialmente, il Teorema 8.2, così come il Teorema 8.1, ci esprimono un concetto del tutto naturale, e cioè che, aumentando opportunamente la capacità della singola cella di memoria e quindi, in sostanza, la potenza di calcolo delle operazioni elementari definite nel modello, è possibile rendere più efficiente (in termini di operazioni elementari eseguite o di celle di memoria utilizzate) un algoritmo definito su tale modello.

Esercizio 8.2 Dato $\Sigma = \{0, 1\}$ e $m = 3$, determinare la struttura di una macchina di Turing $\mathcal{M}$ a due nastri che effettui le operazioni definite per la fase 1 nella dimostrazione del Teorema 8.2. Si richiede cioè che $\mathcal{M}$ legga un stringa $w \in \Sigma^*$ dal primo nastro T_0 e restituisca su un nastro T_1, in un totale di $|\,w\,| + 1$ passi, una stringa compressa corrispondente, definita sull'alfabeto $\Sigma' = \{a, b, c, d, e, f, g, h\}$. Si assuma la codifica che pone a in corrispondenza a 000, b a 001, ..., h a 111.

8.5 Classi di complessità

Come visto nella sezione precedente, i teoremi di compressione ed accelerazione permettono di derivare immediatamente che tutte le classi di complessità definite rispetto a funzioni asintoticamente equivalenti (a meno di costanti moltiplicative), sono in realtà equivalenti. Più precisamente, come conseguenza del Teorema 8.1 si ha che $\mathrm{DSPACE}(s(n)) = \mathrm{DSPACE}(s'(n))$ ed $\mathrm{NSPACE}(s(n)) = \mathrm{NSPACE}(s'(n))$ per ogni $s'(n) = O(s(n))$. Inoltre, come conseguenza del Teorema 8.2 abbiamo che, se $t(n) = \omega(n)$, allora $\mathrm{DTIME}(t(n)) = \mathrm{DTIME}(t'(n))$ ed $\mathrm{NTIME}(t(n)) = \mathrm{NTIME}(t'(n))$ per ogni $t'(n) = O(t(n))$.

Si noti che, in molti casi, non siamo interessati a caratterizzare la complessità di un problema in modo molto preciso, introducendo cioè, ad esempio, per quale funzione $t(n)$ esso appartenga alla classe $\mathrm{DTIME}(t(n))$; in generale, è invece sufficiente disporre di una classificazione più grossolana, stabilendo ad esempio se il problema è risolubile in tempo polinomiale o esponenziale. A tale scopo definiamo le seguenti classi di complessità:

1. la classe dei linguaggi decidibili da una macchina di Turing deterministica in tempo proporzionale ad un polinomio nella dimensione dell'input: $\mathrm{P} = \cup_{k=0}^{\infty}\mathrm{DTIME}(n^k)$;

2. la classe dei linguaggi decidibili da una macchina di Turing deterministica in spazio proporzionale ad un polinomio nella dimensione dell'input: $\mathrm{PSPACE} = \cup_{k=0}^{\infty}\mathrm{DSPACE}(n^k)$;

3. la classe dei linguaggi decidibili da una macchina di Turing deterministica in spazio proporzionale al logaritmo nella dimensione dell'input: $\mathrm{LOGSPACE} = \mathrm{DSPACE}(\log n)$;

4. la classe dei linguaggi decidibili da una macchina di Turing deterministica in spazio proporzionale ad un esponenziale nella dimensione dell'input: $\mathrm{EXPTIME} = \cup_{k=0}^{\infty}\mathrm{DTIME}(2^{n^k})$;

5. la classe dei linguaggi accettabili da una macchina di Turing non deterministica in tempo proporzionale ad un polinomio nella dimensione dell'input: $\mathrm{NP} = \cup_{k=0}^{\infty}\mathrm{NTIME}(n^k)$;

6. la classe dei linguaggi accettabili da una macchina di Turing non deterministica in spazio proporzionale ad un polinomio nella dimensione dell'input: $\mathrm{NPSPACE} = \cup_{k=0}^{\infty}\mathrm{NSPACE}(n^k)$;

7. la classe dei linguaggi accettabili da una macchina di Turing non deterministica in tempo proporzionale ad un esponenziale nella dimensione dell'input: $\mathrm{NEXPTIME} = \cup_{k=0}^{\infty}\mathrm{NTIME}(2^{n^k})$;

Si noti che tale classificazione permette di individuare molte proprietà della complessità di problemi che non varrebbero per classi più raffinate. Ad esempio, una importante varità, che corrisponde alla variante di Tesi di Church introdotta all'inizio del capitolo, un problema si trova in P o in EXPTIME indipendentemente dal modello di calcolo considerato.

8.6 Teoremi di gerarchia

In molti casi non è difficile individuare relazioni di contenimento (non stretto) tra due classi di complessità: ad esempio, vedremo nella successiva Sezione 8.7 che DTIME($t(n)$) $\subseteq$ NTIME($t(n)$) e DSPACE($t(n)$) $\subseteq$ NSPACE($t(n)$), dal che deriva, ad esempio, che P $\subseteq$ NP e PSPACE $\subseteq$ NPSPACE. Molto più complicato risulta, in generale, mostrare che tali relazioni sono di contenimento stretto: date due classi di complessità $\mathcal{C}_1, \mathcal{C}_2$ tali che $\mathcal{C}_1 \subseteq \mathcal{C}_2$, al fine di *separare* $\mathcal{C}_1$ da $\mathcal{C}_2$ (e quindi mostrare che in effetti $\mathcal{C}_1 \subset \mathcal{C}_2$) è necessario mostrare l'esistenza di un linguaggio $L \in \mathcal{C}_2 - \mathcal{C}_1$.

Mentre, nei singoli casi, è necessario individuare tale linguaggio con modalità "ad hoc", il metodo più generale per effettuare la separazione di due classi di complessità $\mathcal{C}_1$ e $\mathcal{C}_2$ consiste nell'individuare un linguaggio $L \in \mathcal{C}_2 - \mathcal{C}_1$ mediante una opportuna applicazione della tecnica di diagonalizzazione, che è stata precedentemente introdotta per individuare problemi indecidibili.

In questa sezione, mostriamo come l'applicazione della tecnica di diagonalizzazione dia luogo a due classici teoremi, detti di gerarchia, che consentono, rispettivamente, la separazione di classi di complessità temporale e spaziale. Tali teoremi si applicano a classi di complessità definite su funzioni aventi particolari proprietà di "costruttività" per mezzo di macchine di Turing.

Definizione 8.12 *Una funzione non decrescente $s : \mathbb{N} \mapsto \mathbb{N}$ è detta* space constructible *se esiste una macchina di Turing $\mathcal{M}$ che, per ogni n, utilizza esattamente $s(n)$ celle per ogni istanza di lunghezza n.*

In modo equivalente, possiamo anche dire che una funzione $s(n)$ è space constructible se esiste una macchina di Turing che per ogni n marca, ad esempio scrivendoci dentro un determinato simbolo predefinito, esattamente $s(n)$ celle di nastro, senza utilizzarne altre.

Definizione 8.13 *Una funzione non decrescente $t : \mathbb{N} \mapsto \mathbb{N}$ è detta* time constructible *se esiste una macchina di Turing $\mathcal{M}$ che, per ogni n, si ferma dopo avere eseguito esattamente $t(n)$ passi su ogni istanza di lunghezza n.*

Anche in questo caso, più informalmente, possiamo dire che una funzione $t(n)$ è time constructible se esiste una macchina di Turing che, per ogni n, "conta" esattamente fino a $t(n)$, in quanto entra in un certo stato (il suo stato finale) dopo esattamente $t(n)$ passi di computazione.

È facile rendersi conto che praticamente tutte le funzioni più familiari, e che crescono sufficientemente, sono sia space che time constructible. Ad esempio, godono di tali proprietà le funzioni n^k, k^n, $n!$, ed è inoltre noto che se f, g sono funzioni space o time constructible così sono le funzioni $f + g$, fg, f^g.

Esercizio 8.3 Dimostrare che la funzione $f(n) = n^k$ è sia space che time constructible per ogni $k \geq 1$.

Al tempo stesso, non sono space constructible funzioni $s(n) = o(\log n)$, mentre non sono time constructible funzioni $s(n) = o(n)$. È possibile inoltre mostrare con tecniche opportune, che non si ritiene però opportuno introdurre in questa trattazione, l'esistenza di ulteriori funzioni non space o time constructible.

Prima di presentare i due teoremi di gerarchia, introduciamo ora alcuni lemmi utili. Nel seguente lemma, mostriamo una proprietà che sarà utilizzata nella dimostrazione del teorema di gerarchia rispetto alla complessità di spazio. Tale proprietà ci dice che, se $s(n)$ è una funzione che cresce sufficientemente, allora ogni linguaggio accettato in spazio $s(n)$ può anche essere accettato nello stesso spazio. In altre parole, essa ci permetterà nel seguito di assumere che, sotto opportune ipotesi, le macchine di Turing considerate terminino sempre la loro esecuzione. Il lemma fa riferimento a macchine di Turing ad 1 nastro, ma è facile estenderlo al caso di macchine di Turing multinastro.

Lemma 8.3 *Data una funzione space constructible $s(n) > \log(n+1)$ per ogni n, sia $\mathcal{M}$ una macchina di Turing che usa al più $s(n)$ celle di nastro su input di lunghezza n. Esiste allora una macchina di Turing $\mathcal{M}'$ a 2 nastri che accetta lo stesso linguaggio di $\mathcal{M}$ in spazio $s(n)$, e che si ferma su ogni input.*

Dimostrazione. Il numero di configurazioni possibili per una macchina di Turing operante su input di lunghezza n (che assumiamo contenuto su un nastro di input di sola lettura) e vincolata su $s(n)$ celle di nastro di lavoro è pari a $(n+1)s(n) \mid Q \mid\mid \bar{\Gamma} \mid^{s(n)}$, dove Γ è l'alfabeto di nastro utilizzato e Q è l'insieme degli stati della macchina: infatti, la testina sul nastro di input può trovarsi su $n+1$ posizioni, la testina sul nastro di lavoro su $s(n)$ posizioni, il numero di possibili stati è $|Q|$ ed il numero di possibili contenuti diversi delle $s(n)$ celle del nastro di lavoro è $|\bar{\Gamma}|^{s(n)}$.

La macchina di Turing $\mathcal{M}'$ utilizza sul nastro aggiuntivo T' un alfabeto Γ' tale che $\mid \Gamma' \mid^{s(n)} > (n+1)s(n) \mid Q \mid\mid \bar{\Gamma} \mid^{s(n)}$ per ogni $n > 0$: la condizione $s(n) > \log(n+1)$ garantisce che la scelta $|\Gamma'| > |\bar{\Gamma}| + |Q| + 2$ rende verificata tale condizione. $\mathcal{M}'$ opera nel modo seguente: dapprima marca[3] $s(n)$ celle

[3] L'operazione di marcatura può essere effettuata considerando il nastro T' come un nastro a due tracce, ed utilizzando la seconda traccia (con alfabeto $\{1, \flat\}$) per rappresentare la presenza o meno della marca: ciò comporta (vedi Sezione 5.4.3) che l'alfabeto del nastro T', per permettere di simulare la presenza di due tracce, dovrà avere in effetti dimensione almeno pari a $2((n+1)s(n) \mid Q \mid\mid \bar{\Gamma} \mid^{s(n)})$.

sul nastro aggiuntivo T' (può far ciò in quanto $s(n)$ è space constructible), poi simula $\mathcal{M}$ sugli altri nastri. Durante la simulazione, essa conta inoltre il numero di passi eseguiti, esprimendo tale numero sulle celle marcate del nastro nastro T' in base $|\Gamma'|$.

Si osservi che il più grande numero in base $|\Gamma'|$ rappresentabile nelle $s(n)$ celle marcate sul nastro T' è pari a $|\Gamma'|^{s(n)} - 1$, che viene raggiunto quando $\mathcal{M}'$ ha simulato un numero di passi di $\mathcal{M}$ maggiore del numero di possibili configurazioni di tale macchina.

In tale eventualità, per il Pigeonhole Principle, $\mathcal{M}$ ha certamente attraversato (almeno) due volte una stessa configurazione, e quindi è entrata in un ciclo infinito.

In definitiva, se $\mathcal{M}'$, dopo avere simulato $|\Gamma'|^{s(n)}$ passi di $\mathcal{M}$, non ha raggiunto una configurazione terminante per tale macchina, allora essa si può comunque fermare in uno stato di non accettazione per la stringa in input, in quanto per quella stessa stringa $\mathcal{M}$ entra in una computazione infinita. □

Il lemma seguente, di cui forniamo il solo enunciato, verrà utilizzato nella dimostrazione del teorema di gerarchia relativo alla complessità di tempo.

Lemma 8.4 *Sia $\mathcal{M}$ una macchina di Turing a $k > 2$ nastri con alfabeti di nastro $\Sigma_1, \ldots, \Sigma_k$, che accetta un linguaggio $L(\mathcal{M})$ in tempo $t(n)$. Esiste allora una macchina di Turing $\mathcal{M}'$ a due nastri con alfabeto di nastro $\{0,1\}$, che accetta $L(\mathcal{M}') = L(\mathcal{M})$ in tempo $t(n)\log t(n)$.*

Passiamo ora ad introdurre il teorema di gerarchia spaziale, che definisce le condizioni che fanno sì che due classi di complessità spaziale differiscano.

Teorema 8.5 (Teorema di gerarchia spaziale) *Siano date due funzioni $s_1, s_2 : \mathbb{N} \mapsto \mathbb{N}$ tali che:*

- *$s_2(n) > \log(n+1)$ per ogni n*
- *$s_1(n) = o(s_2(n))$*
- *$s_2(n)$ è space constructible.*

Allora esiste un linguaggio L separatore *delle due classi di complessità* $\mathrm{DSPACE}(s_1(n))$ *e* $\mathrm{DSPACE}(s_2(n))$*, tale cioè da avere $L \in \mathrm{DSPACE}(s_2(n))$ e $L \notin \mathrm{DSPACE}(s_1(n))$.*

Dimostrazione. Consideriamo uno schema di codifica di tutte le macchine di Turing ad un nastro e con alfabeto di nastro $\{0,1\}$ mediante l'alfabeto $\Sigma = \{0,1\}$ (vedi Sezione 7.2.2). Tale codifica, dato l'indice x di una macchina di Turing $\mathcal{M}_x$, associa a tale macchina di Turing tutte le stringhe del tipo $O^i 1x$, con $i \geq 0$.

Si osservi quindi che, tutte le infinite stringhe del tipo stringa $0^i 1x$ con $x \in \{0,1\}^+$ e $i \geq 0$ descrivono, se x è la codifica di una macchina di Turing,

una stessa macchina di Turing $\mathcal{M}_x$.[4] A partire dalla sequenza di tutte le macchine di Turing ad un nastro e con $\Gamma = \{0, 1\}$, ordinate in base alla relativa codifica, possiamo costruire ora, mediante diagonalizzazione, una nuova macchina di Turing $\mathcal{M}$ a 2 nastri (con alfabeto di nastro $\{0, 1\}$), avente la seguente proprietà: per ogni stringa $y = 0^i 1x$, con $x \in \{0, 1\}^+$ e $i \geq 0$, che codifica una macchina di Turing $\mathcal{M}_x$, $\mathcal{M}$ accetta y se e solo se $\mathcal{M}_x$:

1. usa spazio al più $s_2(|\, y\,|)$;
2. rifiuta y.

Data una stringa di input y, $\mathcal{M}$ dapprima verifica, in spazio costante, che $y = 0^i 1x$ per qualche $i \geq 0$ e che x sia la codifica di una macchina di Turing: in caso contrario rifiuta y.

Altrimenti, $\mathcal{M}$ marca esattamente $s_2(|\, y\,|)$ celle sui nastri T_1, T_2, T_3 (può fare ciò in quanto la funzione $s_2(n)$ è space constructible) e copia la codifica x sul nastro T_1. Le celle marcate saranno le sole che $\mathcal{M}$ utilizzerà nel seguito sui tre nastri: quindi, se $|\, x\,| > s_2(|\, y\,|)$ la macchina di Turing $\mathcal{M}$ rifiuta y.

Successivamente, $\mathcal{M}$ simula la computazione eseguita da $\mathcal{M}_x$ su input y, utilizzando la descrizione di $\mathcal{M}_x$ sul nastro T_1 e rappresentando inoltre:

- su T_2 il contenuto del nastro di $\mathcal{M}_x$;
- su T_3 lo stato attuale di $\mathcal{M}_x$.

La macchina di Turing $\mathcal{M}$ si ferma rifiutando y non appena la simulazione la porta ad uscire dall'insieme di celle marcate. Se la simulazione al contrario termina (all'interno quindi delle zone di nastro marcate), in quanto la computazione eseguita da $\mathcal{M}_x$ su input y termina, allora $\mathcal{M}$ accetta y se e solo se $\mathcal{M}_x$ rifiuta y.

Sia $L(\mathcal{M})$ il linguaggio accettato da $\mathcal{M}$: chiaramente, per definizione, $L(\mathcal{M}) \in \text{DSPACE}(s_2(n))$.

Mostriamo ora che $L(\mathcal{M}) \notin \text{DSPACE}(s_1(n))$. Si assuma il contrario, e sia allora $\mathcal{M}_t$ una macchina di Turing ad un nastro e con alfabeto $\{0, 1\}$ che accetta $L(\mathcal{M})$ in spazio $s_1(n)$.

Si osservi che, per il Lemma 8.3, possiamo assumere che $\mathcal{M}_t$ si fermi su qualunque input (altrimenti potremmo considerare la macchina di Turing equivalente costruita nella dimostrazione del Lemma stesso). Inoltre, dato che $s_1(n) = o(s_2(n))$ per ipotesi, esiste un intero $\overline{n}$ tale che $s_1(n) < s_2(n)$ per $n > \overline{n}$.

Consideriamo ora il comportamento di $\mathcal{M}$ ed $\mathcal{M}_t$ su un input $z = 0^i 1t$ avente i sufficientemente grande da far sì che siano verificate le tre condizioni seguenti:

- $\log(|\, Q_t\,|) \leq s_2(|\, z\,|)$;

[4] Si ricorda che non tutte le stringhe in $\{0, 1\}^+$ sono delle codifiche di macchine di Turing.

- $|t| \leq s_2(|z|)$;
- $s_1(|z|) \leq s_2(|z|)$.

Dalle tre condizioni deriva che $\mathcal{M}$ ha sufficiente spazio da poter rappresentare sia la codifica della macchina di Turing $\mathcal{M}_t$ da simulare che, ad ogni istante, il suo stato attuale che, infine, il contenuto, ad ogni istante, del nastro di $\mathcal{M}_t$.

Possiamo allora osservare che, se $\mathcal{M}_t$ accetta z (in spazio al più $s_1(|z|)$, per ipotesi) allora $\mathcal{M}$ termina la sua simulazione (in quanto anche $\mathcal{M}_t$ termina) senza uscire dall'insieme delle celle marcate, per cui $\mathcal{M}$ rifiuta z. Se invece $\mathcal{M}_t$ rifiuta z (ancora, in spazio al più $s_1(|z|)$), nuovamente $\mathcal{M}$ termina la sua simulazione senza uscire dall'insieme delle celle marcate, in questo caso accettando z.

In definitiva, abbiamo che z viene accettata da $\mathcal{M}$ se e solo se viene rifiutata dalla macchina di Turing $\mathcal{M}_t$ da essa codificata, per cui $L(\mathcal{M}) \neq L(\mathcal{M}_t)$, contrariamente all'ipotesi posta. □

Il teorema seguente definisce le stesse condizioni per classi di complessità temporale.

Teorema 8.6 (Teorema di gerarchia temporale) *Siano* $t_1, t_2 : \mathbb{N} \mapsto \mathbb{N}$ *due funzioni qualunque tali che:*

- $t_2(n) > n$ *per ogni* n
- $t_2(n) = \omega(t_1(n) \log(t_1(n)))$
- $t_2(n)$ *è time constructible.*

Esiste allora un linguaggio L *avente le proprietà che* $L \in \mathrm{DTIME}(t_2(n))$ *e* $L \notin \mathrm{DTIME}(t_1(n))$.

Dimostrazione. La dimostrazione è simile alla precedente: si considera in questo caso una codifica di macchine di Turing a due nastro con alfabeto $\{0, 1\}$ simile a quella introdotta nella dimostrazione del Teorema 8.5. Si ricorda che, in questo caso, una macchina di Turing la cui codifica è rappresentata dalla stringa x è rappresentata da tutte le infinite stringhe $0^i 1x$, con $i \geq 0$.

Consideriamo una macchina di Turing $\mathcal{M}$, che opera su 5 nastri nel modo seguente: dato un input $z \in \{0, 1\}^+$, $\mathcal{M}$ verifica, in tempo $|z|$, che z abbia la forma $0^i 1x$ e che x sia la codifica di una macchina di Turing a due nastri: se così non è $\mathcal{M}$ rifiuta la stringa z. Nel caso affermativo, detta $\mathcal{M}_x$ la macchina di Turing a 2 nastri codificata, $\mathcal{M}$ copia (in tempo $O(|z|)$) la codifica x sul nastro T_1 e l'intera stringa z sul nastro T_5.

Indichiamo con $\overline{\mathcal{M}}$ la macchina di Turing associata alla definizione di funzione time constructible (vedi Definizione 8.13) che, ricevuto un input di lunghezza n, esegue esattamente $t_2(n)$ passi: tale macchina esiste sicuramente, in quanto $t_2(n)$ è time constructible.

La macchina di Turing $\mathcal{M}$ simula $\mathcal{M}_x$ utilizzando il nastro T_2 per rappresentarne lo stato attuale ed i nastri T_3, T_4 per rappresentare il contenuto dei due nastri di $\mathcal{M}_x$. La simulazione viene effettuata eseguendo una serie di passi nell'ambito di ognuno dei quali:

1. simula un passo di computazione di $\mathcal{M}_x$. A tal fine esamina in tempo $\log(|\, Q_x \,|)$, dove Q_x è l'insieme degli stati di $\mathcal{M}_x$, il contenuto del nastro T_2 per determinare lo stato attuale di $\mathcal{M}_x$. Mediante una scansione in tempo $O(|\, x \,|)$ del contenuto del nastro T_1 viene quindi determinata l'applicazione della funzione di transizione di $\mathcal{M}_x$, e lo stato attuale viene aggiornato in tempo $\log(|\, Q_x \,|)$;

2. simula, in tempo costante, un passo di computazione di $\overline{\mathcal{M}}$.

Dato che $|\, x \,|$ e $|\, Q_x \,|$ sono costanti rispetto a $|\, z \,|$, ne consegue che la simulazione di ogni passo di $\mathcal{M}_x$ viene eseguita in tempo costante.

La macchina di Turing $\mathcal{M}$ si ferma dopo avere simulato esattamente $t_2(|\, z \,|)$ passi di $\mathcal{M}_x$, e quindi dopo avere eseguito $O(|\, z \,|) + \Theta(t_2(|\, z \,|)) = \Theta(t_2(|\, z \,|))$ passi, accettando la stringa in input se e solo se la stringa non viene accettata da $\mathcal{M}_x$ nei $t_2(|\, x \,|)$ passi simulati.

Chiaramente $L(\mathcal{M}) \in \mathrm{DTIME}(t_2(n))$. Come sopra, mostriamo che $L(\mathcal{M}) \notin \mathrm{DTIME}(t_1(n))$ per contraddizione, assumendo che esista una qualche macchina di Turing a k nastri che accetta $L(\mathcal{M})$ in tempo $t_1(n)$. Per il Lemma 8.4, esiste allora una macchina di Turing $\mathcal{M}_t$ a 2 nastri che accetta $L(\mathcal{M})$ in tempo $t_1(n) \log t_1(n)$.

Consideriamo, come nella dimostrazione del teorema precedente, un input $z = 0^i 1 t$ avente i sufficientemente grande da far sì che sia verificata la condizione $t_1(|\, z \,|) \log t_1(|\, z \,|) \leq t_2(|\, z \,|)$. Abbiamo allora che se $\mathcal{M}_t$ accetta z (in tempo $t_1(|\, z \,|) \log t_1(|\, z \,|)$), allora $\mathcal{M}$, simulando $t_1(|\, z \,|) \log t_1(|\, z \,|) \leq t_2(|\, z \,|)$ passi di $\mathcal{M}_t$ (in tempo $\Theta(t_2(|\, z \,|))$), rifiuta z. Le stesse considerazioni possono essere effettuate nel caso in cui $\mathcal{M}_t$ non accetti z (in tempo $t_1(|\, t \,|) \log t_1(|\, z \,|)$), mostrando quindi che $L(\mathcal{M}) \neq L(\mathcal{M}_t)$. □

Il nome utilizzato per i due teoremi precedenti deriva dal fatto che tali teoremi ci permettono di stabilire gerarchie di classi di complessità (rispettivamente spaziali e temporali) propriamente contenute le une nelle altre.

Dal Teorema 8.5 e dal Teorema 8.6 derivano infatti, ad esempio, i seguenti corollari.

Corollario 8.7 LOGSPACE $\subset$ PSPACE.

Corollario 8.8 P $\subset$ EXPTIME.

Come visto, i teoremi di gerarchia ci dicono essenzialmente che, se consideriamo le classi di complessità definite da funzioni "sufficientemente diverse"

in termini di rapidità di crescita asintotica, allora tali classi sono diverse, nel senso che l'insieme dei linguaggi associati alle due classi è diverso. Al contrario, nel caso di funzioni generali (non space o time constructible) è possibile dimostrare che esistono casi patologici, in cui esistono coppie di funzioni sostanzialmente diverse a cui corrispondono classi di complessità identiche. Tale fenomeno è illustrato dal seguente teorema, che forniamo senza dimostrazione.

Teorema 8.9 (Gap theorem) *Sia data una qualunque funzione calcolabile $t(n) > n$, esiste allora una funzione calcolabile $T(n)$ tale che* $\mathrm{DTIME}(t(n)) = \mathrm{DTIME}(T(t(n)))$.

Uno stesso risultato è valido nel caso della complessità spaziale.

Ad esempio, se si considera $T(n) = 2^n$, il teorema precedente ci dice che esiste una funzione $t(n)$, evidentemente non costructible, tale che non esiste nessun problema con complessità temporale compresa tra $\mathrm{DTIME}(t(n))$ e $\mathrm{DTIME}(2^{t(n)})$, vale a dire che esiste un intervallo (*gap*) di complessità esponenziale all'interno del quale non cade nessun problema.

8.7 Relazioni tra misure diverse

In questa sezione introduciamo alcuni risultati generali che correlano classi di complessità definite rispetto a misure diverse.

Teorema 8.10 *Per ogni funzione $f : \mathbb{N} \mapsto \mathbb{N}$ valgono le due proprietà:*

- $\mathrm{DTIME}(f(n)) \subseteq \mathrm{NTIME}(f(n))$;
- $\mathrm{DSPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$.

Dimostrazione. Deriva semplicemente dall'osservazione che ogni macchina di Turing deterministica è un caso particolare di macchina di Turing non deterministica. □

Dal teorema precedente derivano immediatamente i seguenti corollari.

Corollario 8.11 $\mathrm{P} \subseteq \mathrm{NP}$.

Corollario 8.12 $\mathrm{PSPACE} \subseteq \mathrm{NPSPACE}$.

Teorema 8.13 *Per ogni funzione $f : \mathbb{N} \mapsto \mathbb{N}$ valgono le due proprietà:*

- $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$;
- $\mathrm{NTIME}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$.

Dimostrazione. È sufficiente notare che in t passi di computazione si possono utilizzare al più t celle di nastro distinte. □

I seguenti corollari derivano in modo immediato.

Corollario 8.14 P $\subseteq$ PSPACE.

Corollario 8.15 NP $\subseteq$ NPSPACE.

Teorema 8.16 *Per ogni coppia di funzioni* $f, g : \mathbb{N} \mapsto \mathbb{N}$*, per le quali si ha che* $g(n) = O(f(n))$*, valgono le due proprietà:*

- DSPACE($g(n)$)$\subseteq$DSPACE($f(n)$)*;*
- NSPACE($g(n)$) $\subseteq$ NSPACE($f(n)$).

Dimostrazione. Ricordiamo che, in conseguenza del fatto che $g(n) = O(f(n))$, esiste una costante c tale che $g(n) \leq cf(n)$ per n sufficientemente grande. Quindi DSPACE($g(n)$)$\subseteq$DSPACE($cf(n)$) e NSPACE($g(n)$) $\subseteq$ NSPACE($cf(n)$). L'enunciato deriva dall'osservazione che, per il Teorema 8.1, si ha che DSPACE($cf(n)$)= DSPACE($f(n)$) e che NSPACE($cf(n)$) = NSPACE($f(n)$). □

Teorema 8.17 *Per ogni coppia di funzioni* $f, g : \mathbb{N} \mapsto \mathbb{N}$*, tali che* $g(n) = O(f(n))$ *e* $f(n) = \omega(n)$*,* DTIME($g(n)$) $\subseteq$ DTIME($f(n)$) *e* NTIME($g(n)$) $\subseteq$ NTIME($f(n)$).

Dimostrazione. La dimostrazione è lasciata per esercizio. □

Esercizio 8.4 Dimostrare il Teorema 8.17.

Teorema 8.18 *Per ogni funzione space constructible* $f : \mathbb{N} \mapsto \mathbb{N}$*, tale che* $f(n) = \Omega(\log n)$*,* NSPACE($f(n)$) $\subseteq$ DTIME($2^{f(n)}$).

Dimostrazione. Data una macchina di Turing non deterministica $\mathcal{M}$ che usa spazio $f(n)$, deriviamo una macchina di Turing deterministica $\mathcal{M}'$ che accetta lo stesso linguaggio di $\mathcal{M}$ in tempo $O(2^{f(n)})$.

Assumiamo, senza perdere in generalità, che ogni volta che $\mathcal{M}$ accetta un input, essa cancelli tutti i simboli dal nastro e posizioni la testina sulla prima cella del nastro stesso: in tal modo, la macchina $\mathcal{M}$ così modificata accetta lo stesso linguaggio ed ha una sola configurazione di accettazione c_a. Assumiamo inoltre che $\mathcal{M}$ si fermi su ogni input: se così non fosse, sarebbe possibile applicare il Lemma 8.3 per ottenere una macchina di Turing con tale proprietà ed equivalente ad $\mathcal{M}$.

Dato un input x di lunghezza n, $\mathcal{M}'$ genera l'insieme di tutte le configurazioni di $\mathcal{M}$ che occupano al più $f(n)$ celle di nastro. Si osservi che tale insieme di $O(2^{f(n)})$ elementi diversi può essere generato utilizzando "etichette" di lunghezza $f(n)$, che $\mathcal{M}'$ può generare in quanto f è space constructible.

Consideriamo ora il grafo orientato $G = (C, A)$ definito nel modo seguente:

- C è l'insieme delle possibili configurazioni di $\mathcal{M}$ che occupano al più $f(n)$ celle di nastro in computazioni eseguite su input x;

- l'unica configurazione accettante c_a è la radice di G;

- per ogni coppia di configurazioni $c', c'' \in C$, esiste un arco $\langle c', c'' \rangle \in A$ se e solo se $\mathcal{M}$ può passare da c'' a c' su un cammino di computazione a partire da input x.

Il grafo G ha $O(2^{f(n)})$ nodi e $O(2^{f(n)})$ archi (in quanto da ogni configurazione si possono eseguire un numero costante di transizioni diverse).

A questo punto, $\mathcal{M}'$ può verificare se $\mathcal{M}$ raggiunge la configurazione accettante c_a a partire dalla configurazione iniziale c_i con input x attraversando il grafo G per mezzo ad esempio di una visita in profondità, che richiede tempo lineare nella dimensione del grafo. Si osservi che la visita può essere effettuata anche senza avere a disposizione una rappresentazione esplicita del grafo, verificando la presenza di un arco (u, v) mediante l'applicazione della funzione di transizione di $\mathcal{M}$ tra le configurazioni corrispondenti ad u e v.

Da ciò deriva che in tempo $O(2^{f(n)})$ $\mathcal{M}'$ verifica se esiste una computazione accettante di $\mathcal{M}$ su input x. □

Teorema 8.19 *Sia $f : \mathbb{N} \mapsto \mathbb{N}$ una funzione time constructible, vale allora la proprietà* $\text{NTIME}(f(n)) \subseteq \text{DSPACE}(f(n))$.

Dimostrazione. Data una macchina di Turing non deterministica $\mathcal{M}$ che usa tempo $f(n)$, deriviamo una macchina di Turing deterministica $\mathcal{M}'$ che accetta lo stesso linguaggio di $\mathcal{M}$ in spazio $O(f(n))$ nel modo seguente.

Per ogni input x di lunghezza n, utilizzando il fatto che f è time constructible, $\mathcal{M}'$ simula $t(n)$ passi di tutti i possibili cammini di computazione a partire dalla configurazione iniziale su input x. L'input viene accettato se uno dei cammini porta ad una configurazione di accettazione per $\mathcal{M}$.

Per ogni cammino di computazione di $\mathcal{M}$, di lunghezza $f(n)$, $\mathcal{M}'$ utilizzerà necessariamente spazio $O(f(n))$. L'enunciato segue osservando che lo stesso spazio può essere utilizzato da $\mathcal{M}'$ per simulare tutti i cammini di computazione di $\mathcal{M}$. □

Dal teorema precedente deriva il seguente corollario.

Corollario 8.20 $\text{NP} \subseteq \text{PSPACE}$.

Teorema 8.21 *Per ogni funzione space constructible $f : \mathbb{N} \mapsto \mathbb{N}$, tale che $f(n) = \Omega(\log n)$,* $\text{DSPACE}(f(n)) \subseteq \text{DTIME}(2^{f(n)})$.

Dimostrazione. La dimostrazione è lasciata per esercizio. □

Esercizio 8.5 Dimostrare il Teorema 8.21.

Dal teorema precedente derivano i seguenti corollari.

Corollario 8.22 LOGSPACE $\subseteq$ P.

Corollario 8.23 PSPACE $\subseteq$ EXPTIME.

Teorema 8.24 *Sia* $f : \mathbb{N} \mapsto \mathbb{N}$ *una funzione time constructible, allora vale la proprietà* $\text{NTIME}(f(n)) \subseteq \text{DTIME}(2^{f(n)})$.

Dimostrazione. Il risultato deriva immediatamente come corollario del Teorema 5.2, osservando che $d^k = \Theta(2^k)$ per ogni costante d. Dall'esame della dimostrazione del Teorema 5.2 si può osservare anche che la funzione $f(n)$ deve essere time constructible, in quanto è necessario "contare" fino ad $f(n)$ durante la fase di generazione delle stringhe rappresentanti cammini di computazione.[5] □

Dal teorema precedente deriva il corollario seguente.

Corollario 8.25 NP $\subseteq$ EXPTIME.

Il Teorema 8.24 mostra come, se si considera il tempo di computazione, il passaggio da modello non deterministico a modello deterministico comporta un aumento al più esponenziale del costo di esecuzione di un algoritmo. Come già osservato quando è stato dimostrato il Teorema 5.2, nessuna simulazione in tempo sub esponenziale di una macchina di Turing non deterministica mediante macchine di Turing deterministiche è stata individuata fino ad oggi, e si congettura che in effetti il passaggio da modello non deterministico a modello deterministico richieda necessariamente un aumento esponenziale del tempo di computazione.

Il teorema seguente mostra invece come, a differenza che per il tempo di computazione, nel caso dello spazio l'ipotesi di utilizzare un modello deterministico comporta un maggior costo limitato (dell'ordine del quadrato) rispetto al caso non deterministico.

Teorema 8.26 (Teorema di Savitch) *Sia data* $f : \mathbb{N} \mapsto \mathbb{N}$ *una funzione space constructible tale che per ogni* n $f(n) \geq \log n$, *allora* $\text{NSPACE}(f(n)) \subseteq \text{DSPACE}((f(n))^2)$.

Dimostrazione. Per dimostrare questo Teorema, mostriamo prima un risultato intermedio relativo al problema Raggiungibilità, definito nel modo seguente.

[5] Una dimostrazione alternativa di questo teorema si può ottenere immediatamente dai Teoremi 8.13 e 8.18.

RAGGIUNGIBILITÀ
ISTANZA: Un grafo orientato $G = (V, E)$, una coppia di nodi $s, t \in V$.
PREDICATO: Esiste un cammino orientato da s a t in G?

In particolare, mostreremo che RAGGIUNGIBILITÀ può essere risolto in spazio $O(\log^2 n)$.

Si osservi che la soluzione più semplice al problema, basata su una visita in profondità di G a partire da s, richiede di memorizzare ogni nodo attraversato durante la ricerca, risultando in spazio $\Theta(n)$.

Una soluzione più efficiente, dal punto di vista dello spazio utilizzato, deriva dalla seguente osservazione. Se noi consideriamo il predicato $C(u, v, 2^k)$, con $u, v \in V$ e $k \leq \lceil \log_2 n \rceil$ intero, definito come vero se e solo se esiste un cammino da u a v in G di lunghezza al più 2^k, allora:

- la soluzione di RAGGIUNGIBILITÀ è data da $C(s, t, 2^{\lceil \log_2 n \rceil})$;
- $C(u, v, 2^0)$ è facilmente valutabile, in quanto è vero se e solo se esiste un arco $\langle u, v \rangle$ in G;
- Dato che ogni cammino di lunghezza 2^k può essere decomposto in due cammini concatenati di lunghezza 2^{k-1}, $C(u, v, 2^k)$ (per $k > 0$) è vero se e solo se esiste un nodo $w \in V$ tale che siano veri $C(u, w, 2^{k-1})$ e $C(w, v, 2^{k-1})$.

Ciò fornisce il seguente semplice algoritmo ricorsivo per effettuare la valutazione di $C(u, v, 2^k)$. Dal punto di vista dello spazio utilizzato dall'Algoritmo 8.1

```
input u, v: nodi, k intero;
output boolean;
begin
  if k = 0
    then if ⟨u, v⟩ ∈ E then return true
    else
      begin
        for each w ∈ V − {u, v} do
          if C(u, w, 2^{k−1}) and C(w, v, 2^{k−1}) then return true;
        return false
      end
end.
```

Algoritmo 8.1: Algoritmo ricorsivo per valutare la funzione $C(u, v, 2^k)$

per valutare $C(s, t, n)$ (e quindi per risolvere RAGGIUNGIBILITÀ), possiamo osservare che, in ogni istante, esso deve tenere conto di $k \leq \lceil \log_2 n \rceil$ coppie

di nodi, $\langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle$ con $u_1 = s$, $v_1 = t$, e $u_i = u_{i-1}$ o $v_i = v_{i-1}$ per $i = 2, \ldots, k$, coppie corrispondenti ai cammini che esso sta ricorsivamente considerando. Quindi, l'Algoritmo 8.1 deve rappresentare ad ogni istante $O(\log n)$ nodi, ognuno dei quali necessita a sua volta, per la rappresentazione della relativa etichetta di $O(\log n)$ spazio, per una complessità di spazio totale $O(\log^2 n)$.

L'enunciato del Teorema segue quindi osservando che la costruzione nella dimostrazione del Teorema 8.18 può essere effettuata anche in questo caso. Consideriamo infatti il grafo $G = (C, A)$, definito in tale dimostrazione, delle configurazioni di lunghezza al più $f(n)$ di una macchina di Turing non deterministica $\mathcal{M}$ che accetta un linguaggio $\mathcal{L}(\mathcal{M}$ in spazio $O(f(n))$. Tale grafo avrà $O(2^{f(n)})$ nodi e $O(2^{f(n)})$ archi. Nuovamente, una macchina di Turing deterministica $\mathcal{M}'$ può determinare se $\mathcal{M}$ accetta una stringa x determinando se, a partire dalla configurazione iniziale, è possibile raggiungere la configurazione finale. Da quanto mostrato sopra, $\mathcal{M}'$ può effettuare tale compito utilizzando spazio $O(\log(2^{f(n)})) = O((f(n))^2)$. □

Dal Teorema di Savitch e dal Teorema 8.10 deriva il corollario seguente.

Corollario 8.27 NPSPACE = PSPACE.

Si osservi come il corollario precedente mostri una relazione di equivalenza tra modello deterministico e modello non deterministico quando si consideri computazioni a spazio limitato polinomialmente. Nel caso della complessità temporale, un risultato di tale tipo non è disponibile, ed anzi la caratterizzazione precisa della relazione esistente tra le due classi corrispondenti a quelle considerate nel Corollario 8.27, vale a dire P ed NP, è un importante problema aperto.

A conclusione di questa sezione, in Figura 8.2 sono illustrate graficamente le relazioni di contenimento (stretto o meno) individuate tra le classi di complessità introdotte nella Sezione 8.5.

8.8 Riducibilità tra problemi diversi

In molti casi interessanti la complessità di soluzione di un problema non è nota in modo esatto né sufficientemente approssimato, in quanto il divario tra upper e lower bound di complessità del problema stesso è estremamente ampio. Una tipica situazione di questo genere si presenta in molti problemi di decisione di grande interesse pratico (come ad esempio SODDISFACIBILITÀ, che sarà introdotto nel seguito di questa sezione), per i quali il miglior lower bound noto è $\Omega(n \log n)$ o $\Omega(n^2)$, mentre il più efficiente algoritmo noto per la loro soluzione ha complessità $O(c^n)$ per qualche $c > 1$. In tali casi risulta di interesse caratterizzare la complessità di un problema relativamente ad altri

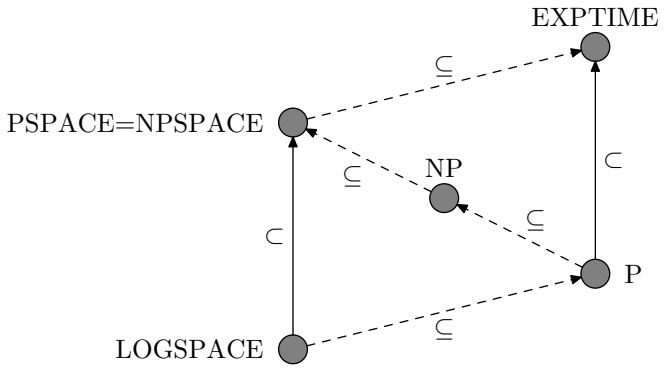


FIGURA 8.2 Relazioni tra le classi di complessità introdotte.

problemi e, in questo ambito, i concetti di *riducibilità* e di *riduzione* rappresentano uno strumento di fondamentale importanza per individuare relazioni tra le complessità di problemi diversi.

In generale, una riduzione da un problema $\mathcal{P}_1$ ad un altro problema $\mathcal{P}_2$ fornisce un metodo per risolvere $\mathcal{P}_1$ assumendo di essere in grado di risolvere $\mathcal{P}_2$ e facendo uso di un algoritmo per $\mathcal{P}_2$. Se è possibile far ciò allora possiamo dedurre che risolvere $\mathcal{P}_2$ è almeno tanto difficile quanto risolvere $\mathcal{P}_1$ (dato che sappiamo risolvere quest'ultimo nell'ipotesi di saper risolvere l'altro), assunto che applicare la riduzione comporti una computazione "sufficientemente semplice".

Possiamo definire tipi diversi di riducibilità, sotto diverse ipotesi su come la soluzione di $\mathcal{P}_2$ possa essere utilizzata per risolvere $\mathcal{P}_1$.

Definizione 8.14 *Un problema di decisione $\mathcal{P}_1$ è* Karp–riducibile *(o riducibile molti-a-uno) a un problema di decisione $\mathcal{P}_2$ ($\mathcal{P}_1 \leq_m \mathcal{P}_2$) se e solo se esiste un algoritmo $\mathcal{R}$ che trasforma ogni istanza $x \in I_{\mathcal{P}_1}$ di $\mathcal{P}_1$ in una istanza $y \in I_{\mathcal{P}_2}$ di $\mathcal{P}_2$ in modo tale che $x \in Y_{\mathcal{P}_1}$ se e solo se $y \in Y_{\mathcal{P}_2}$. $\mathcal{R}$ è detta* Karp–riduzione *da $\mathcal{P}_1$ a $\mathcal{P}_2$.*

Si noti che se esiste una riduzione $\mathcal{R}$ da $\mathcal{P}_1$ a $\mathcal{P}_2$ e se si conosce un algoritmo $\mathcal{A}_2$ per $\mathcal{P}_2$, allora si può ottenere un algoritmo $\mathcal{A}_1$ per $\mathcal{P}_1$ nel modo seguente:

1. data una istanza $x \in I_{\mathcal{P}_1}$, si applica $\mathcal{R}$ a x e si ottiene $y \in I_{\mathcal{P}_2}$;
2. si applica $\mathcal{A}_2$ a y: se $\mathcal{A}_2$ restituisce VERO allora si restituisce VERO, altrimenti ($\mathcal{A}_2$ restituisce FALSO) si restituisce FALSO.

Inoltre, dato che $t_{\mathcal{A}_1}(|\ x\ |) = t_R(|\ x\ |) + t_{\mathcal{A}_2}(|\ y\ |)$, se abbiamo anche che $t_R(|x|) = O(t_{\mathcal{A}_2}(|y|))$ allora ne consegue $t_{\mathcal{A}_1}(|x|) = \Theta(t_{\mathcal{A}_2}(|y|))$. Inoltre, per

quanto riguarda la complessità di soluzione dei problemi, la complessità di $\mathcal{P}_1$ risulta essere un lower bound della complessità di $\mathcal{P}_2$ (e, conseguentemente, la complessità di $\mathcal{P}_2$ è un upper bound di quella di $\mathcal{P}_1$).

Un caso particolare di Karp-riducibilità, particolarmente rilevante in Teoria della Complessità, è rappresentato dalla *Karp-riducibilità polinomiale*. Diciamo che $\mathcal{P}_1$ è polinomialmente Karp–riducibile a $\mathcal{P}_2$ ($\mathcal{P}_1 \leq_m^p \mathcal{P}_2$) se e solo se $\mathcal{P}_1$ è Karp–riducibile a $\mathcal{P}_2$ e la riduzione relativa $\mathcal{R}$ è un algoritmo calcolabile in tempo polinomiale.

È importante osservare che, se $\mathcal{P}_1 \leq_m^p \mathcal{P}_2$, allora saper risolvere efficientemente (in tempo polinomiale) $\mathcal{P}_2$ comporta che anche $\mathcal{P}_1$ può essere risolto efficientemente. Cioè, $\mathcal{P}_2 \in \mathrm{P} \Rightarrow \mathcal{P}_1 \in \mathrm{P}$ e, conseguentemente, $\mathcal{P}_1 \notin \mathrm{P} \Rightarrow \mathcal{P}_2 \notin \mathrm{P}$.

Esempio 8.8 Sia $X = \{x_1, x_2, \ldots, x_n\}$ un insieme di variabili booleane e sia $\mathcal{T}(X) = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$ il corrispondente insieme di *termini*. Un *assegnamento di verità* su X, $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$ soddisfa, per ogni variabile $x_i \in X$, il termine x_i se $f(x_i) = \text{VERO}$ e il termine $\overline{x}_i$ se $f(x_i) = \text{FALSO}$.

Chiamiamo *clausola* (disgiuntiva) un insieme di termini $c = \{t_1, t_2, \ldots, t_k\} \subseteq T(X)$. Una clausola è soddisfatta da f se e solo se esiste (almeno) un termine $t_i \in c$ soddisfatto da f.

Una formula in CNF (forma normale congiuntiva) è definita come una collezione di clausole $\mathcal{F} = \{c_1, c_2, \ldots, c_m\}$. Una formula in CNF è soddisfatta da f se e solo se ogni clausola c_i, $i = 1, \ldots, m$, è soddisfatta da f.

Il problema SODDISFACIBILITÀ è definito nel modo seguente.

SODDISFACIBILITÀ
ISTANZA: Una formula CNF $\mathcal{F}$ su un insieme X di variabili booleane
PREDICATO: Esiste una assegnazione $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$ che soddisfa $\mathcal{F}$?

Consideriamo inoltre il problema di decisione PROGRAMMAZIONE LINEARE $0-1$ definito nel modo seguente.

PROGRAMMAZIONE LINEARE $0-1$
ISTANZA: Insieme di variabili $Z = \{z_1, \ldots, z_n\}$ con dominio {0,1}, insieme $\mathcal{I}$ di disequazioni lineari su Z
PREDICATO: Esiste una soluzione a $\mathcal{I}$, cioè una assegnazione di valori alle variabili in Z che verifica tutte le disequazioni?

Non è difficile rendersi conto che:

$$\text{SODDISFACIBILITÀ} \leq_m^p \text{PROGRAMMAZIONE LINEARE } 0-1.$$

Infatti, ogni instanza $x_{SAT} = \{V, \mathcal{F}\}$ di SODDISFACIBILITÀ può essere ridotta ad una istanza corrispondente $x_{LP} = \{Z, \mathcal{I}\}$ di PROGRAMMAZIONE LINEARE $0-1$ nel modo seguente.

Sia $\{t_{j_1}, t_{j_2}, \ldots, t_{j_{n_j}}\}$ la j-ma clausola di $\mathcal{F}$: da tale clausola possiamo derivare una disequazione corrispondente $\zeta_{j_1} + \zeta_{j_2} + \cdots + \zeta_{j_{n_j}} > 0$ per x_{LP}, dove $\zeta_{j_k} = z_{j_k}$ se

$t_{j_k} = v_{j_k}$ (per qualche $v_{j_k} \in V$) e $\zeta_{j_k} = (1 - z_{j_k})$ altrimenti, cioè se $t_{j_k} = \overline{v}_{j_k}$ (per qualche $v_{j_k} \in V$).

È immediato verificare che ogni assegnazione di verità $f : V \mapsto \{\text{VERO}, \text{FALSO}\}$ soddisfa $\mathcal{F}$ se e solo se tutte le disequazioni in $\mathcal{I}$ sono verificate da una assegnazione di valori $f' : Z \mapsto \{0, 1\}$ tale che $f'(z_i) = 1$ se e solo se $f(v_i) = \text{VERO}$. Si noti che la riduzione è chiaramente calcolabile in tempo polinomiale.

Numerosi altri tipi di riducibilità tra problemi, non necessariamente di decisione, possono essere introdotti: tra questi, risulta di particolare rilevanza la *Turing–riducibilità*, che essenzialmente modella la possibilità, nello scrivere un programma per la soluzione di un problema, di usare un sottoprogramma per risolvere problemi diversi, sottoprogramma che, al momento dell'esecuzione, può essere chiamato tutte le volte che sono necessarie. Tale situazione è modellata in Teoria della Complessità mediante il concetto di *oracolo*.

Definizione 8.15 *Sia $\mathcal{P}$ il problema di calcolare una funzione $f : \mathcal{I}_\mathcal{P} \mapsto \mathcal{S}_\mathcal{P}$ (eventualmente a più valori). Un oracolo per $\mathcal{P}$ è un dispositivo astratto che per ogni $x \in \mathcal{I}_\mathcal{P}$ restituisce una soluzione $f(x) \in \mathcal{S}_\mathcal{P}$. Si assume che l'oracolo calcoli e restituisca la soluzione in un solo passo di computazione.*

Definizione 8.16 *Sia $\mathcal{P}_1$ il problema di calcolare una funzione $g : \mathcal{I}_{\mathcal{P}_1} \mapsto \mathcal{S}_{\mathcal{P}_1}$ (eventualmente a più valori). $\mathcal{P}_1$ si dice* Turing–riducibile *ad un problema $\mathcal{P}_2$ ($\mathcal{P}_1 \leq_T \mathcal{P}_2$) se e solo se esiste un algoritmo $\mathcal{R}$ che risolve $\mathcal{P}_1$ interrogando un oracolo per $\mathcal{P}_2$. In tal caso, $\mathcal{R}$ è detta* Turing–riduzione *da $\mathcal{P}_1$ a $\mathcal{P}_2$.*

Come per la Karp–riducibilità, possiamo definire una Turing–riducibilità polinomiale $\leq_T^p$, assumendo che $\mathcal{P}_1 \leq_T^p \mathcal{P}_2$ se e solo se $\mathcal{P}_1 \leq_T \mathcal{P}_2$ e la Turing–riduzione $\mathcal{R}$ è calcolabile in tempo (deterministico) polinomiale.

Si noti che la Karp–riducibilità non è altro che un caso particolare di Turing–riducibilità, corrispondente al caso in cui:

1. $\mathcal{P}_1$ e $\mathcal{P}_2$ sono entrambi problemi di decisione;
2. l'oracolo per $\mathcal{P}_2$ viene interrogato una sola volta;
3. $\mathcal{R}$ restituisce lo stesso valore fornito dall'oracolo.

In generale, la Karp–riducibilità è più debole della Turing–riducibilità. Ad esempio, per ogni problema di decisione $\mathcal{P}$, il problema complemento co-$\mathcal{P}$ è definito in modo tale che $I_\mathcal{P} = I_{\text{co-}\mathcal{P}}$ e, per ogni $x \in I_\mathcal{P}$, $x \in Y_{\text{co-}\mathcal{P}}$ se e solo se $x \in N_\mathcal{P}$. Cioè, co-$\mathcal{P}$ accetta tutte le istanze non accettate in $\mathcal{P}$. Non è difficile vedere che $\mathcal{P}$ è Turing–riducibile a co-$\mathcal{P}$ (e viceversa), mentre ciò non è vero per quanto riguarda la Karp–riducibilità.

Esercizio 8.6 Mostrare che le relazioni di Karp riducibilità, Karp riducibilità polinomiale, Turing riducibiltà e Turing riducibilità polinomiale godono tutte della proprietà transitiva.

Per finire, introduciamo alcune definizioni valide per ogni classe di complessità e per ogni riducibilità.

Definizione 8.17 *Per ogni classe di complessità $\mathcal{C}$, un problema di decisione $\mathcal{P}$ viene detto* difficile *in $\mathcal{C}$ ($\mathcal{C}$–hard) rispetto ad una riducibilità $\leq_r$ se e solo se per ogni altro problema di decisione $\mathcal{P}_1 \in \mathcal{C}$ si ha che $\mathcal{P}_1 \leq_r \mathcal{P}$.*

Definizione 8.18 *Per ogni classe di complessità $\mathcal{C}$, un problema di decisione $\mathcal{P}$ viene detto* completo *in $\mathcal{C}$ ($\mathcal{C}$–completo) rispetto ad una riducibilità $\leq_r$ se $\mathcal{P}$ è $\mathcal{C}$–hard (rispetto a $\leq_r$) e $\mathcal{P} \in \mathcal{C}$.*

Come conseguenza immediata delle definizioni precedenti, abbiamo che, dati due problemi $\mathcal{P}_1$, $\mathcal{P}_2$, se $\mathcal{P}_1$ è $\mathcal{C}$–hard (rispetto a $\leq_r$) e $\mathcal{P}_1 \leq_r \mathcal{P}_2$, allora anche $\mathcal{P}_2$ è $\mathcal{C}$–hard. Inoltre, se $\mathcal{P}_2 \in \mathcal{C}$, $\mathcal{P}_2$ è $\mathcal{C}$–completo.

Non è difficile verificare che, per ogni coppia di classi di complessità $\mathcal{C}_1$, $\mathcal{C}_2$ tali che $\mathcal{C}_1 \subset \mathcal{C}_2$ e $\mathcal{C}_1$ sia chiusa rispetto ad una riducibilità $\leq_r$, ogni problema $\mathcal{P}$ $\mathcal{C}_2$–completo appartiene a $\mathcal{C}_2 - \mathcal{C}_1$. Infatti, per ogni problema $\mathcal{P}_1 \in \mathcal{C}_2 - \mathcal{C}_1$, si ha che $\mathcal{P}_1 \leq_r \mathcal{P}$ per la completezza di $\mathcal{P}$. Allora, per la chiusura di $\mathcal{C}_1$, $\mathcal{P} \in \mathcal{C}_1$ comporterebbe che $\mathcal{P}_1 \in \mathcal{C}_1$, e si otterrebbe una contraddizione.

Questa osservazione suggerisce che, date due classi di complessità $\mathcal{C}_1$ e $\mathcal{C}_2$, se $\mathcal{C}_1 \subseteq \mathcal{C}_2$, allora il miglior approccio per determinare se $\mathcal{C}_1 \subset \mathcal{C}_2$ o se, al contrario, $\mathcal{C}_1 \equiv \mathcal{C}_2$ è quello di studiare la complessità dei problemi $\mathcal{C}_2$–completi (rispetto ad una riducibilità per la quale $\mathcal{C}_2$ è chiusa). Infatti, se per un qualche problema $\mathcal{P}$ $\mathcal{C}_2$–completo rispetto a tale riducibilità si riesce a mostrare che $\mathcal{P} \in \mathcal{C}_1$, allora, come conseguenza, si è provato che $\mathcal{C}_1 \equiv \mathcal{C}_2$; al contrario, mostrare che $\mathcal{P} \notin \mathcal{C}_1$ implica banalmente che $\mathcal{C}_1 \subset \mathcal{C}_2$.

Definizione 8.19 *Una classe di complessità $\mathcal{C}$ è detta* chiusa *rispetto ad una riducibilità $\leq_r$ se e solo se per ogni coppia di problemi $\mathcal{P}_1$, $\mathcal{P}_2$ tali che $\mathcal{P}_1 \leq_r \mathcal{P}_2$, $\mathcal{P}_2 \in \mathcal{C}$ implica $\mathcal{P}_1 \in \mathcal{C}$.*

La proprietà di chiusura di una classe (rispetto ad una riducibilità) risulta di rilevante importanza per derivare risultati di *collasso* di classi di complessità, per mostrare cioè che, per due classi $\mathcal{C}_1 \subseteq \mathcal{C}_2$, si ha $\mathcal{C}_1 = \mathcal{C}_2$.

L'approccio più immediato per derivare tale risultato è il seguente:

1. individuare una riducibilità $\leq_r$ tale che $\mathcal{C}_1$ è chiusa rispetto a $\leq_r$;
2. identificare un problema $\mathcal{P}$ $\mathcal{C}_2$–completo rispetto a $\leq_r$;
3. dimostrare che $\mathcal{P} \in \mathcal{C}_1$.

Infatti, da ciò deriva che per ogni $\mathcal{P}_1 \in \mathcal{C}_2$ si ha $\mathcal{P}_1 \leq_r \mathcal{P} \in \mathcal{C}_1$, da cui consegue, per la chiusura di $\mathcal{C}_1$, che $\mathcal{P}_1 \in \mathcal{C}_1$.

Esercizio 8.7 Dimostrare che le classi $\mathrm{P}, \mathrm{NP}, \mathrm{PSPACE}, \mathrm{EXPTIME}$ sono tutte chiuse rispetto alla Karp–riducibilità polinomiale.

Capitolo 9

Trattabilità ed intrattabilità dei problemi

Dal punto di vista delle applicazioni pratiche l'analisi della complessità dei problemi è volta ad individuare per quali problemi sia possibile determinare le relative soluzioni in modo efficiente, cioè con un costo computazionale che cresce al più in modo polinomiale rispetto alla taglia dell'input, e per quali problemi, al contrario, è invece necessario utilizzare algoritmi di costo esponenziale, o comunque super-polinomiale. I problemi del primo tipo sono detti *trattabili* e quelli del secondo tipo *intrattabili*.

In questo capitolo prendiamo in considerazione, in particolare, alcune classi di complessità più direttamente legate alla determinazione della trattabilità o intrattabilità computazionale: le classi P, NP e PSPACE. Come vedremo, anche se (vedi Sezione 8.7) è possibile mostrare facilmente delle relazioni di inclusione tra tali classi, tuttavia, per ogni coppia di tali classi, il problema di determinare se l'inclusione è propria o meno, vale a dire se le due classi coincidono o no, è un problema aperto. Il più importante di tali problemi, probabilmente il più rilevante problema aperto nella teoria degli algoritmi, si chiede se le classe P dei problemi decidibili in tempo polinomiale mediante una macchina di Turing deterministica e la classe NP dei problemi accettabili in tempo polinomiale da una macchina di Turing non deterministica coincidono o meno.

9.1 La classe P

La classe P viene tradizionalmente considerata come una ragionevole caratterizzazione formale dei problemi trattabili.

Ciò è dovuto alle seguenti osservazioni:

1. La crescita di funzioni non polinomiali (ad esempio esponenziali) è così rapida che i valori diventano rapidamente troppo grandi. Questo può essere osservato in Figura 9.1, dove viene mostrato il numero di passi eseguiti da algoritmi con diverse complessità temporali, al crescere della dimensione dell'istanza.

   Assumendo che sia utilizzata una macchina che esegue un passo di computazione in 10 nanosecondi (10^{-8} secondi), otteniamo per gli stessi algoritmi i tempi di esecuzione in Figura 9.2 (si noti che l'età dell'universo viene stimata tra 10^{10} e $2 \cdot 10^{10}$ anni).

2. Per n sufficientemente piccolo, un algoritmo polinomiale può eseguire più passi di uno esponenziale, implicando quindi che eventualmente un algoritmo esponenziale potrebbe fornire migliori prestazioni di uno polinomiale su tutte le istanze trattate "in pratica". Comunque, tale fenomeno è piuttosto raro, anche in conseguenza del fatto che in quasi tutti i casi gli algoritmi polinomiali individuati hanno una complessità limitata da un polinomio di grado basso (raramente maggiore di 5) e con costanti moltiplicative piccole. In conseguenza di ciò, per praticamente tutte le istanze di interesse pratico, un algoritmo polinomiale richiede molto meno tempo di calcolo di uno non polinomiale.

3. Gli algoritmi polinomiali sono gli unici per i quali eventuali avanzamenti tecnologici comportano la possibilità di trattare, negli stessi tempi, istanze di dimensioni significativamente maggiori. In Figura 9.3 viene illustrato come varino le dimensioni delle istanze trattabili nello stesso tempo di computazione per algoritmi di diversa complessità, nei casi in cui il singolo passo di computazione possa essere eseguito in tempi più rapidi di un fattore (10, 1000, 10^6, 10^9) o, equivalentemente, se si assume che vengano usate macchine parallele con 10, 1000, 10^6, 10^9 processori (senza costi aggiuntivi di comunicazione tra i processori stessi).

	$n = 2$	$n = 5$	$n = 10$	$n = 100$	$n = 1000$
n	2	5	10	100	1000
n^2	4	25	100	10^4	10^6
n^3	8	125	1000	10^6	10^9
n^6	64	15625	10^6	10^{12}	10^{18}
2^n	4	32	$\approx 10^3$	$\approx 10^{30}$	$\approx 10^{300}$
2^{n^2}	16	$\approx 3 \cdot 10^7$	$\approx 10^{30}$	$\approx 10^{3000}$	$\approx 10^{300000}$

Figura 9.1 Passi eseguiti da algoritmi con diverse complessità

Tipici problemi di complessità polinomiale, che vengono quindi efficientemente risolti dai calcolatori nelle applicazioni quotidiane, sono i problemi di ordina-

	$n = 2$	$n = 5$	$n = 10$	$n = 100$	$n = 1000$
n	20 ns	50 ns	100 ns	1 μs	10 μs
n^2	40 ns	250 ns	1 μs	100 μs	10 ms
n^3	80 ns	1.25 μs	10 μs	10 μs	10 sec
n^6	640 ns	150 μs	10 ms	$\approx$ 3 ore	$\approx$ 300 anni
2^n	40 ns	320 ns	$\approx$ 10 μs	$\approx 10^{14}$ anni	$\approx 10^{284}$ anni
2^{n^2}	160 ns	0.3 sec	$\approx 10^{14}$ anni	$\approx 10^{3000}$ anni	$\approx 10^{300000}$ anni

Figura 9.2 Tempi di calcolo di algoritmi con diverse complessità

	t	$t/10$	$t/1000$	$t/10^6$	$t/10^9$
n	N	$10 \cdot N$	$1000 \cdot N$	$N \cdot 10^6$	$N \cdot 10^9$
n^2	N	$\approx 3.16 \cdot N$	$\approx 31.6 \cdot N$	$1000 \cdot N$	$\approx N \cdot 31600$
n^3	N	$\approx 2.15 \cdot N$	$10 \cdot N$	$100 \cdot N$	$1000 \cdot N$
n^6	N	$\approx 1.46 \cdot N$	$\approx 3.16 \cdot N$	$10 \cdot N$	$\approx 31.6 \cdot N$
2^n	N	$\approx N + 3.32$	$\approx N + 9.96$	$\approx N + 19.93$	$\approx N + 29.89$
2^{n^2}	N	$< N + 1.82$	$< N + 3.15$	$< N + 4.46$	$< N + 5.46$

Figura 9.3 Aumento della dimensione delle istanze trattabili in uno stesso limite di tempo ottenuto da incrementi di velocità di calcolo

mento, di interrogazione di basi di dati, di ricerca di percorsi di costo minimo in grafi, di programmazione lineare, etc.

Si noti che, allo stato attuale delle conoscenze, è un problema aperto se tutti i problemi appartenenti alla classe P abbiano effettivamente un livello di complessità equivalente. Ad esempio, mentre sappiamo che alcuni problemi in P sono risolubili in spazio logaritmico nella dimensione dell'input, per altri problemi, che potrebbero quindi essere più difficili, non si sa se tale proprietà sia verificata, nel senso che, anche se non è stato derivato un lower bound super-logaritmico sulla quantità di spazio necessario per la soluzione, non è stato però individuato alcun algoritmo di soluzione che usi spazio logaritmico.

Le riducibilità polinomiali rappresentano tuttavia uno strumento eccessivamente potente per riuscire a discriminare la difficoltà di soluzione di problemi diversi in P. Infatti, non è difficile rendersi conto che, per ogni coppia di problemi $\mathcal{P}_1, \mathcal{P}_2 \in$ P, $\mathcal{P}_1 \leq_m^p \mathcal{P}_2$ e $\mathcal{P}_2 \leq_m^p \mathcal{P}_1$.

Al fine di caratterizzare la complessità relativa di problemi in P, facciamo riferimento ad un tipo di riducibilità meno potente, una versione di Karp-riducibilità vincolata ad operare in spazio logaritmico.

Definizione 9.1 *Dati due problemi $\mathcal{P}_1, \mathcal{P}_2$, diciamo che $\mathcal{P}_1$ è log-space Karp-riducibile a $\mathcal{P}_2$ ($\mathcal{P}_1 \leq_m^{logsp} \mathcal{P}_2$) se e solo se $\mathcal{P}_1$ è Karp–riducibile a $\mathcal{P}_2$ e la riduzione relativa $\mathcal{R}$ è un algoritmo calcolabile in spazio logaritmico.*

Dato che un algoritmo che termina utilizzando spazio logaritmico non può richiedere tempo più che polinomiale: da ciò deriva che una Karp-riduzione *log*-space è anche polinomiale (ma non viceversa), per cui la Karp-riducibilità *log*-space è in effetti meno potente della Karp-riducibilità polinomiale.

È quindi possibile individuare problemi P-completi (rispetto alla Karp-riducibilità *log*-space): tali problemi, pur essendo risolubili comunque in tempo polinomiale, sono comunque in un qualche senso, in virtù della loro completezza, i più "difficili" tra i problemi in P. Si noti che la maggiore difficoltà rappresentata dalla relazione di Karp-riducibilità *log*-space non ha nulla a che fare con la complessità di risoluzione dei problemi: in altri termini, il fatto che un problema $\mathcal{P}_1$ è più difficile di un altro problema $\mathcal{P}_2$ in quanto $\mathcal{P}_2 \leq_m^{logsp} \mathcal{P}_1$ è cosa ben diversa dalla maggiore difficoltà rappresentata dal fatto che $\mathcal{P}_1$ è risolubile in tempo $\Theta(n^k)$, mentre $\mathcal{P}_2$ è risolubile in tempo $\Theta(n^h)$, con $k > h$.

Esercizio 9.1 Dimostrare che la relazione di Karp-riducibilità *log*-space è transitiva.

Come corollario immediato di quanto detto sopra consegue la proprietà che se un problema P-completo è risolubile in spazio logaritmico, allora P = LOGSPACE.

Un'altra caratteristica rilevante dei problemi P-completi deriva dalla seguente osservazione: definito un problema come *efficientemente parallelizzabile* se ogni istanza di dimensione n è risolubile da $O(n^k)$ processori operanti in parallelo in tempo $O(\log^h n)$, con h, k costanti opportune, allora è possibile mostrare che, dati due problemi $\mathcal{P}_1, \mathcal{P}_2$, se $\mathcal{P}_1$ è efficientemente parallelizzabile e $\mathcal{P}_2 \leq_m^{logsp} \mathcal{P}_1$, allora anche $\mathcal{P}_2$ è efficientemente parallelizzabile.[1]

Da tale proprietà deriva che il mostrare che un qualche problema P-completo è efficientemente parallelizzabile comporta provare che *ogni* problema in P è parallelizzabile in modo efficiente.

9.2 P-completezza

La P-completezza di un problema di decisione $\mathcal{P} \in$ P viene generalmente mostrata individuando una Karp riduzione *log*-space da qualche altro problema P-completo $\mathcal{P}_1$. Infatti, dato che per ogni $\mathcal{P}_2 \in$ P si ha per definizione che $\mathcal{P}_2 \leq_m^{logsp} \mathcal{P}_1$, mostrare che $\mathcal{P}_1 \leq_m^{logsp} \mathcal{P}$ comporta, per la transitività della Karp-riducibilità *log*-space (vedi Esercizio 9.1), che $\mathcal{P}_2 \leq_m^{logsp} \mathcal{P}$, e quindi che $\mathcal{P}$ è P-completo. È allora necessario identificare, mediante qualche altra tecnica, un problema P-completo *iniziale*.

Prima di introdurre il problema Valore Calcolato da Circuito, che utilizzeremo come problema P-completo iniziale (vedi Teorema 9.2), consideriamo alcune definizioni preliminari.

[1] La classe dei problemi efficientemente parallelizzabili è comunemente indicata come NC, e gode della proprietà di chiusura rispetto alla Karp-riducibilità *log*-space.

Definizione 9.2 *Un* circuito booleano *è un grafo orientato aciclico in cui ogni nodo ha* $d_{in} \leq 2$ *archi entranti. In particolare:*

1. *i nodi aventi* $d_{in} = 0$ *sono detti* nodi di input*;*
2. *i nodi aventi* $d_{in} = 1$ *sono detti* porte NOT*;*
3. *ogni nodo avente* $d_{in} = 2$ *è una* porta OR *oppure una* porta AND.

Tutti i nodi hanno $d_{out} > 0$ *archi uscenti, eccetto un solo* nodo di output.

La *dimensione* del circuito è data dal numero di nodi del circuito stesso, mentre la sua *profondità* è pari alla lunghezza del più lungo cammino da un nodo di input al nodo di output.

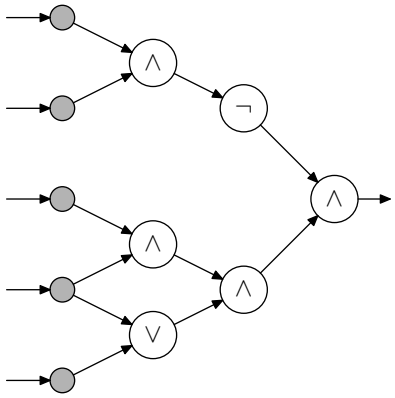


FIGURA 9.4 Esempio di circuito.

Sia $I(\mathcal{C})$, con $|I(\mathcal{C})| = n$, l'insieme dei nodi input di un circuito $\mathcal{C}$: data allora una qualunque assegnazione $\langle b_0, b_1, \ldots, b_{n-1}\rangle \in \{0,1\}^n$ di valori booleani[2] agli n nodi di input, il circuito calcola, applicando su ogni porta l'operatore ad essa associato ai relativi input, i valori booleani in uscita su tutti i nodi del circuito stesso. In particolare , $\mathcal{C}$ calcola la funzione $f_{\mathcal{C}} : \{0,1\}^n \mapsto \{0,1\}$ tale che $f_{\mathcal{C}}(b_0, b_1, \ldots, b_{n-1}) = 1$ se e solo se, con input $b_0, b_1, \ldots, b_{n-1}$, il circuito $\mathcal{C}$ restituisce il valore 1 in uscita al nodo di output.

Esempio 9.1 Il circuito booleano in Figura 9.4 ha 5 nodi di input e 6 porte, delle quali la più a destra è anche nodo di output. Se indichiamo con $b_0, \ldots, b_4$ i valori booleani associati ai nodi di input (dall'alto in basso), il circuito booleano calcola la funzione $(\overline{b_0 \wedge b_1}) \wedge ((b_2 \wedge b_3) \wedge (b_3 \vee b_4))$.

[2]Nel seguito di questa sezione, indicheremo i valori booleani come 0 (per FALSO) e (per VERO).

Definizione 9.3 *Dato un circuito $\mathcal{C}$ con $|I(\mathcal{C})|= n$, il* linguaggio deciso *da $\mathcal{C}$ è l'insieme delle stringhe in $w \in \{0,1\}^n$ tali che $f_{\mathcal{C}}(w) = 1$.*

Possiamo ora definire il problema di decisione Valore Calcolato da Circuito, il quale chiede, dato un circuito booleano ed una sua stringa di input, se il circuito restituisce il valore 1.

Valore Calcolato da Circuito
Istanza: Circuito booleano $\mathcal{C}$ con nodi di input $I(\mathcal{C})$ ($\|I(\mathcal{C})\|= n$), stringa $w \in \{0,1\}^n$.
Predicato: Si ha $f_{\mathcal{C}}(w) = 1$?

Al fine di dimostrare che Valore Calcolato da Circuito è P-completo, mostriamo ora il seguente teorema, che ci dice che ogni linguaggio in P può essere accettato da circuiti di dimensione polinomiale.

Teorema 9.1 *Sia $L \in \{0,1\}^n$ un linguaggio tale che $L \in P$; allora esiste un circuito $\mathcal{C}$ con n nodi di input e di dimensione e profondità polinomiali in n che decide L.*

Dimostrazione. Se $L \in$ P allora esiste una macchina di Turing $\mathcal{M} = \langle \Gamma, \b, Q, q_0, F, \delta \rangle$, con alfabeto di input $\Sigma = \{0,1\} \subseteq \Gamma = \{a_1, a_2, \ldots, a_{\|\Gamma\|}\}$, che decide ogni stringa $w \in L$ in al più $p(n)$ passi, dove $p(n)$ è un polinomio. Assumiamo che $\mathcal{M}$ operi su un solo nastro semi-infinito (vedi Teorema 5.3), nelle cui prime n celle l'input è inizialmente contenuto.

Consideriamo, senza perdita di generalità, la macchina di Turing $\mathcal{M}' = \langle \Gamma, \b, Q, q_0, F, \delta' \rangle$, la cui funzione di transizione è definita nel modo seguente:

$$\delta'(q,a) = \begin{cases} \delta(q,a) \text{ se } \delta(q,a) \text{ è definita} \\ (q,a,i) \text{ altrimenti.} \end{cases}$$

La macchina di Turing $\mathcal{M}'$ ha lo stesso comportamento di $\mathcal{M}$ eccetto che ad ogni computazione massimale di $\mathcal{M}$ corrisponde una computazione di $\mathcal{M}'$ che cicla indefinitamente sulla stessa configurazione. Quindi, ad ogni computazione accettante di $\mathcal{M}$ corrisponde una computazione di $\mathcal{M}'$ che cicla su una configurazione di accettazione, mentre ad ogni computazione non accettante di $\mathcal{M}$ corrisponde una computazione di $\mathcal{M}'$ che cicla su una configurazione non di accettazione. Dato che $\mathcal{M}$ decide ogni stringa in al più $p(n)$ passi, ne consegue che, data una stringa w in input, la configurazione raggiunta da $\mathcal{M}'$ dopo avere eseguito $p(n)$ passi è di accettazione se $w \in L$ e di non accettazione altrimenti.

Evidentemente, in $p(n)$ passi la testina di $\mathcal{M}'$ può al massimo posizionarsi sulle prime $p(n)+1$ celle di nastro, che quindi rappresentano l'unica parte del nastro di interesse nella computazione.

Si consideri allora una matrice (*tableau*) T di dimensioni $(p(n)+1) \times (p(n)+1)$, dove ogni riga corrisponde ad un istante di tempo (e quindi ad una configurazione) ed ogni colonna ad una particolare cella di nastro. Ogni elemento di T può contenere una coppia in $\bar{\Gamma} \times (Q \cup \{\perp\})$, dove $\perp \notin Q$.

La riga i-esima del tableau descrive nel modo seguente la configurazione di $\mathcal{M}'$ dopo che è stato eseguito l'i-esimo passo di computazione. Sia $\alpha_j \in \bar{\Gamma}$ il contenuto della cella j-esima di nastro $(0 \leq j \leq p(n))$, sia q_k lo stato attuale di $\mathcal{M}'$ e si assuma che la testina sia posizionata sulla cella t-esima: la riga i-esima di T ha allora la forma

$$\langle \alpha_0, \perp \rangle, \langle \alpha_1, \perp \rangle, \ldots, \langle \alpha_{t-1}, \perp \rangle, \langle \alpha_t, q_k \rangle, \langle \alpha_{t+1}, \perp \rangle, \ldots, \langle \alpha_{p(n)}, \perp \rangle.$$

Si noti che, dato il tableau T rappresentante la computazione effettuata da $\mathcal{M}'$ su input w, è possibile determinare se $w \in L$ semplicemente osservando se la configurazione rappresentata nell'ultima riga di T è di accettazione, e quindi se per l'unica cella in tale riga che contiene una coppia il cui secondo elemento è diverso da $\perp$, tale elemento è un qualche stato finale $q_F \in F$. È inoltre importante osservare che, per come è definita la funzione di transizione di una macchina di Turing, il contenuto della cella $T[i,j]$ del tableau è determinato (attraverso la funzione di transizione stessa) soltanto dal contenuto delle tre celle $T[i-1, j-1]$, $T[i-1, j]$ e $T[i-1, j+1]$.

Il circuito $\mathcal{C}$ essenzialmente si limita a codificare, attraverso opportuni blocchi di porte, tale dipendenza del contenuto di una cella di T dalle tre celle della riga precedente. Il circuito calcola, in particolare, il seguente insieme di funzioni:

1. $S(i,j,k)$ $(0 \leq i \leq p(n), 0 \leq j \leq p(n), 0 \leq k \leq |\, Q \,|)$, definita come $S(i,j,k) = 1$ se e solo se il secondo componente del contenuto di $T[i,j]$ è $\perp$, se $k = |\, Q \,|$, o q_k se $k < |\, Q \,|$.

2. $C(i,j,k)$ $(0 \leq i \leq p(n), 0 \leq j \leq p(n), 0 \leq k \leq |\, \Gamma \,|)$, definita come $S(i,j,k) = 1$ se e solo se il primo componente del contenuto di $T[i,j]$ è $\bar{b}$, se $k = 0$, o a_k se $k > 0$.

Il circuito $\mathcal{C}$ ha dimensione $O((p(n))^2)$ e profondità $O(p(n))$, ed è strutturato su tre sezioni (di input, di calcolo, di output) nel modo seguente (vedi Figura 9.5).

1. La sezione di input è composta da una riga di n nodi di input corrispondenti agli n caratteri $\{0,1\}$ in input, e da una seconda riga comprendente $p(n)+1$ blocchi di porte, ognuno dei quali è associato ad una cella del nastro di $\mathcal{M}'$. Il primo blocco $B_{0,0}$ è collegato al primo nodo di input, corrispondente al primo carattere b_0 di w, e fornisce in output i valori delle funzioni $S(0,0,k)$ (dove $S(0,0,0) = 1$ e $S(0,0,k) = 0$ per $k > 0$) e delle funzioni $C(0,0,k)$ (dove $C(0,0,k) = 1$ se $b_0 = a_k$ e $C(0,0,k) = 0$ altrimenti).

Ogni blocco $B_{0,j}$, con $1 \leq j \leq n-1$, è collegato al nodo di input corrispondente al j-esimo carattere b_0 di w, e fornisce in output i valori delle funzioni $S(0,j,k)$ (dove $S(0,j,k) = 0$ per $k \geq 0$) e delle funzioni $C(0,j,k)$ (dove $C(0,j,k) = 1$ se $b_j = a_k$ e $C(0,j,k) = 0$ altrimenti).

Ogni blocco $B_{0,j}$, con $n \leq j \leq p(n)$, è collegato al nodo di input corrispondente all'n-esimo carattere b_{n-1} di w, e fornisce in output, in modo del tutto indipendente dal valore di b_{n-1}, i valori delle funzioni $S(0,j,k)$ (dove $S(0,j,k) = 0$ per $k \geq 0$) e delle funzioni $C(0,j,k)$ (dove $C(0,j,0) = 1$ e $C(0,j,k) = 0$ per $k > 0$).

Si osservi che i blocchi $B_{0,j}$ con $1 \leq j \leq n-1$ calcolano la stessa funzione e quindi sono identici tra loro, così come i blocchi $B_{0,j}$ con $n \leq j \leq p(n)$; inoltre, la dimensione di tali blocchi è chiaramente indipendente da n. Da ciò deriva che la sezione di input ha dimensione $O(p(n))$ e profondità $O(1)$.

2. La sezione di calcolo è composta da $p(n)$ righe (una per ogni passo di computazione di $\mathcal{M}'$). La riga i-esima è composta da $p(n)+1$ blocchi identici (uno per ogni cella del nastro di $\mathcal{M}'$): il blocco $B_{i,j}$ determina il contenuto della cella $T[i,j]$ del tableau in funzione del contenuto delle tre celle $T[i-1,j-1]$, $T[i-1,j]$ e $T[i-1,j+1]$.[3]

   In particolare, il blocco $B_{i,j}$ ha una struttura dipendente dalla funzione di transizione δ' e riceve in input i valori delle funzioni $S(i-1,t,k)$ $(j-1 \leq t \leq j+1,\ 0 \leq k \leq |\, Q \,|)$ e $C(i-1,t,k)$ $(j-1 \leq t \leq j+1,\ 0 \leq k \leq |\, \Gamma \,|)$. A partire da tali valori, $B_{i,j}$ calcola le funzioni $S(i,j,k)$ $(0 \leq k \leq |\, Q \,|)$ e $C(i,j,k)$ $(0 \leq k \leq |\, \Gamma \,|)$.

   Anche la dimensione dei blocchi nella sezione di calcolo è indipendente da n, per cui abbiamo che la sezione di calcolo ha dimensione $O(p(n)^2)$ e profondità $O(p(n))$.

3. La sezione di output è composta da una prima riga di $p(n)+1$ blocchi identici, ognuno associato ad una cella del nastro di $\mathcal{M}'$. Il blocco $B_{p(n)+1,j}$ restituisce un solo valore booleano, pari ad 1 se e solo se il secondo componente del contenuto di $T[p(n),j]$ è uno stato finale $q_F \in F$. Il blocco $B_{p(n)+1,j}$ riceve quindi in input i valori delle funzioni $S(p(n),j,k)$ $(0 \leq k \leq |\, Q \,|)$ e restituisce un valore pari a $\bigvee_{q_k \in F} S(p(n),j,k)$. Si osservi che tutti i blocchi su tale riga sono identici ed hanno dimensione indipendente da n; si osservi inoltre che la configurazione raggiunta è di accettazione se e solo se esiste uno dei blocchi $B_{p(n)+1,j}$ che restituisce valore 1.

   La seconda riga della sezione di output è composta da $\lceil n/2 \rceil$ porte OR $B_{p(n)+2,j}$ ognuna delle quali riceve in input i valori calcolati dai due

[3] Naturalmente, se $j = 0$ la dipendenza è solo dalle celle $T[i-1,0]$ e $T[i-1,1]$, mentre se se $j = p(n)$ la dipendenza è solo dalle celle $T[i-1,p(n)-1]$ e $T[i-1,p(n)]$.

blocchi $B_{p(n)+1,2j}$ e $B_{p(n)+1,2j+1}$. La terza riga, composta da $\lceil n/4 \rceil$ porte OR opera allo stesso modo sui valori restituiti dai blocchi della riga precedente, e così via. La $\lceil \log_2(p(n)+1) \rceil$-esima riga ha una sola porta OR, definita come nodo di output del circuito, che calcola il valore 1 se e solo se la stringa viene accettata da $\mathcal{M}$ in tempo al più $p(n)$.

Chiaramente, la sezione di output ha dimensione $O(p(n))$ e profondità $O(\log(p(n))) = O(\log n)$.

Per concludere, osserviamo che, come si voleva dimostrare, il circuito $\mathcal{C}$ così costruito ha dimensione $O(p(n)^2)$ e profondità $O(p(n))$, quindi polinomiali in n, e, ricevendo in input una stringa $w \in \{0,1\}^n$, restituisce valore 1 se e solo se $w \in L$. □

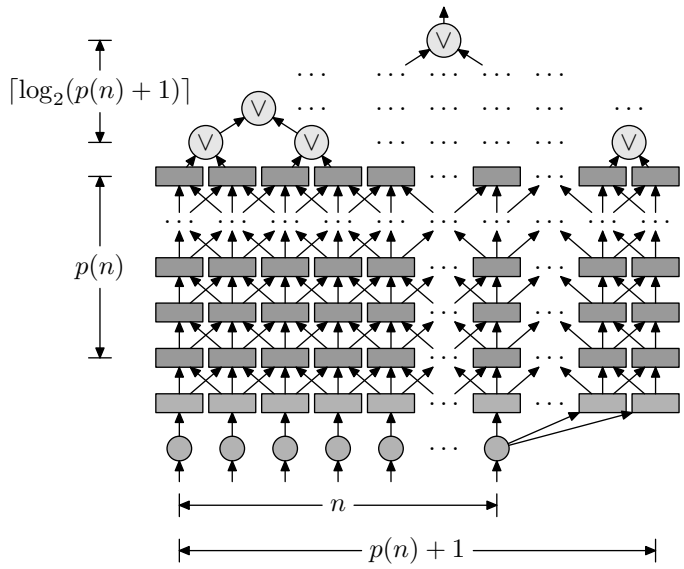


FIGURA 9.5 Struttura del circuito nella dimostrazione del Teorema 9.1.

Possiamo ora mostrare che il problema VALORE CALCOLATO DA CIRCUITO è tale che ogni problema in P può essere ridotto ad esso mediante una Karp-riduzione *log*-space.

Teorema 9.2 *Il problema* VALORE CALCOLATO DA CIRCUITO *è* P*-completo rispetto alla Karp-riducibilità* log*-space.*

Dimostrazione. Per mostrare che VALORE CALCOLATO DA CIRCUITO è in P basta osservare che, per la aciclicità del circuito, è possibile, mediante una

visita in ordine topologico dei nodi, determinare il valore in uscita da tutte le porte, e quindi anche il valore in output, in tempo lineare nella dimensione del circuito.

Il Teorema 9.1 mostra poi una riduzione da ogni problema in P a VALORE CALCOLATO DA CIRCUITO. L'ultimo passo della dimostrazione, che lasciamo come esercizio (vedi Esercizio 9.2) comporta la verifica che tale riduzione è *log*-space. □

Esercizio 9.2 Dimostrare che la riduzione da un qualunque problema in P a VALORE CALCOLATO DA CIRCUITO mostrata nel Teorema 9.1 richiede spazio logaritmico.

Dalla P-completezza di VALORE CALCOLATO DA CIRCUITO sopra dimostrata deriva quindi che se tale problema fosse efficientemente parallelizzabile[4] allora così sarebbe ogni problema risolubile in tempo polinomiale.

Un aspetto interessante nella dimostrazione precedente è che, come vedremo, la sua struttura di fondo viene ripresa nella dimostrazione del Teorema di Cook (Teorema 9.4) che enuncia un corrispondente risultato per la classe NP, identificando un primo problema NP-completo, così come il Teorema 9.2 identifica un primo problema P-completo.

Un ulteriore esempio di problema P-completo è fornito dal seguente problema SODDISFACIBILITÀ DI FORMULE DI HORN, per introdurre il quale forniamo dapprima la seguente definizione.

Definizione 9.4 *Una clausola di Horn su un insieme X di variabili booleane è una disgiunzione $C = \overline{x}_1 \vee x_2 \vee \ldots x_t$ di termini relativi a tutte o a parte delle variabili in X, con il vincolo che in esattamente uno di tali termini la variabile corrispondente compare vera, mentre in tutti gli altri compare negata.*

Si osservi che, dato che per l'implicazione logica vale l'equivalenza tra $x \vee \overline{y}$ e $y \Rightarrow x$, abbiamo che ogni clausola di Horn corrisponde ad una implicazione.

Osserviamo inoltre che una implicazione del tipo $x \wedge y \wedge z \Rightarrow w$ corrisponde alla clausola di Horn $\overline{x} \vee \overline{y} \vee \overline{z} \vee w$, e che una implicazione $x \vee y \vee z \Rightarrow w$ corrisponde alla formula $(\overline{x} \wedge \overline{y} \wedge \overline{z}) \vee w$ e quindi, equivalentemente, all'insieme di clausole di Horn $\overline{x} \vee w$, $\overline{y} \vee w$, $\overline{z} \vee w$.

Definizione 9.5 *Una formula di Horn $\mathcal{F}_H$ su un insieme X di variabili booleane è una congiunzione $\mathcal{F}_H = C_1 \wedge C_2 \wedge \ldots \wedge C_t$ di clausole di Horn definite su X.*

Si noti che una formula di Horn non è altro che u caso speciale di formula CNF, ed in effetti il problema SODDISFACIBILITÀ DI FORMULE DI HORN non

[4] Si osservi che VALORE CALCOLATO DA CIRCUITO è un problema che per sua natura si presenta come "parallelo", in quanto le porte in un circuito booleano operano in condizioni di parallelismo. La profondità del circuito descritto in una istanza di VALORE CALCOLATO DA CIRCUITO non è però necessariamente polinomiale nel logaritmo della sua dimensione.

è altro che il problema SODDISFACIBILITÀ ristretto al caso particolare in cui si considerino formule di Horn.

SODDISFACIBILITÀ DI FORMULE DI HORN

ISTANZA: Una formula di Horn $\mathcal{F}_H$ su un insieme X di variabili booleane

PREDICATO: Esiste una assegnazione $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$ che soddisfa $\mathcal{F}_H$?

Teorema 9.3 *Il problema* SODDISFACIBILITÀ DI FORMULE DI HORN *è* P*-completo rispetto alla Karp-riducibilità* log*-space.*

Dimostrazione. Come prima cosa mostriamo che il problema SODDISFACIBILITÀ DI FORMULE DI HORN è in P: a tal fine, è possibile derivare un algoritmo polinomiale che verifica se una formula di Horn $\mathcal{F}_H$ è soddisfacibile (e che anzi, in tal caso, costruisce l'assegnamento di verità corrispondente) semplicemente osservando che:

- se una clausola ha un solo termine, che quindi è una variabile vera, allora tale variabile deve essere posta a VERO;
- se in una clausola con più di un termine tutte le variabili che compaiono negate sono state poste a VERO, allora la variabile che compare vera deve anch'essa essere posta a VERO.

Per mostrare la P-completezza di SODDISFACIBILITÀ DI FORMULE DI HORN introduciamo il seguente problema, restrizione di VALORE CALCOLATO DA CIRCUITO al caso in cui si considerino circuiti privi di porte NOT.

VALORE CALCOLATO DA CIRCUITO MONOTONO

ISTANZA: Circuito booleano monotono (privo di porte NOT) $\mathcal{C}$, con nodi di input $I(\mathcal{C})$ ($|I(\mathcal{C})| = n$), stringa $w \in \{0,1\}^n$.

PREDICATO: Si ha $f_{\mathcal{C}}(w) = 1$?

Il problema VALORE CALCOLATO DA CIRCUITO MONOTONO può essere facilmente mostrato essere P-completo (vedi Esercizio 9.4).

Mostriamo ora che esiste una Karp-riduzione *log*-space da VALORE CALCOLATO DA CIRCUITO MONOTONO a SODDISFACIBILITÀ DI FORMULE DI HORN. La riduzione opera nel modo seguente: l'insieme X delle variabili corrisponde all'insieme dei nodi del circuito. L'insieme delle clausole viene derivato nel modo seguente:

1. ad ogni nodo di input, cui è associata ad esempio la variabile x_i, se al nodo viene assegnato il valore 1 in input, introduciamo la clausola x_i, altrimenti, se al nodo viene assegnato il valore 0 in input, introduciamo la clausola $\overline{x}_i$;

2. per ogni porta AND, siano x_i e x_j le variabili associate alle due porte da cui essa riceve i propri input: se x_k è la variabile ad essa associata, alla porta corrispondo le tre clausole $\overline{x}_i \vee \overline{x}_j \vee x_k$, $x_i \vee \overline{x}_k$ e $x_j \vee \overline{x}_k$;

3. per ogni porta OR, siano x_i e x_j le variabili associate alle due porte da cui essa riceve i propri input: se x_k è la variabile ad essa associata, alla porta corrispondono le tre clausole $x_i \vee x_j \vee \overline{x}_k$, $\overline{x}_i \vee x_k$ e $\overline{x}_j \vee x_k$;

4. sia x_t la variabile associata al nodo di output: introduciamo allora la clausola x_t.

Non è difficile verificare che la formula di Horn derivata in tal modo è soddisfacibile se e solo l'istanza corrispondente di Valore Calcolato da Circuito è una istanza positiva del problema. In particolare, il valore assunto da ogni variabile nell'assegnazione di verità che soddisfa la formula specifica il valore calcolato dalla porta corrispondente del circuito.

Infine, per verificare che la riduzione è *log*-space basta osservare che per derivare la formula di Horn è sufficiente considerare i nodi del circuito uno alla volta, tenendo traccia dell'indice della relativa variabile e degli indici di eventuali variabili associate ai suoi input. Dato che ognuno di tali indici occupa spazio logaritmico nella dimensione del circuito, e che ogni volta che viene considerato un nodo è necessario rappresentare un numero costante di indici, ne deriva che la riduzione richiede spazio logaritmico. □

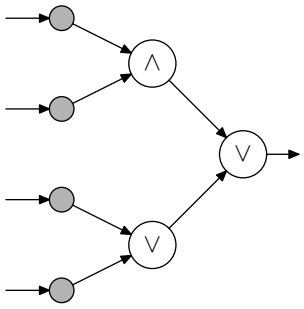


Figura 9.6 Esempio di circuito monotono.

Esempio 9.2 Si consideri l'istanza di Valore Calcolato da Circuito Monotono rappresentata dal circuito in Figura 9.6 e dai valori in input 1001. Se consideriamo la riduzione a Soddisfacibilità di formule di Horn descritta nella dimostrazione del Teorema 9.3 possiamo associare ai nodi di input le variabili x_1, x_2, x_3, x_4, ed alle 3 porte le variabili x_5, x_6, x_7. Per quanto riguarda le clausole, otteniamo quanto segue:

1. i quattro nodi di input danno luogo alle quattro clausole x_1, $\overline{x}_2$, $\overline{x}_3$, x_4;
2. la porta AND al primo livello dà luogo alle tre clausole $\overline{x}_1 \vee \overline{x}_2 \vee x_5$, $x_1 \vee \overline{x}_5$ e $x_2 \vee \overline{x}_5$;
3. la porta OR al primo livello dà luogo alle tre clausole $x_3 \vee x_4 \vee \overline{x}_6$, $\overline{x}_3 \vee x_6$ e $\overline{x}_4 \vee x_6$;
4. la porta AND al secondo livello dà luogo alle tre clausole $\overline{x}_5 \vee \overline{x}_6 \vee x_7$, $x_5 \vee \overline{x}_7$ e $x_6 \vee \overline{x}_7$;
5. il nodo di output corrisponde alla porta AND sul secondo livello, il che dà luogo alla clausola x_7.

Si può verificare che l'unico assegnamento di verità che soddisfa la formula di Horn risultante è dato da $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 0, x_6 = 1, x_7 = 1$, che, come si può vedere, fornisce i valori calcolati dalle varie porte del circuito in presenza dell'input considerato.

Esercizio 9.3 Definire un algoritmo operante in tempo polinomiale che risolva il problema SODDISFACIBILITÀ DI FORMULE DI HORN.

Esercizio 9.4 Dimostrare che la P-completezza del problema VALORE CALCOLATO DA CIRCUITO MONOTONO.
[Suggerimento: Utilizzare la regola di De Morgan $\overline{x \wedge y} = \overline{x} \vee \overline{y}$, $\overline{x \vee y} = \overline{x} \wedge \overline{y}$ per definire una riduzione da VALORE CALCOLATO DA CIRCUITO].

9.3 La classe NP

Molti problemi di decisione di significativa importanza presentano un insieme di caratteristiche comuni. Per tali problemi, infatti, non sono stati individuati algoritmi polinomiali per la loro soluzione, ma d'altra parte non è stata trovata alcuna dimostrazione di una loro esponenzialità: al tempo stesso, tutti questi problemi sono decidibili in tempo polinomiale da macchine di Turing non deterministiche.

Ad esempio, si può dimostrare che il problema SODDISFACIBILITÀ, definito nell'Esempio 8.8 è risolubile in tempo polinomiale da una macchina di Turing non deterministica, che accetta tutte e sole le istanze relative a formule soddisfacibili.

In generale, per ogni problema di decisione $\mathcal{P}$ si vuole, data una istanza x di $\mathcal{P}$ (codificata come stringa di simboli da un qualche alfabeto Σ), determinare un singolo valore binario che esprime il fatto che x appartenga o meno all'insieme $Y_{\mathcal{P}}$ delle istanze positive di $\mathcal{P}$. Come già osservato nella Sezione 8.2, nella maggior parte dei casi determinare se una istanza x appartiene a $Y_{\mathcal{P}}$ comporta l'individuare una soluzione $s(x)$ di x nel corrispondente problema di ricerca: ad esempio, determinare se una istanza di SODDISFACIBILITÀ rappresentata

da una formula CNF $\mathcal{F}$ su un insieme di variabili X è soddisfacibile equivale a chiedersi se esiste una soluzione rappresentata da una assegnazione di verità $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$ che soddisfi la formula $\mathcal{F}$ stessa.

Esercizio 9.5 Mostrare che, data una istanza di SODDISFACIBILITÀ, la relativa soluzione del corrispondente problema di ricerca , se esiste, ha lunghezza polinomiale.

Osserviamo ora che, per ogni problema di decisione $\mathcal{P}$ che richiede di decidere se, data una istanza x del problema, tale istanza abbia una soluzione (e per ogni corrispondente problema di ricerca che, data x, chiede di determinare tale soluzione) è possibile considerare un corrispondente problema di *verifica* il quale, data una istanza x di $\mathcal{P}$ ed una possibile soluzione s, chiede se s sia effettivamente una soluzione di x. Nel caso di SODDISFACIBILITÀ ciò significa che, data una formula $\mathcal{F}$ su un insieme di variabili X ed una assegnazione di verità $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$, si vuole determinare se f soddisfa $\mathcal{F}$.

Si osservi che, in realtà, tutte le considerazioni precedenti valgono ancora se al concetto di soluzione del problema sostituiamo quello, più generale, di certificato.

Definizione 9.6 *Dato un linguaggio $L \subseteq \Sigma^*$, una funzione $c : L \mapsto \Sigma^*$ associa ad ogni stringa $u \in L$ un* certificato *$v = c(u)$ se e solo se che esiste una macchina di Turing deterministica $\mathcal{M}$ la quale decide il linguaggio $L' = \{uv \mid u \in L, v = c(u)\}$.*

In altri termini, un certificato è una informazione aggiuntiva che "dimostra" che una data stringa appartiene al linguaggio L, e che può quindi essere utilizzata per decidere se una stringa u appartiene al linguaggio stesso. Come vedremo più avanti, l'utilizzo del concetto di certificato ci consentirà di formulare una definizione particolarmente utile di problema in NP.

In generale, verificare una soluzione sembra decisamente più semplice che risolvere il problema di decisione nella sua forma originaria. In particolare, si noti che verificare le soluzioni di un problema è equivalente a risolvere, mediante accettazione, il problema stesso in modo non deterministico: infatti, una macchina di Turing non deterministica potrebbe essere definita in modo tale da generare inizialmente, attraverso una sequenza opportuna di passi non deterministici, tutte le stringhe che potrebbero rappresentare possibili soluzioni, eventualmente in numero esponenziale, associando ad ogni possibile soluzione una distinta computazione.

Tale situazione è rappresentata in Figura 9.7, in cui, nella parte sinistra, viene mostrato come, non deterministicamente, vengono generate tutte le stringhe che potrebbero codificare possibili soluzioni (nodi ombreggiati), a partire da ognuno dei quali una computazione deterministica, nella parte destra, verifica se la stringa codifica effettivamente una soluzione dell'istanza.

Nell'ambito di ogni computazione, quindi, la macchina di Turing verifica deterministicamente la soluzione generata: evidentemente, se l'istanza del pro-

blema è una istanza positiva, esisterà una soluzione che soddisfa il predicato posto dal problema e tale soluzione corrisponderà ad uno dei cammini generati dalla macchina di Turing non deterministica, cammino che sarà quindi un cammino accettante. Inoltre, se per il problema considerato le relative soluzioni hanno dimensione polinomiale rispetto all'input la fase di generazione non deterministica di tutte le stringhe richiede tempo polinomiale e quindi, se il costo di verifica di una soluzione è anch'esso polinomiale, la macchina di Turing non deterministica opera in tempo polinomiale.

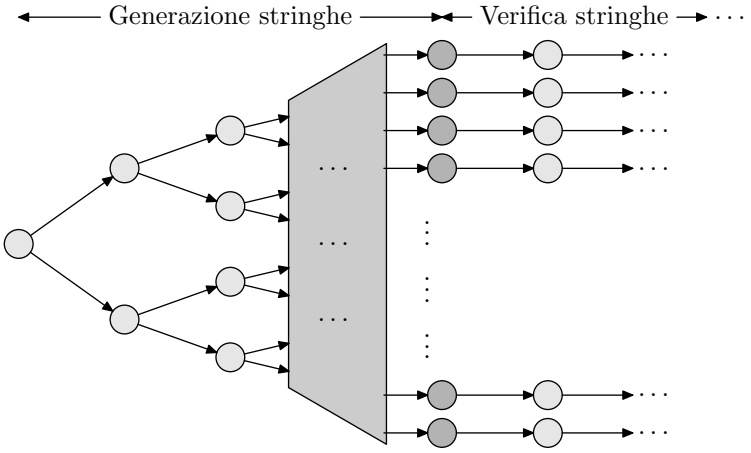


FIGURA 9.7 Schema di soluzione di un problema in NP mediante macchina di Turing non deterministica .

Tali considerazioni permettono di introdurre una diversa definizione di algoritmo (o macchina di Turing) non deterministico, equivalente al modello considerato fino ad ora. Secondo tale modello, un algoritmo non deterministico è un algoritmo che, oltre a tutte le istruzioni usuali tipiche del modello di calcolo adottato, ha la possibilità di eseguire comandi del tipo "'**guess** $y \in \{0,1\}$": un'istruzione di questo tipo, eseguita in un solo passo di computazione, fa sì che y assuma come valore un elemento in $\{0,1\}$.

Essenzialmente, un algoritmo non deterministico può essere interpretato come un algoritmo che ha la possibilità aggiuntiva di *indovinare* (guess) una continuazione nell'ambito di un insieme (finito) di possibili continuazioni della computazione eseguita fino al momento dell'esecuzione dell'istruzione *guess*. Si noti che per ogni insieme S, un elemento di S può essere "indovinato" per mezzo di $O(\log |S|)$ istruzioni "guess" sull'insieme $\{0,1\}$.

A questo punto, possiamo riformulare la definizione di algoritmo non de-

deterministico che risolve un problema di decisione, nell'ambito di questo nuovo modello, nel modo seguente:

Definizione 9.7 *Dato un problema di decisione $\mathcal{P}$, un algoritmo non deterministico $\mathcal{A}$ risolve $\mathcal{P}$ se e solo se, per ogni istanza $x \in Y_{\mathcal{P}}$, esiste almeno una sequenza di* guess *che fa sì che $\mathcal{A}$ restituisca il valore* VERO *e se, per ogni istanza $x \notin Y_{\mathcal{P}}$, non esiste alcuna sequenza di* guess *che fa sì che $\mathcal{A}$ restituisca il valore* VERO.

Si osservi che, senza perdita di generalità, possiamo sempre considerare un modello semplificato di algoritmo non deterministico polinomiale che esegue una sola istruzione *guess* su un insieme di taglia esponenziale, eseguita all'inizio della sua esecuzione. Tale operazione può essere vista come una unica *guess* complessiva dei risultati della sequenza (di lunghezza polinomiale) di *guess* binarie eseguite nell'ambito di ogni computazione non deterministica. In altri termini, assumiamo che le $f(n)$ *guess* binarie eseguite nell'ambito di una computazione siano tutte eseguite all'inizio, con una sola guess su un insieme di dimensione al più $2^{f(n)}$.

Consideriamo ad esempio il seguente problema INSIEME INDIPENDENTE.

INSIEME INDIPENDENTE

ISTANZA: Grafo $G = (V, E)$, $K \in \mathbb{N}$.

PREDICATO: Esiste un insieme indipendente in G di dimensione $\geq K$, cioè un sottoinsieme $U \subseteq V$ tale che $\mid U \mid \geq K$ e $\nexists (u, v) \in E$ con $u \in U$ and $v \in U$?

Ogni istanza positiva $x \in Y_{II}$ di INSIEME INDIPENDENTE ha (almeno) una soluzione associata $s(x)$ rappresentata da un sottoinsieme $U \subseteq V$ dell'insieme dei nodi, con $|U| \geq K$.

Mentre decidere se, dato un grafo G, esiste un insieme indipendente di taglia almeno K è un problema difficile (non è noto alcun algoritmo che lo risolva in tempo polinomiale), è possibile verificare in tempo lineare se un insieme $U \subseteq V$ di nodi è un insieme indipendente di dimensione almeno K.

Un algoritmo non deterministico (polinomiale) per risolvere il probelma INSIEME INDIPENDENTE è dato in Figura 9.1. Come si può osservare, l'algoritmo essenzialmente "indovina" un possibile insieme indipendente tra i $2^{|V|}$ possibili e quindi verifica se la *guess* ha avuto successo. Si noti che la descrizione "indovinata" di un insieme indipendente ha lunghezza polinomiale (lineare, in effetti) nella dimensione dell'input e la sua generazione richiede una quantità polinomiale (lineare) di *guess* binarie. Inoltre, dato che la fase di verifica richiede tempo polinomiale, l'algoritmo non deterministico presenta complessità polinomiale.

A questo punto, la classe NP può essere definita, in modo alternativo, come la classe di tutti i problemi che possono essere risolti da algoritmi deterministici che operano in tempo polinomiale a partire da *guess* di lunghezza polinomiale.

```
input G = (V, E): graph;
output boolean;
begin
  guess U ⊆ V;
  if | U |< K then return FALSO;
  for each (u, v) ∈ E do
    if u ∈ U and v ∈ U then return FALSO;
  return VERO
end.
```

Algoritmo 9.1: Algoritmo non deterministico per INSIEME INDIPENDENTE

Come preannunciato, una diversa caratterizzazione dei linguaggi in NP può essere introdotta adattando in modo opportuno il concetto di certificato.

Definizione 9.8 *Dato un linguaggio $L \subseteq \Sigma^*$, $L \in$ NP se e solo se esiste una funzione $c : L \mapsto \Sigma^*$ che associa ad ogni stringa $u \in L$ un* certificato $v = c(u)$ di lunghezza polinomiale verificabile in tempo polinomiale*, vale a dire tale che:*

1. *v ha lunghezza polinomiale in $|u|$;*
2. *la macchina di Turing deterministica $\mathcal{M}$ decide il linguaggio $L' = \{uv \mid u \in L, v = c(u)\}$ in tempo polinomiale.*

Come è possibile osservare, nel caso di problemi in NP il concetto di soluzione di un problema non è altro che un caso particolare del concetto di certificato di lunghezza polinomiale verificabile in tempo polinomiale.

La questione "P = NP?" può essere interpretato come chiedersi se verificare efficientemente (in tempo polinomiale) un problema di decisione è equivalente come difficoltà a risolvere efficientemente lo stesso problema.

Ricordiamo ancora una volta che, come già osservato in precedenza, la presenza del non determinismo comporta una asimmetria tra istanze in $x \in Y_{\mathcal{P}}$ ed istanze in $x \in N_{\mathcal{P}} \cup D_{\mathcal{P}}$. Infatti, mentre affinché una istanza x sia riconosciuta come appartenente a $x \in Y_{\mathcal{P}}$ deve esistere *almeno una* computazione (*guess*) accettante, affinché la stessa istanza sia riconosciuta in $x \in N_{\mathcal{P}} \cup D_{\mathcal{P}}$ sarà necessario verificare che *tutte* le computazioni siano non accettanti. Anche nel caso in cui tutte le computazioni terminino, questo rende la condizione da verificare inerentemente diversa, richiedendo la verifica rispetto ad un quantificatore universale, piuttosto che rispetto ad un quantificatore esistenziale.

Consideriamo ad esempio il seguente problema di decisione FALSITÀ.

FALSITÀ

ISTANZA: Formula $\mathcal{F}$ in forma normale congiuntiva, definita su un insieme X di variabili booleane

PREDICATO: Non esiste nessuna assegnazione $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$ che soddisfa $\mathcal{F}$

Chiaramente, FALSITÀ è definito sullo stesso insieme di istanze di SODDISFACIBILITÀ, ma il suo insieme di istanze positive corrisponde alle istanze negative di SODDISFACIBILITÀ, e viceversa.

Alla luce di quanto suesposto, verificare se una istanza $(X, \mathcal{F})$ di FALSITÀ è una istanza positiva, corrispondendo a verificare se la stessa istanza è istanza negativa di SODDISFACIBILITÀ, risulta essere inerentemente diverso (e probabilmente più difficile) rispetto al caso in cui si intenda verificare se $(X, \mathcal{F})$ è istanza positiva di SODDISFACIBILITÀ.

9.4 NP-completezza

Definiamo un problema $\mathcal{P} \in$ NP come NP-completo se $\mathcal{P}$ è completo in NP rispetto alla Karp-riducibilità polinomiale, vale a dire che $\mathcal{P}$ è NP-completo se, per ogni $\mathcal{P}_1 \in$ NP, $\mathcal{P}_1 \leq_m^p \mathcal{P}$.

La relazione $\mathcal{P}_1 \leq_m^p \mathcal{P}$ fa sì che saper risolvere $\mathcal{P}$ in tempo polinomiale comporti saper risolvere in tempo polinomiale anche $\mathcal{P}_1$. Infatti, sia $\mathcal{A}$ un algoritmo polinomiale che decide $\mathcal{P}$: allora, dato che per ipotesi deve esistere una Karp-riduzione polinomiale $\mathcal{R}$ da $\mathcal{P}_1$ a $\mathcal{P}$, ogni istanza x di $\mathcal{P}_1$ può essere decisa trasformandola in tempo polinomiale, applicando $\mathcal{R}$, in una istanza $\mathcal{R}(x)$ di $\mathcal{P}$ ed applicando quindi $\mathcal{A}$ a tale istanza.

Inoltre, la Karp-riducibilità polinomiale presenta l'ulteriore proprietà che la classe NP, così come le classi P e PSPACE, è chiusa rispetto ad essa (vedi Esercizio 8.7). Da ciò deriva che, dati due problemi $\mathcal{P}_1$ e $\mathcal{P}_2$, se $\mathcal{P}_1$ è NP–completo e $\mathcal{P}_2 \leq_m^p \mathcal{P}_1$, allora $\mathcal{P}_2 \in$ NP.

I problemi NP-completi possono svolgere un ruolo importante nell'ambito della determinazione della verità o della falsità della relazione P = NP: infatti, dato che ogni problema in NP è per definizione riducibile ad un problema NP-completo, la possibilità di risolvere in tempo polinomiale un problema NP-completo comporterebbe che P = NP.

Così come già osservato per la classe P, il metodo più naturale per mostrare l'NP-completezza di un problema di decisione $\mathcal{P}$ è quello di trovare una Karp riduzione polinomiale da qualche altro problema $\mathcal{P}_1$ che già si sa essere NP-completo. Infatti, dato che per ogni $\mathcal{P}_2 \in$ NP si ha per definizione che $\mathcal{P}_2 \leq_m^p \mathcal{P}_1$, mostrare che $\mathcal{P}_1 \leq_m^p \mathcal{P}$ comporta, per la transitività della Karp-riducibilità polinomiale (vedi Esercizio 8.6), che $\mathcal{P}_2 \leq_m^p \mathcal{P}$, e quindi che $\mathcal{P}$ è NP-completo.

Chiaramente, come già evidenziato nella Sezione 9.1 per la P-completezza,

questo processo richiede un problema NP-completo *iniziale*, la cui completezza deve evidentemente essere dimostrata mediante qualche altra tecnica.

L'importanza del teorema di Cook, presentato sotto, sta proprio nel fatto che tale teorema mostra, per mezzo di una tecnica di riduzione basata sul concetto di *tableau* introdotto nella dimostrazione del Teorema 9.1, che ogni problema in NP è polinomialmente Karp riducibile al problema SODDISFACIBILITÀ.

Teorema 9.4 (Cook) *Il problema* SODDISFACIBILITÀ *è* NP*-completo rispetto alla Karp-riducibilità polinomiale.*

Dimostrazione. Il problema si può facilmente mostrare essere in NP: infatti un algoritmo non deterministico per SODDISFACIBILITÀ viene presentato come Algoritmo 9.2. Come è possibile verificare, l'algoritmo genera mediante

```
input X: insieme di variabili booleane, (c_1, ..., c_n): insieme di clausole su V;
output boolean;
begin
  guess una assegnazione f : X ↦{TRUE,FALSE};
  for each clausola c_i ∈ F do
    if non esiste alcun t_{i,j} ∈ c_i soddisfatto da f
      then return NO;
  return VERO
end.
```

Algoritmo 9.2: Algoritmo non deterministico per SODDISFACIBILITÀ.

una *guess* una assegnazione f, di lunghezza polinomiale, per poi verificare, in tempo polinomiale, che f soddisfa $\mathcal{F}$. Alternativamente, SODDISFACIBILITÀ può essere dimostrato appartenere ad NP semplicemente osservando che l'assegnazione f è un certificato di lunghezza polinomiale e verificabile in tempo polinomiale.

Passiamo ora a mostrare che, per ogni problema $\mathcal{P} \in$ NP, è possibile definire una Karp-riduzione polinomiale da $\mathcal{P}$ a SODDISFACIBILITÀ.

La dimostrazione procede nel modo seguente, in cui per semplicità di esposizione facciamo riferimento al linguaggio L associato a $\mathcal{P}$ ed al linguaggio SAT associato a SODDISFACIBILITÀ.

Dato che $L \in$ NP esiste una macchina di Turing non deterministica $\mathcal{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ che accetta ogni stringa $x \in L$ con $\mid x \mid = n$ in tempo al più $p(n)$, dove p è un opportuno polinomio. Dati $\mathcal{M}$ ed $x \in \Sigma^*$, costruiremo una formula ϕ in forma normale congiuntiva tale che ϕ è soddisfacibile se e solo se $\mathcal{M}$ accetta x in tempo al più $p(n)$, cioè se e solo se $x \in L$.

Al fine di semplificare la dimostrazione, e senza perdere in generalità, facciamo le seguenti ipotesi relativamente ad $\mathcal{M}$:

1. $\mathcal{M}$ ha un solo nastro semi-infinito;

2. all'inizio, il nastro contiene la stringa in input x nelle prime n celle, mentre tutte le altre celle contengono il simbolo ƀ.

Come nel caso della dimostrazione del Lemma 9.5, consideriamo, senza perdita di generalità, la macchina di Turing non deterministica $\mathcal{M}' = \langle \Gamma, \text{ƀ}, Q, q_0, F, \delta' \rangle$, la cui funzione di transizione è definita nel modo seguente:

$$\delta'(q,a) = \begin{cases} \delta(q,a) \text{ se } \delta(q,a) \text{ è definita} \\ \{(q,a,i)\} \text{ altrimenti.} \end{cases}$$

La macchina di Turing $\mathcal{M}'$ ha lo stesso comportamento di $\mathcal{M}$ eccetto che ad ogni computazione massimale di $\mathcal{M}$ corrisponde una computazione di $\mathcal{M}'$ che cicla indefinitamente sulla stessa configurazione. Quindi, ad ogni computazione accettante di $\mathcal{M}$ corrisponde una computazione di $\mathcal{M}'$ che cicla su una configurazione di accettazione.

Ricordiamo inoltre che, dato che ogni computazione accettante ha lunghezza al più $p(n)$, non più di $p(n)+1$ celle del nastro sono coinvolte nella computazione stessa.

Ogni computazione eseguita da $\mathcal{M}'$, in presenza di una specifica *guess*, su input x può essere rappresentata, così come nella dimostrazione del Lemma 9.5, mediante un *tableau* T di dimensione $(p(n)+1) \times (p(n)+1)$, i cui elementi contengono coppie in $\bar{\Gamma} \times (Q \cup \{\perp\})$, con $\perp \notin Q$. Tale tableau deve soddisfare alcune proprietà: anzitutto, ogni riga deve rappresentare una configurazione lecita di $\mathcal{M}'$; inoltre, la prima riga deve rappresentare la configurazione iniziale con x sulle prime n celle del nastro; infine, ogni riga deve rappresentare una configurazione derivata dalla configurazione rappresentata nella riga precedente mediante l'applicazione della funzione di transizione di $\mathcal{M}'$. Si osservi poi che, per quanto detto, la condizione $x \in L$ è vera se e solo se esiste una computazione rappresentata da un tableau la cui ultima riga descrive una configurazione di accettazione.

La formula ϕ che deriviamo rappresenta in modo formale le condizioni che devono essere verificate affinché un tableau rappresenti una computazione di accettazione di x, e si basa sui seguenti predicati, rappresentati nella formula da variabili booleane:

1. $S(i,j,k)$ $(0 \leq i \leq p(n), 0 \leq j \leq p(n), 0 \leq k \leq |\, Q \,|)$, definita come $S(i,j,k)$ se e solo se il secondo componente del contenuto di $T[i,j]$ è $\perp$, se $k = |\, Q \,|$, o q_k, se $k < |\, Q \,|$.

2. $C(i,j,k)$ $(0 \leq i \leq p(n), 0 \leq j \leq p(n), 0 \leq k \leq |\, \Gamma \,|)$, definita come $S(i,j,k)$ se e solo se il primo componente del contenuto di $T[i,j]$ è ƀ, se $k = 0$, o a_k, se $k > 0$.

La formula ϕ è la congiunzione di 4 diverse formule, $\phi = \phi^M \wedge \phi^I \wedge \phi^A \wedge \phi^T$, che codificano le proprietà, sopra illustrate, che devono essere verificate da un tableau che rappresenti una computazione di accettazione. In particolare:

1. $\phi^M = \bigwedge_i \phi_i^M$, dove ϕ_i^M specifica le condizioni affinché la riga i-esima di T rappresenti una configurazione di $\mathcal{M}'$;
2. ϕ^I specifica che la prima riga di T deve rappresentare la configurazione iniziale di $\mathcal{M}'$ con input x;
3. ϕ^A specifica che l'ultima riga di T deve rappresentare una configurazione di accettazione di $\mathcal{M}'$;
4. ϕ^T specifica che ogni riga d T deve rappresentare una configurazione che deriva da quella rappresentata dalla riga precedente applicando la funzione di transizione di $\mathcal{M}$.

Vediamo ora più in dettaglio come le varie sottoformule di ϕ sopra introdotte sono definite.

1. ϕ_i^M è la congiunzione di tre formule $\phi_i^M = \phi_i^{M1} \wedge \phi_i^{M2} \wedge \phi_i^{M3}$. Dove:
   (a) ϕ_i^{M1} specifica che ogni cella $T[i,j]$ deve rappresentare uno ed un solo valore come secondo componente: ciò viene ottenuto esprimendo tale formula come congiunzione $\phi_i^{M1} = \bigwedge_j \left(\phi_{i,j}^{M11} \wedge \phi_{i,j}^{M12}\right)$ di due sottoformule, in cui:
      - $\phi_{i,j}^{M11} = \bigvee_k S(i,j,k)$ specifica che $T[i,j]$ deve rappresentare almeno un valore come secondo componente;
      - $\phi_{i,j}^{M12} = \bigwedge_{k_1 \neq k_2} \left(\overline{S(i,j,k_1)} \vee \overline{S(i,j,k_2)}\right)$ specifica che $T[i,j]$ deve rappresentare al più un valore come secondo componente.
   (b) ϕ_i^{M2} specifica che esattamente una cella $T[i,j]$ deve avere come secondo componente un valore diverso da $|\,Q\,|$: ciò viene ottenuto esprimendo tale formula come congiunzione $\phi_i^{M2} = \phi_i^{M21} \wedge \phi_i^{M22}$ di due sottoformule, in cui:
      - $\phi_i^{M21} = \bigvee_j \overline{S(i,j,|\,Q\,|)}$ specifica che almeno una cella $T[i,j]$ deve avere come secondo componente un valore diverso da $|\,Q\,|$ (corrispondente al simbolo $\perp$);
      - $\phi_i^{M22} = \bigwedge_{j_1 \neq j_2} (S(i,j_1,|\,Q\,|) \vee S(i,j_2,|\,Q\,|))$ specifica che al più una cella $T[i,j]$ deve avere come secondo componente un valore diverso da $|\,Q\,|$.
   (c) ϕ_i^{M3} specifica che ogni cella $T[i,j]$ deve rappresentare uno ed un solo valore come primo componente: ciò viene ottenuto esprimendo tale formula come congiunzione $\phi_i^{M3} = \bigwedge_j \left(\phi_{i,j}^{M31} \wedge \phi_{i,j}^{M32}\right)$ di due sottoformule, in cui:

- $\phi_{i,j}^{M31} = \bigvee_k C(i,j,k)$ specifica che $T[i,j]$ deve rappresentare almeno un valore come primo componente;
- $\phi_{i,j}^{M32} = \bigwedge_{k_1 \neq k_2} \left(\overline{C(i,j,k_1)} \vee \overline{C(i,j,k_2)}\right)$ specifica che $T[i,j]$ deve rappresentare al più un valore come primo componente.

Come si può vedere, ϕ^M è in CNF ed ha lunghezza $O(p(n)^3 \log n)$, in quanto è composta da $O(p(n)^3)$ variabili booleane, ognuna identificata mediante $O(\log n)$ bit. Inoltre, la formula è derivabile in tempo $O(p(n)^3)$.

2. ϕ^I è la congiunzione di due formule $\phi^I = \phi^{I1} \wedge \phi^{I2}$. Dove:
   (a) ϕ^{I1} specifica il contenuto iniziale del nastro di $\mathcal{M}'$, vale a dire i valori delle prime componenti in tutte le celle $T[0,j]$. Se $x = a_{i_0} a_{i_1} \ldots a_{i_{n-1}}$ è la stringa in input, allora si avrà $\phi^{I1} = C(0,0,i_0) \wedge C(0,1,i_1) \wedge \ldots \wedge C(0,n-1,i_{n-1}) \wedge C(0,n,0) \wedge \ldots \wedge C(0,p(n),0)$.
   (b) ϕ^{I2} specifica, attraverso i valori delle seconde componenti in tutte le celle $T[0,j]$, che inizialmente la macchina di Turing $\mathcal{M}'$ si trova nello stato è q_0 e la sua testina è posizionata sulla prima cella del nastro. Si ha quindi $\phi^{I2} = S(0,0,0) \wedge S(0,1,|\,Q\,|) \wedge \ldots \wedge S(0,p(n),|\,Q\,|)$.

   Chiaramente, ϕ^I è in CNF (in effetti è composta da una unica clausola), ha lunghezza $O(p(n) \log n)$ ed è derivabile in tempo $O(p(n))$.

3. ϕ^A è la disgiunzione di $|\,F\,|$ formule booleane, ognuna delle quali specifica la condizione che $\mathcal{M}$ si trovi nello stato $q_k \in F$, con la testina su una delle $p(n)+1$ celle di nastro: $\phi^A = \bigvee_{q_k \in F} \left(\bigvee_j S(0,j,k)\right)$.

   Anche ϕ^A è chiaramente in CNF (è composta da $|\,F\,|\ (p(n)+1)$ clausole composte da un solo predicato), ha lunghezza $O(p(n) \log n)$ ed è derivabile in tempo $O(p(n))$.

4. $\phi^T = \bigwedge_i \phi_i^T$, dove ϕ_i^T specifica le condizioni affinché la riga i-esima di T rappresenti una configurazione di $\mathcal{M}'$ derivata, attraverso l'applicazione della funzione di transizione δ, dalla configurazione rappresentata alla riga $(i-1)$-esima. A sua volta, ogni formula ϕ_i^T ha la struttura $\phi_i^T = \bigwedge_i \phi_{i,j}^T$, dove $\phi_{i,j}^T$ specifica che il contenuto delle tre celle $T[i,j-1]$, $T[i,j]$, $T[i,j+1]$ del tableau deve poter derivare, per mezzo della funzione di transizione, dal contenuto delle celle $T[i-1,j-1]$, $T[i-1,j]$, $T[i-1,j+1]$.[5]

   In particolare, sia $S = (s_0, s_1, \ldots, s_r)$ una qualunque enumerazione dei possibili passi di computazione deterministica definiti nella funzione di

[5] In particolare, se $j = 0$ la formula $\phi_{i,0}^T$ si riferirà alle sole celle $T[i-1,0]$, $T[i-1,1]$, $T[i,0]$, $T[i,1]$, mentre $j = p(n)$ la formula $\phi_{i,p(n)}^T$ si riferirà alle sole celle $T[i-1,p(n)-1], T[i-1,p(n)], T[i,p(n)-1], T[i,p(n)]$.

transizione $\delta_{\mathcal{M}}$ di $\mathcal{M}$, vale a dire che ogni s_i corrisponde ad una 5-pla $(p_i, c_i, p'_i, c'_i, m_i)$, con $p_i, p'_i \in Q$, $c_i, c'_i \in \bar{\Gamma}$, $m_i \in \{d, s, i\}$, e $(p'_i, c'_i, m_i) \in \delta_{\mathcal{M}}(p_i, c_i)$. Ogni formula $\phi^T_{i,j}$ avrà allora la struttura $\phi^T_{i,j} = \phi^{T0}_{i,j} \vee \phi^{T1}_{i,j} \vee \phi^{T2}_{i,j} \vee \phi^{T3}_{i,j}$, dove:

- $\phi^{T0}_{i,j}$ specifica la correttezza della situazione in cui $T[i-1, j-1]$, $T[i-1, j]$, $T[i-1, j+1]$ rappresentano celle di nastro su cui non è posizionata la testina, $T[i, j]$ rappresenta anch'essa una cella su cui non si trova la testina, ed i caratteri contenuti nelle celle rimangono gli stessi nel passare dalla riga $i-1$ alla riga i. Ciò viene realizzato definendo $\phi^{T0}_{i,j}$ nel modo seguente:

$$\begin{aligned}\phi^{T0}_{i,j} = \quad & S(i-1, j-1, |\,Q\,|) \wedge S(i-1, j, |\,Q\,|) \wedge S(i-1, j+1, |\,Q\,|) \wedge \\ & \wedge S(i, j, |\,Q\,|) \wedge \left(\bigvee_{0 \leq k \leq \Gamma} (C(i-1, j, k) \wedge C(i, j, k)) \right).\end{aligned}$$

- $\phi^{T1}_{i,j}$ specifica la correttezza della situazione in cui $T[i-1, j-1]$ rappresenta una cella di nastro sulla quale è posizionata la testina (e quindi si ha $S(i-1, j-1, k)$ per qualche $k <| Q |$): in questo caso, la formula considera tutte le possibili applicazioni della funzione di transizione mediante la disgiunzione $\phi^{T1}_{i,j} = \bigvee_{s_k \in S} \phi^{T1}_{i,j,k}$ dove la formula $\phi^{T1}_{i,j,k}$ enumera tutti i contenuti delle 6 celle considerate che sono correlati mediante l'applicazione del passo di computazione $s_k \in S$.
  Ciò viene realizzato definendo $\phi^{T1}_{i,j,k}$ nel modo seguente. Sia $s_k = (p_k, c_k, p'_k, c'_k, m_k)$, e sia, senza perdita di generalità $p_k = q_{l_1} \in Q$, $p'_k = q_{l_2} \in Q$, $c_k = a_{h_1} \in \bar{\Gamma}$, $c_k = a_{h_2} \in \bar{\Gamma}$: consideriamo separatamente i tre casi in cui m_k è uguale a d, s, o i.
  (a) se $m_k = d$, allora

$$\begin{aligned}\phi^{T1}_{i,j,k} = \quad & \bigvee_{0 \leq t,u \leq \Pi} (S(i-1, j-1, l_1) \wedge C(i-1, j-1, h_1) \wedge \\ & \wedge C(i-1, j, t) \wedge C(i-1, j+1, u) \wedge C(i, j-1, h_2) \wedge \\ & \wedge C(i, j, t) \wedge C(i, j+1, u) \wedge S(i, j, h_2));\end{aligned}$$

  (b) se $m_k = s$, allora

$$\begin{aligned}\phi^{T1}_{i,j,k} = \quad & \bigvee_{0 \leq t,u \leq \Pi} (S(i-1, j-1, l_1) \wedge C(i-1, j-1, h_1) \wedge \\ & \wedge C(i-1, j, t) \wedge C(i-1, j+1, u) \wedge C(i, j-1, h_2) \wedge \\ & \wedge C(i, j, t) \wedge C(i, j+1, u));\end{aligned}$$

  (c) se, infine, $m_k = i$, allora

$$\begin{aligned}\phi^{T1}_{i,j,k} = \quad & \bigvee_{0 \leq t,u \leq \Pi} (S(i-1, j-1, l_1) \wedge C(i-1, j-1, h_1) \wedge \\ & \wedge C(i-1, j, t) \wedge C(i-1, j+1, u) \wedge C(i, j-1, h_2) \wedge \\ & \wedge C(i, j, t) \wedge C(i, j+1, u) \wedge S(i, j-1, h_2)).\end{aligned}$$

- $\phi_{i,j}^{T2}$ e $\phi_{i,j}^{T2}$ specificano la correttezza delle situazioni in cui $T[i-1,j]$ o $T[i-1,j+1]$ rappresentano celle di nastro sulla quale è posizionata la testina. La loro struttura è simile a quella descritta per $\phi_{i,j}^{T1}$, e viene lasciata per esercizio (vedi Esercizio 9.6).

Si osservi che ogni formula $\phi_{i,j}^{T}$ è composta da una quantità costante di predicati C ed S, ognuno dei quali ha dimensione $O(\log n)$. Pur non essendo in CNF, ogni formula può poi essere facilmente trasformata in una formula $\overline{\phi}_{i,j}^{T}$ in CNF equivalente, anch'essa composta da $O(1)$ predicati, e quindi anch'essa di dimensione $O(\log n)$.

Da ciò deriva che l'intera formula ϕ^T ha dimensione $O((p(n))^2 \log n)$ ed è derivabile in tempo $O((p(n))^2)$.

Se si considera ora l'intera formula ϕ, è possibile verificare che la componente dominante nella sua dimensione è quella relativa alla dimensione di ϕ_M, dal che risulta che la dimensione di ϕ è $O(p(n)^3 \log n)$, e l'intera formula può essere derivata in tempo $O(p(n)^3)$.

Lasciamo per esercizio (vedi Esercizio 9.7) la semplice verifica che la formula ϕ è soddisfacibile se e solo se esiste una computazione accettante di $\mathcal{M}$. □

Esercizio 9.6 Individuare la struttura delle formule $\phi_{i,j}^{T2}$ e $\phi_{i,j}^{T2}$ nella dimostrazione del teorema precedente.

Esercizio 9.7 Verificare che, nella dimostrazione del teorema precedente, la formula ϕ è soddisfacibile, dato un certo input x, se e solo se esiste una computazione accettante di $\mathcal{M}$ a partire da tale input.

Una volta individuato, mediante il teorema di Cook, un primo problema NP-completo è possibile ora fare riferimento alla tecnica standard per provare che un problema $\mathcal{P}$ è NP-completo. Tale tecnica, riassumiamo, è basata su due passi:

1. dimostrare che $\mathcal{P} \in$ NP;
2. esiste un problema $\mathcal{P}'$ NP-completo, mostrare che $\mathcal{P}' \leq_m^p \mathcal{P}$, fornendo una procedura che, in tempo polinomiale, trasforma ogni istanza di $\mathcal{P}'$ in una istanza equivalente di $\mathcal{P}$.

Come primo esempio di dimostrazione "standard" di NP-completezza, mostriamo che il problema 3-SODDISFACIBILITÀ, ottenuto da SODDISFACIBILITÀ considerando nella sua definizione soltanto clausole aventi 3 termini, è NP-completo.

3-SODDISFACIBILITÀ

ISTANZA: Una formula CNF $\mathcal{F}$ su un insieme X di variabili booleane, con ogni clausola composta da al più 3 termini.

PREDICATO: Esiste una assegnazione $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$ che soddisfa $\mathcal{F}$?

Teorema 9.5 *Il problema* 3-SODDISFACIBILITÀ *è* NP*-completo.*

Dimostrazione. Mostriamo inizialmente che 3-SODDISFACIBILITÀ$\in$ NP: infatti, dato $f : X \mapsto \{\text{VERO}, \text{FALSO}\}$ è chiaramente un certificato di lunghezza polinomiale verificabile in tempo polinomiale.

Al fine di provare che 3-SODDISFACIBILITÀ$\leq_m^p$SODDISFACIBILITÀ mostriamo dapprima una possibile riduzione da istanze di SODDISFACIBILITÀ ad istanze di 3-SODDISFACIBILITÀ.

Consideriamo una istanza generica X, C di 3-SODDISFACIBILITÀ, dove $X = \{x_1, x_2, \ldots, x_n\}$ è un insieme di variabili booleane e $C = \{c_1, c_2, \ldots, c_m\}$ è un insieme di clausole su X. Da X, C otterremo una istanza X', C' di 3-SODDISFACIBILITÀ in cui da ogni clausola $c_i \in C$ vengono derivati un insieme di clausole a tre termini C'_i definite su un insieme di variabili X'_i: si avrà che l'istanza di 3-SODDISFACIBILITÀ risultante avrà $X' = X \cup (\cup_{i=1}^m X'_i)$ $C' = \cup_{i=1}^m C'_i$.

Tale corrispondenza dipenderà dal numero $|\, c_i \,|$ di termini nella clausola c_i:

1. $|\ c_i\ | = 1$. Sia $c_i = \{z\}$: allora, $X'_i = \{y_i^1, y_i^2\}$ e $C'_i = \{\{z_1, y_i^1, y_i^2\}, \{z_1, \overline{y}_i^1, y_i^2\}, \{z_1, y_i^1, \overline{y}_i^2\}, \{z_1, \overline{y}_i^1, \overline{y}_i^2\}\}$. Si noti che una assegnazione che soddisfa c_i (che cioè rende vero il termine z), comunque esteso a piacere a y_i^1, y_i^2 soddisfa anche C'_i; inoltre, una assegnazione che soddisfa C'_i dovrà necessariamente rendere z vero (altrimenti una delle quattro clausole non può essere soddisfatta) e, quindi, soddisfa anche c_i. Da ciò possiamo concludere che l'insieme di clausole in C'_i è soddisfatto da tutti e soli gli assegnamenti (comunque estesi) tali che $z = \text{VERO}$, gli assegnamenti cioè che soddisfano c_i.

2. $|\, c_i \,| = 2$. Sia $c_i = \{z, v\}$: allora, $X'_i = \{y_i^1\}$ e $C'_i = \{\{z, v, y_i^1\}, \{z, v, \overline{y}_i^1\}\}$. Si noti che una assegnazione che soddisfa c_i (che cioè rende vero uno almeno tra i termini z e v), comunque esteso a piacere a y_i^1 soddisfa anche C'_i; inoltre, una assegnazione che soddisfa C'_i dovrà necessariamente rendere almeno un termine tra z e v vero (altrimenti una delle due clausole non può essere soddisfatta) e, quindi, soddisfa anche c_i. In questo caso, le due clausole in C'_i sono soddisfatte da tutti e soli gli assegnamenti (comunque estesi) tali che $z = \text{VERO}$ o $v = \text{FALSO}$, e quindi da tutti e soli gli assegnamenti che soddisfano c_i.

3. $|\, c_i \,| = 3$. Sia $c_i = \{z, v, w\}$: dato che in questo caso c_i ha esattamente i tre termini richiesti nella definizione di 3-SODDISFACIBILITÀ, $X'_i = \emptyset$ e $C'_i = \{\{z, v, w\}\}$. Chiaramente, C'_i è soddisfatta da tutti e soli gli assegnamenti che soddisfano c_i.

4. $|\, c_i \,| = k \geq 4$. Sia $c_i = \{z_1, z_2, \ldots, z_k\}$: allora avremo $X'_i = \{y_i^j,\ 1 \leq j \leq k-3\}$ e $C'_i = \{\{z_1, z_2, y_i^1\},\ \{\overline{y}_i^1, z_3, y_i^2\},\ \{\overline{y}_i^2, z_4, y_i^2\},\ \ldots,\ \{\overline{y}_i^{k-4}, z_{k-2}, y_i^{k-3}\},\ \{\overline{y}_i^{k-3}, z_{k-1}, z_k\}\}$. In questo caso, si noti che una assegnazione f che soddisfa c_i dovrà rendere almeno un termine in $\{z_i \ldots, z_k\}$ vero: sia z_r il primo di tali termini (siano cioè tutti i termini z_j con $j < i$ posti a falso dall'assegnazione), allora, l'estensione f' di f in cui $y_i^1 =$ VERO, $y_i^2 =$ VERO, $\ldots$, $y_i^{r-2} =$ VERO, $y_i^{r-1} =$ VERO, $\ldots, y_i^{k-3} =$ FALSO soddisfa l'insieme di clausole C'_i. Inoltre, si osservi che una assegnazione f' che soddisfa C'_i dovrà necessariamente porre a vero almeno un termine z_r (altrimenti una delle clausole non viene soddisfatta, indipendentemente dai valori assegnati alle variabili y_i^j): evidentemente, tale assegnazione soddisferà anche c_i.

Da quanto detto, si può concludere che (1) se esiste una assegnazione f su X che soddisfa C, esso può essere esteso, come illustrato sopra, ad ottenere una assegnazione f' su X' che soddisfa C' e (2) se esiste una assegnazione f' su X' che soddisfa C', la stessa assegnazione soddisfa anche C.

Da ciò evidentemente deriva che C è soddisfacibile se e solo se lo è C'. Si osservi infine che la trasformazione può chiaramente essere effettuata in tempo polinomiale, e precisamente $O(nm)$. □

Come altro esempio di dimostrazione di NP-completezza, mostriamo ora che il seguente problema COPERTURA CON NODI è NP-completo.

COPERTURA CON NODI
ISTANZA: Grafo $G = (V, E)$, intero $K > 0$
PREDICATO: Esiste un sottoinsieme $V' \subseteq V$ tale che $\|V'\| \leq K$ e $\forall (u, v) \in E$ $u \in V'$ o $v \in V'$?

Teorema 9.6 *Il problema* COPERTURA CON NODI *è* NP*-completo.*

Dimostrazione. Mostriamo che COPERTURA CON NODI appartiene ad NP: infatti, dato un sottoinsieme $V' \subseteq V$ tale che $|\, V' \,| \leq K$ e $\forall (u, v) \in E$ $u \in V'$ o $v \in V'$ è chiaramente un certificato di lunghezza polinomiale, verificabile in tempo polinomiale.

Per provare la completezza di COPERTURA CON NODI in NP, mostriamo che 3-SODDISFACIBILITÀ$\leq_m^p$COPERTURA CON NODI. Sia data una istanza generica (X, C) di 3-SODDISFACIBILITÀ: a partire da (X, C) deriviamo una

grafo $G = (V, E)$ ed un intero $K > 0$ tali che G ha una copertura con nodi di dimensione non superiore a K se e solo se l'insieme di clausole C è soddisfacibile.

L'insieme V dei nodi di G è definito come segue:

- ad ogni variabile $x_i \in X$ sono associati due nodi $x_i, \overline{x}_i \in V$;
- ad ogni clausola $c_j \in C$ sono associati tre nodi $c_j^1, c_j^2, c_j^3 \in V$.

Per quanto riguarda l'insieme E degli archi di G (vedi Figura 9.8):

- esiste un arco $(x_i, \overline{x}_i) \in E$ per ogni $x_i \in X$;
- per ogni clausola $c_j = \{t_j^1, t_j^2, t_j^3\} \in C$, introduciamo tre archi (c_j^1, c_j^2), (c_j^2, c_j^3), (c_j^3, c_j^1). Inoltre, introduciamo un arco per ogni termine t_j^s ($s = 1, 2, 3$) in c_j nel modo seguente: sia $t_j^s = x_k \in X$: introduciamo allora un arco (c_j^s, x_k) se $t_j^s = x_k$ o un arco $(c_j^s, \overline{x}_k)$ se $t_j^s = \overline{x}_k$.

Si osservi che $\mid V \mid = 2 \mid X \mid + 3 \mid C \mid$ e che $\mid E \mid = \mid X \mid + 6 \mid C \mid$, e che la trasformazione può essere effettuata in tempo polinomiale. Poniamo inoltre $K = \mid X \mid + 2 \mid C \mid$. Ad una prima analisi, si può verificare che non esistono coperture di nodi su G di taglia inferiore a K: infatti, per ognuno degli $\mid X \mid$ archi del tipo $(x_i, \overline{x}_i)$ almeno uno tra i due nodi estremi dovrà essere inserito in una copertura, mentre per ognuna delle $\mid C \mid$ componenti c_j^1, c_j^2, c_j^3 almeno due dei tre nodi devono a loro volta entrare a far parte della copertura stessa.

Mostriamo ora che se (X, C) è soddisfacibile allora esiste una copertura V' di taglia K su G. Sia f una assegnazione che soddisfa (X, C): allora, per ogni variabile x_i, se $f(x_i) =$ VERO inseriamo in V' il nodo x_i, altrimenti inseriamo il nodo $\overline{x}_i$. Inoltre, per ogni clausola $c_j = \{t_j^1, t_j^2, t_j^3\}$, sia $t_j^s \in c_j$ un termine, che deve necessariamente esistere, tale che $f(t_j^s) =$ VERO: inseriamo allora in V' i due nodi $\{c_j^r \mid r \neq s\}$. È immediato verificare che tale insieme di K nodi rappresenta effettivamente una copertura di nodi di G.

Per quanto riguarda l'implicazione inversa, sia V' una copertura dei nodi di G con $\mid V' \mid \leq K$. Per quanto osservato sopra, dovrà essere necessariamente $\mid V' \mid = K$: in effetti, V' dovrà necessariamente includere 1 nodo per ogni arco del tipo $(x_i, \overline{x}_i)$ e due nodi per ogni componente c_j^1, c_j^2, c_j^3. Deriviamo una assegnazione f che soddisfa (X, C) nel modo seguente: per ogni $x_i \in X$, $f(x_i) =$ VERO se $x_i \in V'$, altrimenti $f(x_i) =$ FALSO. Tale assegnazione soddisferà (X, C) in quanto, per ogni clausola c_j, la corrispondente componente c_j^1, c_j^2, c_j^3 in G avrà esattamente un nodo $c_j^s \in V - V'$: tale nodo necessariamente sarà collegato ad un nodo associato al termine t_j^s, appartenente necessariamente a V'. Dato che è stato posto $f(t_j^s) =$ VERO, ne deriva che c_j è soddisfatta. □

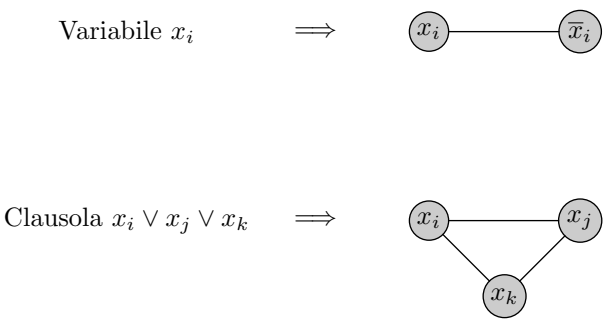


FIGURA 9.8 Riduzione da 3-SODDISFACIBILITÀ a COPERTURA CON NODI.

Esercizio 9.8 Dimostrare che il problema CRICCA introdotto nella Sezione 8.2 è NP-completo.
[Suggerimento: Definire una Karp-riduzione polinomiale da COPERTURA CON NODI.]

Introduciamo infine il seguente problema.

COPERTURA CON INSIEMI

ISTANZA: Insieme S, collezione $\mathcal{C} = \{s_1, s_2, \ldots, s_n\}$ di sottoinsiemi di S, intero $K > 0$.

PREDICATO: Esiste un sottoinsieme $\mathcal{C}'$ di $\mathcal{C}$ che copre S, vale a dire tale che $\bigcup_{s_i \in \mathcal{C}'} = S$, avente cardinalità $|\mathcal{C}'| \leq K$?

Esercizio 9.9 Dimostrare che il problema COPERTURA CON INSIEMI è NP-completo.
[Suggerimento: Definire una Karp-riduzione polinomiale da COPERTURA CON NODI.]

Osserviamo ora che in generale, dato un problema $\mathcal{P}$, se restringiamo il problema stesso ad un sottoinsieme delle possibili istanze otteniamo un suo *sottoproblema* $\mathcal{P}'$ (come nel caso di 3-SODDISFACIBILITÀ rispetto a SODDISFACIBILITÀ) che certamente non è più difficile da risolvere rispetto a $\mathcal{P}$: in effetti vale banalmente la relazione $\mathcal{P}' \leq_r \mathcal{P}$ per qualunque riducibilità $\leq_r$, in quanto ogni istanza di $\mathcal{P}'$ è istanza di $\mathcal{P}$.

È interessante però in tale situazione chiedersi se la restrizione introdotta sull'insieme delle istanze comporta una maggiore facilità di risoluzione del problema. Ci chiediamo cioè se $\mathcal{P} \leq_r \mathcal{P}'$: se tale relazione è vera allora, almeno rispetto alla riducibilità considerata, $\mathcal{P}$ e $\mathcal{P}'$ sono equivalenti mentre,

altrimenti, $\mathcal{P}$ è effettivamente più difficile di $\mathcal{P}'$. Ciò risulta particolarmente interessante nel caso in cui $\mathcal{P}$ sia NP-completo, in quanto ci chiediamo se il sottoproblema $\mathcal{P}'$ considerato è ancora NP-completo o meno.

Ad esempio, nel caso di SODDISFACIBILITÀ e di 3-SODDISFACIBILITÀ abbiamo già visto che la restrizione introdotta dal considerare soltanto istanze con clausole di esattamente 3 termini non rende il problema più facile, rispetto alla Karp-riducibilità polinomiale, in quanto 3-SODDISFACIBILITÀ è anch'esso NP-completo e quindi SODDISFACIBILITÀ$\leq_m^p$3-SODDISFACIBILITÀ.

Esistono però delle diverse restrizioni di SODDISFACIBILITÀ per le quali si ha un effettivo miglioramento della relativa difficoltà di soluzione. Una di tali restrizioni è quella relativa a SODDISFACIBILITÀ DI FORMULE DI HORN: chiaramente SODDISFACIBILITÀ DI FORMULE DI HORN$\leq_m^p$SODDISFACIBILITÀ (e quindi SODDISFACIBILITÀ DI FORMULE DI HORN$\in$ NP), mentre non si ha SODDISFACIBILITÀ$\leq_m^p$SODDISFACIBILITÀ DI FORMULE DI HORN, a meno che LOGSPACE = P = NP.

Un secondo sottoproblema di SODDISFACIBILITÀ che è sostanzialmente più semplice di esso è 2-SODDISFACIBILITÀ, vale a dire la restrizione al caso in cui ogni clausola è composta da al più 2 termini. Anche in questo caso è possibile mostrare che 2-SODDISFACIBILITÀ$\in$ P mostrando che esiste un algoritmo che risolve tale problema in tempo polinomiale (vedi Esercizio 9.10): ne deriva quindi che non si ha SODDISFACIBILITÀ$\leq_m^p$2-SODDISFACIBILITÀ, a meno che P = NP.

Esercizio 9.10 Trovare un algoritmo polinomiale che risolva 2-SODDISFACIBILITÀ.

In generale, possiamo considerare la relazione $<_R$ di restrizione di un problema in un suo sottoproblema, definita come $\mathcal{P} <_R \mathcal{P}'$ se e solo se $\mathcal{P}'$ è una restrizione di $\mathcal{P}$, vale a dire le sue istanze sono un sottoinsieme delle istanze di $\mathcal{P}$: come si può immediatamente verificare, tale relazione è una relazione di ordine parziale. Dato un insieme di problemi in NP, possiamo associare ad esso un grafo orientato aciclico i cui nodi corrispondono ai problemi e si ha un arco dal nodo u al nodo v se e solo se, detti $\mathcal{P}_u$ e $\mathcal{P}_v$ i problemi associati a u e v, si ha che $\mathcal{P}_u <_R \mathcal{P}_v$ e che non esiste alcun problema $\mathcal{P}_w$ tale che $\mathcal{P}_u <_R \mathcal{P}_w <_R \mathcal{P}_v$.

Tale grafo può allora essere decomposto in tre regioni (vedi Figura 9.9):

1. una regione "superiore", di tutti problemi NP-completi;
2. una regione "inferiore", di problemi in P;
3. una regione "intermedia", di problemi per i quali non si dispone di dimostrazioni né di NP-completezza, né di polinomialità.

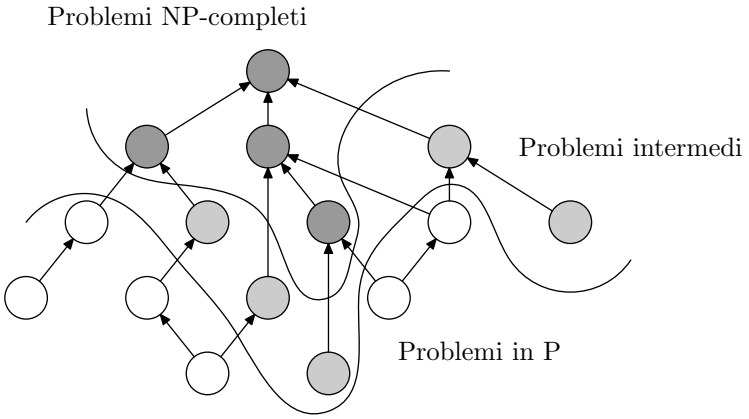


FIGURA 9.9 Relazione di restrizione fra problemi in NP.

9.5 Ancora sulla classe NP

9.5.1 La classe CO-NP

La relazione tra FALSITÀ e SODDISFACIBILITÀ osservata nella Sezione 9.3, è un tipico esempio della relazione generale di complementarietà tra problemi di decisione, definita qui sotto.

Definizione 9.9 *Dato un problema di decisione $\mathcal{P}$, il problema complementare co-$\mathcal{P}$ è un problema di decisione tale che $I_{co\text{-}\mathcal{P}} = I_{\mathcal{P}}$, $Y_{co\text{-}\mathcal{P}} = N_{\mathcal{P}}$ e $N_{co\text{-}\mathcal{P}} = Y_{\mathcal{P}}$.*

Per ogni classe di complessità $\mathcal{C}$ possiamo quindi definire la corrispondente classe complementare co-$\mathcal{C}$ come la classe contenente tutti e soli i problemi complementari a problemi in $\mathcal{C}$, co-$\mathcal{C} = \{\mathcal{P} : \text{co-}\mathcal{P} \in \mathcal{C}\}$. Coerentemente a ciò, indichiamo con CO-NP la classe complementare di NP.

Si osservi che, dato che risulta immediato che P = CO-P (in quanto per risolvere un problema complementare di un problema $\mathcal{P} \in$ P basta cambiare l'algoritmo per $\mathcal{P}$ in modo da scambiare risposte VERO con risposte FALSO), allora se NP $\neq$ CO-NP si ha che P $\neq$ NP, ma si badi che non è vero il contrario, in quanto NP = CO-NP non implica necessariamente che P = NP.

Non è difficile rendersi conto che se $\mathcal{P}$ è NP–completo, allora co-$\mathcal{P}$ è CO-NP–completo. Infatti:

1. per definizione, $\mathcal{P} \in$ NP implica co-$\mathcal{P} \in$ CO-NP (e viceversa);

2. dato che per ogni $\mathcal{P}_1 \in$ NP vale la proprietà $\mathcal{P}_1 \leq_m^p \mathcal{P}$, si ha che co-$\mathcal{P}_1 \in$ CO-NP e che co-$\mathcal{P}_1 \leq_m^p$ co-$\mathcal{P}$.

Possiamo inoltre dimostrare i seguenti teoremi. Il primo di tali teoremi ci mostra come la proprietà fondamentale dei problemi NP-completi, per la quale se un problema NP-completo è anche polinomiale allora P = NP, è valida anche per i problemi co-NP-completi.

Teorema 9.7 *Se $\mathcal{P}$ è* co-NP*–completo, allora $\mathcal{P} \in$ P implica* P = NP*.*

Dimostrazione. Dato che $\mathcal{P}$ è co-NP–completo, allora co-$\mathcal{P}$ è NP–completo. Inoltre, dato che $\mathcal{P} \in$ P, ne segue che co-$\mathcal{P} \in$ co-P = P. Per ogni $\mathcal{P}_1 \in$ NP abbiamo quindi che $\mathcal{P}_1 \leq_m^p$ co-$\mathcal{P} \in$ P, da cui deriva che $\mathcal{P}_1 \in$ P. □

I due teoremi seguenti ci mostrano una proprietà più debole della precedente, per la quale se un problema NP-completo appartiene a co-NP (o, simmetricamente, se un problema co-NP-completo appartiene a NP) allora vale la condizione NP=co-NP. Si osservi che in generale si ritiene che, così come P ≠ NP, si abbia anche NP ≠ co-NP.

Teorema 9.8 *Se un problema $\mathcal{P}$ è* co-NP*–completo, allora $\mathcal{P} \in$ NP comporta* NP = co-NP*.*

Dimostrazione. Per ipotesi, dato un qualunque $\mathcal{P}_1 \in$ co-NP, avremo $\mathcal{P}_1 \leq_m^p \mathcal{P} \in$ NP. Ne consegue, per la chiusura della classe NP rispetto alla Karp-riducibilità polinomiale (vedi Esercizio 8.7), che $\mathcal{P}_1 \in$ NP, e quindi, in generale, che co-NP ⊆ NP.

D'altro canto, in modo simmetrico, per ogni $\mathcal{P}_2 \in$ NP avremo che $\mathcal{P}_2 \leq_m^p$ co-$\mathcal{P} \in$ co-NP. Di conseguenza, per la chiusura della classe co-NP rispetto alla Karp-riducibilità polinomiale (vedi Esercizio 9.11), si avrà che $\mathcal{P}_2 \in$ co-NP, e quindi, in generale, che NP ⊆ co-NP, il che, in conseguenza delle considerazioni precedenti, comporta NP = co-NP. □

Esercizio 9.11 Dimostrare che la classe co-NP è chiusa rispetto alla Karp riduzione polinomiale.

Per simmetria con il Teorema 9.8, vale evidentemente anche il seguente teorema.

Teorema 9.9 *Se un problema $\mathcal{P}$ è* NP*–completo, allora $\mathcal{P} \in$ co-NP implica* NP = co-NP*.*

Dimostrazione. La dimostrazione deriva immediatamente, nella sua struttura, da quella del Teorema 9.8 □

In Figura 9.10 è riportata la struttura dele classi NP e co-NP nel caso in cui esse non coincidano.

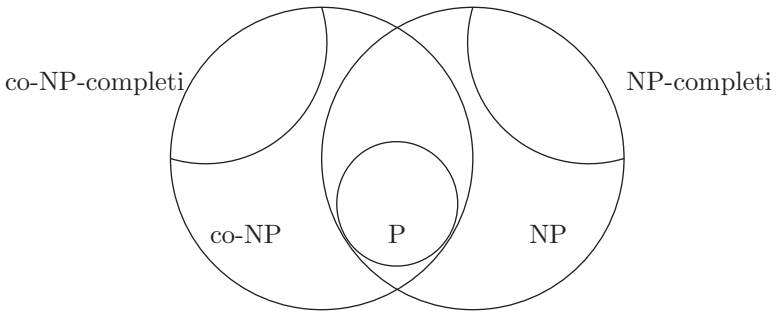


FIGURA 9.10 Struttura di NP e CO-NP se NP $\neq$ CO-NP.

9.5.2 Problemi intermedi in NP

Come osservato sopra, un problema si può trovare nella regione intermedia del grafo in Figura 9.9 in quanto non sono disponibili algoritmi polinomiali per la sua soluzione, né riduzioni da problemi NP-completi che ne mostrino la NP-completezza. Esiste anche la possibilità che il problema in questione sia inerentemente intermedio, vale a dire che, pur non essendo NP-completo, esso non sia in P. L'esistenza di problemi di questo tipo è stata dimostrata dal seguente teorema, che enunciamo senza dimostrazione.

Teorema 9.10 (Ladner) *Se* P $\neq$ NP *allora per ogni* $k \geq 1$ *esiste una sequenza di linguaggi* $L_1, L_2, \ldots, L_k$ *tali che:*

$$L_P \leq_m^p L_1 \leq_m^p L_2 \leq_m^p \ldots \leq_m^p L_k \leq_m^p L_{NPC},$$

dove $L_P \in$ P *e* L_{NPC} *è* NP*-completo.*

Tuttavia, non sono noti al momento problemi "naturali" che siano intermedi in NP.

Due importanti problemi di decisione che sono dei buoni candidati ad essere inerentemente intermedi in NP sono ISOMORFISMO TRA GRAFI e NUMERO PRIMO, definiti nel modo seguente:

ISOMORFISMO TRA GRAFI

ISTANZA: Grafi $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$.

PREDICATO: I due grafi sono isomorfi, esiste cioè una funzione biiettiva $f : V_1 \mapsto V_2$ tale che, per ogni coppia di nodi $u, v \in V_1$, $(u, v) \in E_1$ se e solo se $(f(u), f(v)) \in E_2$?

NUMERO PRIMO
ISTANZA: Intero $n > 0$.
PREDICATO: L'intero n è primo?

Mentre è immediato mostrare che ISOMORFISMO TRA GRAFI $\in$ NP, in quanto una qualunque funzione biiettiva da V_1 a V_2 può essere descritta in spazio polinomiale e la relativa verifica richiede tempo polinomiale, lo stesso non è vero per NUMERO PRIMO. In effetti, l'individuazione di un possibile modo di costruire certificati di dimensione polinomiale e verificabili in tempo polinomiale associati alle istanze positive di tale problema non è immediata e richiede l'uso del seguente risultato di teoria dei numeri.

Teorema 9.11 *Un intero $n > 1$ è primo se e solo se esiste un intero r tale che:*

1. $r^{n-1} \equiv 1$ *(mod n);*
2. *per ogni numero primo divisore di $n-1$ si ha $r^{(n-1)/q} \not\equiv 1$ (mod n).*

È possibile quindi introdurre un certificato di primalità di un intero n come un vettore $c(n) = (r, q_1, c(q_1), q_2, c(q_2, \ldots, q_k, c(q_k)))$, dove $q_1, \ldots, q_k$ sono i divisori primi di $n-1$ e $c(q_1), \ldots, c(q_k)$ sono i relativi certificati di primalità (aventi la medesima struttura). Verificare tale certificato richiederà verificare le due proprietà sopra enunciate, oltre che accertarsi che ogni q_i divide $n-1$, che $q_1, \ldots, q_k$ sono tutti i divisori di $n-1$, e che ogni q_i a sua volta è primo, verificando ricorsivamente il certificato $c(q_i)$.

Esercizio 9.12 Dimostrare che il certificato di primalità introdotto sopra ha lunghezza polinomiale (in $\log n$) ed è verificabile in tempo polinomiale.
[Suggerimento: Ricordare che il numero di divisori primi di un intero p è minore di $\log p$.]

Si consideri ora il problema complementare di NUMERO PRIMO, definito nel modo seguente.

NUMERO COMPOSTO
ISTANZA: Intero $n > 0$.
PREDICATO: L'intero n è composto?

Chiaramente, NUMERO COMPOSTO $\in$ NP, come si può concludere osservando che, data una istanza positiva di tale problema (un intero non primo n), un qualunque intero $m < n$ è un certificato di lunghezza polinomiale e verificabile in tempo polinomiale, determinando se m divide n. Dall'appartenenza di NUMERO COMPOSTO a NP deriva evidentemente che NUMERO PRIMO

$\in$ NP $\cap$ co-NP e quindi, per il Teorema 9.9, se Numero primo fosse NP-completo, allora NP = co-NP.

Un ulteriore problema correlato ai precedenti, di grande importanza sia teorica che per tutte le applicazioni legate alla sicurezza ed alla crittografia, che può essere dimostrato appartenere a NP $\cap$ co-NP è il seguente problema Fattorizzazione.

Fattorizzazione
Istanza: Interi $n, k > 0$.
Predicato: L'intero n ha un divisore $d \leq k$?

Il problema Fattorizzazione può essere dimostrato appartenere a NP osservando che un suo certificato è dato da un intero $d \leq k$ che divide n: chiaramente, il certificato ha lunghezza polinomiale ed è verificabile in tempo polinomiale, determinando che d sia effettivamente minore o uguale a k e che divida n.

L'appartenenza di Fattorizzazione a co-NP deriva osservando che un certificato del problema complementare, vale a dire che un certificato che n non ha divisori minori o uguali a k è dato dalla decomposizione di n in fattori primi $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$, dove e_i è la molteplicità del fattore primo p_i nel decomposizione di n e dove per ogni fattore p_i si ha $p_i > k$, corredata dall'insieme dei certificati $c(p_1), \dots, c(p_k)$ di primalità dei fattori di n.

9.5.3 Problemi fortemente NP*-completi e pseudopolinomialità*

È interessante osservare che, nell'ambito dei problemi NP-completi, esistono problemi in un certo senso "strutturalmente difficili". Tali problemi hanno la caratteristica di rimanere NP-completi indipendentemente dai valori numerici eventualmente presenti nelle istanze. Consideriamo ad esempio il seguente problema Taglio.

Taglio
Istanza: Grafo $G = (V, E)$, funzione peso $w : E \mapsto \mathbb{N}, K \in \mathbb{N}$
Predicato: Esiste una partizione (V_1, V_2) di V tale che $\sum_{\substack{u \in V_1 \\ v \in V_2}} w(u, v) \geq K$?

Si può facilmente mostrare che questo problema è NP–completo. Inoltre, è possibile mostrare che Taglio rimane NP–completo anche se ristretto al caso in cui tutti gli archi in G hanno lo stesso peso, unitario. Da questo possiamo osservare che Taglio è un problema computazionalmente difficile (se P $\neq$ NP) indipendentemente dai valori dei pesi associati agli archi.

Consideriamo ora un problema NP–completo diverso, come il problema Bisaccia.

BISACCIA

ISTANZA: Insieme $I = \{e_1, e_2, \ldots, e_n\}$, funzione costo $s : I \mapsto \mathbb{N}$, funzione profitto $p : I \mapsto \mathbb{N}$, $B, K \in \mathbb{N}$

PREDICATO: Esiste un sottoinsieme $I_1 \subseteq I$ tale che $\sum_{e \in I_1} s(e) \leq B$ e $\sum_{e \in I_1} p(e) \geq K$?

Questo problema può essere risolto, come vedremo, in tempo $O(Bn)$ nel modo seguente, utilizzando una tecnica algoritmica nota come *programmazione dinamica*.

Data una istanza $\kappa(n) = \langle I, s, p, B, K \rangle$ di BISACCIA, per ogni $j \leq n$ indichiamo con $\kappa(j) = \langle I', s', p', B, K \rangle$ l'istanza (più semplice) tale che $I' = \{e_1, e_2, \ldots, e_j\}$ e le funzioni s', p' sono le restrizioni di s, p al dominio $I' \subseteq I$.

Un algoritmo di programmazione dinamica per $\kappa(n)$ consiste di n iterazioni, associate alle istanze $\kappa(1), \kappa(2), \ldots, \kappa(n)$. Per ogni istanza $\kappa(j)$ $(j > 1)$, l'algoritmo costruisce un insieme $S(j)$ di al più $B + 1$ soluzioni a partire dal corrispondente insieme $S(j-1)$, relativo all'istanza $\kappa(j-1)$.

L'insieme iniziale $S(1)$ viene definito come $S(1) = \{\{e_1\}, \emptyset\}$: cioè in $S(1)$ vengono incluse le due soluzioni rappresentate dal solo elemento e_1 e dall'insieme vuoto.

Ad ogni iterazione, l'insieme $S(j)$ viene derivato dall'insieme $S(j-1)$ considerando, per ogni soluzione $s \in S(j-1)$, le due soluzioni $s_1 = s$ e $s_2 = s \bigcup \{e_j\}$. Ciò determina la generazione di $2\,|S(j-1)|$ soluzioni.

Consideriamo ora il seguente *criterio di dominanza*, che potremo utilizzare per ridurre l'insieme delle soluzioni da mantenere: se s_1, s_2 sono due soluzioni tali che $\sum_{e \in s_1} s(e) \leq \sum_{e \in s_2} s(e)$ e $\sum_{e \in s_1} p(e) \geq \sum_{e \in s_2} p(e)$, allora possiamo eliminare s_2 da $S(j)$.

L'applicazione del criterio di dominanza consente di mantenere, per ogni possibile dimensione $1 \leq i \leq B$, soltanto al più una soluzione di massimo profitto.

Da quanto detto, deriva che, per ogni $1 \leq j \leq n$, l'algoritmo genera un insieme di al più $B + 1$ soluzioni diverse. È facile rendersi conto che la soluzione di massimo profitto generata comparirà necessariamente in $S(n)$, e sarà quindi individuabile mediante una scansione sequenziale di tale insieme. A questo punto, sarà sufficiente confrontare il profitto di tale soluzione con K per determinare se l'istanza è positiva o negativa.

È possibile visualizzare l'algoritmo di programmazione dinamica in termini di riempimento di una tabella T di n righe e $B+1$ colonne, le cui celle possono contenere o un insieme $I' \subseteq I$ o il simbolo "$-$". In particolare, se $T(j, i) = I'$ allora I' è un sottoinsieme di massimo profitto di $S(j)$ tale che $\sum_{e \in I'} s(e) = i$; se non esiste alcun $I' \subseteq I$ di dimensione i, allora $T(j, i) =$ "$-$". L'Algoritmo 9.3 riempie una tabella di $n(B + 1)$ celle utilizzando tempo costante per ognuna, da cui deriva che la sua complessità è $O(nB)$.

Esempio 9.3 Consideriamo l'istanza di BISACCIA in cui:

```
input I: insieme, s funzione taglia, p funzione profitto, B, K: naturali;
output boolean;
begin
  for j := 1 to n do
    for i := 0 to B do T(j,i) := "-";
  s := s(e_1);
  T(1,0) := ∅;
  T(1,s) := {e_1};
  for j := 2 to n do begin
    s := s(e_j);
    for i := 0 to B do begin
      if T(j-1,i) ≠ "-" then begin
        if T(j,i)= "-" then T(j,i) := T(j-1,i)
          else Inserisci in T(j,i) la soluzione di massimo profitto
            tra T(j-1,i) e T(j,i);
        if i + s ≤ B
          then if T(j,i+s) = "-"
                 then T(j,i+s) := T(j-1,i) ∪ {e_j}
                 else Inserisci in T(j,i+s) la soluzione di massimo profitto
                   tra T(j-1,i) ∪ {e_j} e T(j,i+s);
      end
    end
  end;
  Sia T(n,k) la soluzione di massimo profitto nella riga n-ma di T;
  Sia p̄ il corrispondente profitto;
  if p̄ ≥ K then return VERO
    else return FALSO;
end.
```

Algoritmo 9.3: Algoritmo di programmazione dinamica per BISACCIA.

- $I = \{e_1, e_2, e_3, e_4, e_5, e_6\}$;
- $s(e_1) = 3, s(e_2) = 4, s(e_3) = 1, s(e_4) = 5, s(e_5) = 3, s(e_6) = 2\}$;
- $p(e_1) = 2, p(e_2) = 3, p(e_3) = 1, p(e_4) = 5, p(e_5) = 3, p(e_6) = 1\}$;
- $B = 8, K = 6$.

L'Algoritmo 9.3 applicato a questa istanza riempie una tabella di 6 righe e 9 colonne nel modo seguente.

$\emptyset$	-	-	$\{e_1\}$	-	-	-	-	
$\emptyset$	-	-	$\{e_1\}$	$\{e_2\}$	-	-	$\{e_1, e_2\}$	-
$\emptyset$	$\{e_3\}$	-	$\{e_1\}$	$\{e_2\}$	$\{e_1, e_3\}$	-	$\{e_1, e_2\}$	$\{e_1, e_2, e_3\}$
$\emptyset$	$\{e_3\}$	-	$\{e_1\}$	$\{e_2\}$	$\{e_4\}$	$\{e_3, e_4\}$	$\{e_1, e_2\}$	$\{e_1, e_4\}$
$\emptyset$	$\{e_3\}$	-	$\{e_5\}$	$\{e_3, e_5\}$	$\{e_4\}$	$\{e_3, e_4\}$	$\{e_2, e_5\}$	$\{e_4, e_5\}$
$\emptyset$	$\{e_3\}$	$\{e_6\}$	$\{e_5\}$	$\{e_3, e_5\}$	$\{e_4\}$	$\{e_3, e_4\}$	$\{e_2, e_5\}$	$\{e_4, e_5\}$

La soluzione di profitto massimo nell'ultima riga è $\{e_4, e_5\}$ con profitto $\overline{p} = 8 > K = 6$, per cui l'algoritmo dà risposta VERO.

Esercizio 9.13 Dimostrare che l'Algoritmo 9.3 fornisce la risposta corretta.

Contrariamente a quanto può apparire, l'Algoritmo 9.3 non è polinomiale nella lunghezza dell'istanza. Infatti, una codifica ragionevole di una istanza di BISACCIA prevede:

- $O(\log s(e_i)) = O(\log s_M)$ bit per codificare la dimensione dell'elemento e_i (dove $s_M = \max_{e \in I} s(e)$);
- $O(\log p(e_i)) = O(\log p_M)$ bit per codificare il profitto dell'elemento e_i (dove $p_M = \max_{e \in I} p(e)$);
- $\log B + \log K$ bit per codificare B e K.

Ciò ci fornisce un totale di $O(n(\log s_M + \log p_M) + \log B + \log K)$ bit e, dato che possiamo assumere senza perdita di generalità che $s_M \leq B$ e $p_M \leq K$, l'istanza può essere codificata con $O(n(\log B + \log K))$ bit.

Si noti che in generale nB non è limitata superiormente da nessuna funzione polinomiale di $n(\log B + \log K)$, per cui l'algoritmo risulta avere complessità non polinomiale.

Si noti però che, se si considerano istanze comprendenti soltanto valori "piccoli" di B e, come conseguenza, di tutti gli $s(e_i)$, allora l'Algoritmo 9.3 risulta polinomiale. Ciò vale in particolare nel caso in cui $B = O(n^r)$ per qualche costante r, nel caso cioè in cui tutte le dimensioni presenti nell'istanza sono polinomialmente limitate in n.

In questo caso, quindi, la NP-completezza di BISACCIA deriva non dalla struttura combinatoria del problema, ma dalla (possibile) presenza di numeri molto grandi nell'istanza.

Passiamo ora a generalizzare e formalizzare meglio le considerazioni precedenti, estendendo le considerazioni effettuate a proposito della valutazione della dimensione di un'istanza.

Definizione 9.10 *Dato un problema $\mathcal{P}$, definiamo come* funzione massimo *la funzione $\mu : I_{\mathcal{P}} \mapsto \mathbb{Q}$ che mappa ogni istanza di $\mathcal{P}$ in una stima del valore più grande che compare al suo interno.*

Due coppie $\langle \lambda, \mu \rangle$, $\langle \lambda', \mu' \rangle$ sono polinomialmente correlate se esistono due polinomi bivariati p, p' tali che, per ogni $x \in I_{\mathcal{P}}$:

- λ e λ' sono polinomialmente correlate
- $\mu(x) \leq p'(\lambda'(x), \mu'(x))$
- $\mu'(x) \leq p(\lambda(x), \mu(x))$

Esercizio 9.14 Si considerino, per ogni istanza di PARTIZIONE, le seguenti funzioni lunghezza:

1. $\sum_{e \in I} \log_2 e$
2. $\max_{e \in I} \log_2 e$
3. $n\lceil \log_2 e \rceil$

E le seguenti funzioni massimo:

1. $\max_{e \in I} e$
2. $\sum_{e \in I} e$
3. $\left\lceil \sum_{e \in I} e / \, |I| \right\rceil$

Mostrare che tutte le combinazioni tra funzioni lunghezza e funzioni massimo sono tra loro polinomialmente correlate.

Definizione 9.11 *Dato un problema $\mathcal{P}$ e date due funzioni $\lambda : I_{\mathcal{P}} \mapsto \mathbb{N}$ e $\mu : I_{\mathcal{P}} \mapsto \mathbb{N}$, un algoritmo che risolve $\mathcal{P}$ è detto* pseudo-polinomiale *se, per ogni istanza x, la sua complessità è polinomiale nelle due variabili $\lambda(x)$ e $\mu(x)$.*

Ad esempio l'Algoritmo 9.3 è pseudo-polinomiale, in quanto la sua complessità $O(nB)$ è polinomiale rispetto sia a $\lambda(x)$ che a $\mu(x)$ per ogni istanza x e per ogni coppia di funzioni λ, μ ragionevoli.

Nel caso di TAGLIO, per ogni istanza $x = \langle G = (V, E), w \rangle$ possiamo definire $\mu(x) = \max_{e \in E} w(e)$, mentre, nel caso di BISACCIA, possiamo porre $\mu(x) = \max\{B, K\}$.

Definizione 9.12 *Un problema $\mathcal{P} \in$ NP è detto* numerico *se non esiste alcun polinomio p tale che, per ogni istanza $x \in I_{\mathcal{P}}$, $\mu(x) \leq p(\lambda(x))$.*

Essenzialmente, un problema è numerico se all'interno di sue istanze possono comparire valori indipendenti dalla struttura combinatoria del problema.

Esempio 9.4 Il problema TAGLIO è numerico in quanto i pesi assegnati agli archi possono assumere valori arbitrariamente grandi, indipendentemente dalla struttura del grafo.

Esempio 9.5 Il problema COPERTURA CON NODI non è numerico, in quanto l'unico valore presente in una sua istanza $x = \langle G = (V, E), K \rangle$ è K, che per definizione, sarà $K \leq |V|$, cioè non superiore alla dimensione di un insieme che deve essere enumerato nella descrizione dell'istanza.

Esempio 9.6 Un altro esempio di algoritmo di programmazione dinamica può essere introdotto relativamente ad un altro problema NP-completo, il problema PARTIZIONE, definito come segue.

PARTIZIONE
ISTANZA: Insieme $I = \{e_1, e_2, \ldots, e_n\}$ di naturali
PREDICATO: Esiste una partizione $\{I_1, I_2\}$ di I tale che $\sum_{e \in I_1} e = \sum_{e \in I_2} e = \frac{1}{2} \sum_{e \in I} e$?

L'algoritmo riempie una tabella T di elementi booleani con $n = | I |$ righe e $B = \frac{1}{2} \sum_{e \in I} e$ colonne in modo tale che

$$T(i,j) = \begin{cases} \text{VERO se } \exists I' = \{e_1, e_2, \ldots, e_i\} \text{ tale che } \sum_{e \in I'} e = j \\ \text{FALSO altrimenti} \end{cases}$$

La tabella viene riempita utilizzando le relazioni seguenti:

$$T(1,j) := \text{VERO se} \begin{cases} j = 0 \\ j = e_1 \end{cases}$$

$$T(i,j) := \text{VERO se} \begin{cases} T(i-1,j) = \text{VERO} \\ T(i-1, j-e_i) = \text{VERO} \end{cases} \quad i > 1$$

L'istanza è positiva se $T(n, B) = \text{VERO}$.

Esercizio 9.15 Assumendo noto che il problema BISACCIA sia NP-completo, dimostrare l'NP-completezza di PARTIZIONE.

Chiaramente, non è possibile avere algoritmi pseudo-polinomiali che risolvono un problema non numerico NP–completo, a meno che P = NP.

Dato un problema $\mathcal{P} \in$ NP e dato un polinomio p, indichiamo con $\mathcal{P}^p$ il sottoproblema ottenuto da $\mathcal{P}$ considerando soltanto le istanze x per cui $\mu(x) \leq p(\lambda(x))$. Per ipotesi, $\mathcal{P}^p$ non è un problema numerico. Inoltre, se $\mathcal{P}$ è risolubile da un algoritmo pseudo-polinomiale allora $\mathcal{P}^p \in$ P.

Definizione 9.13 *Un problema $\mathcal{P}$ è detto* fortemente NP*–completo se $\mathcal{P} \in$ NP e se esiste un polinomio p tale che $\mathcal{P}^p$ è* NP*–completo.*

Si osservi che tutti i problemi NP–completi non numerici sono necessariamente fortemente NP–completi (si consideri il polinomio $p(n) = 1$).

Essenzialmente, possiamo dire che in un problema fortemente NP–completo la difficoltà di soluzione deriva dalla stessa complessità strutturale delle istanze del problema, al contrario dei problemi non fortemente NP–completi, risolubili mediante algoritmi pseudo-polinomiali.

Teorema 9.12 *Un problema fortemente* NP*–completo non può essere risolto da un algoritmo pseudo–polinomiale, a meno che* P = NP*.*

Dimostrazione. Supponiamo per assurdo che esista un algoritmo pseudo-polinomiale $\mathcal{A}$ che risolve il problema $\mathcal{P}$ fortemente NP–completo.

Da ciò deriva che $\mathcal{A}$ risolve una istanza generica $x \in I_{\mathcal{P}}$ in tempo $p(\lambda(x), \mu(x))$, dove p è un opportuno polinomio. Ma allora, per ogni polinomio q, $\mathcal{P}^q$ può essere risolto in tempo $p(\lambda(x), q(\lambda(x)))$, e quindi in tempo polinomiale, contrariamente alle ipotesi di NP–completezza forte di $\mathcal{P}$. □

Esempio 9.7 Il problema COMMESSO VIAGGIATORE è definito come segue.

COMMESSO VIAGGIATORE

ISTANZA: Insieme $C = \{c_1, c_2, \ldots, c_n\}$ (città), matrice simmetrica D $n \times n$ di razionali (distanze), $B \in \mathbb{Q}$

PREDICATO: Esiste un circuito che attraversa tutte le città ed ha lunghezza non superiore a B, vale a dire, esiste una permutazione $C' = \{c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)}\}$ di C tale che $\sum_{i=1}^{n} D(\pi(i), \pi(i+1)) + D(\pi(n), \pi(1)) \leq B$?

COMMESSO VIAGGIATORE è un problema numerico, in quanto i valori in D sono indipendenti dal resto dell'istanza. È possibile comunque provare che esso è fortemente NP–completo mostrando una Karp–riduzione polinomiale dal problema NP–completo CICLO HAMILTONIANO al sottoproblema COMMESSO VIAGGIATORE-{1,2} di COMMESSO VIAGGIATORE in cui tutti i valori in D sono ristretti ad avere dominio $\{1, 2\}$.

Data una istanza $G = (V, E)$ di CICLO HAMILTONIANO, costruiamo da essa una istanza di COMMESSO VIAGGIATORE-{1,2} nel modo seguente:

- $C = V$;
- per ogni coppia c_i, c_j, $D(i, j) = D(j, i) = 1$ se e solo se $(v_i, v_j) \in E$, altrimenti $D(i, j) = D(j, i) = 2$;
- $B = n$.

Chiaramente, esiste un ciclo hamiltoniano in G se e solo se esiste un circuito di lunghezza B tra gli elementi di C.

9.6 La gerarchia polinomiale

Riprendendo le considerazioni relative alla caratterizzazione della classe NP, fatte nella Sezione 9.3, possiamo ora mostrare come le classi NP e CO-NP costituiscanoClasse di complessità!CO-NP—(solo il primo livello di una gerarchia di classi di problemi di decisione, aventi struttura combinatoria via via più complessa.

Ricordando quanto detto nella Sezione 9.3, possiamo dire che un problema di decisione $\mathcal{P}$ è in NP se e solo se esiste un predicato $P(x, y)$ decidibile in tempo polinomiale tale che $x \in Y_{\mathcal{P}}$ se e solo se $\exists y, |y| \leq p(|x|) : P(x, y)$, dove quindi y è il certificato relativo all'istanza positiva x.

In tal caso, il non determinismo della macchina di Turing non deterministica viene utilizzato per "indovinare" un valore della variabile quantificata per il quale il predicato è vero.

Ad esempio, consideriamo il seguente problema.

INSIEME INDIPENDENTE
ISTANZA: Grafo $G = (V, E)$, $K > 0$ intero
PREDICATO: Esiste un insieme indipendente di taglia $\geq K$, cioè un $U \subseteq V$ tale che $\mid U \mid \geq K$ e $\nexists (u, v) \in E$ con $u \in U \wedge v \in U$?

Nel caso di INSIEME INDIPENDENTE la stringa x è costituita dalla rappresentazione del grafo G e del valore K, mentre il certificato y è la rappresentazione di un particolare sottoinsieme $U \subseteq V$. La verifica espressa dal predicato $P(x, y)$ richiede di accertarsi che $\mid U \mid \geq K$ e che U è un insieme indipendente. Una computazione non deterministica per questo problema consiste di un passo non deterministico di scelta (*guess*) di un sottoinsieme di V, seguito da una verifica in tempo deterministico polinomiale.

Una ulteriore, interessante interpretazione dei problemi in NP vede tali problemi come *giochi* di una sola mossa, eseguita da un giocatore (A) in cui ci si chiede se A, muovendo, può raggiungere una posizione vincente, assunte le non restrittive ipotesi che il valutare se una posizione è vincente per A richieda tempo deterministico polinomiale[6] e che una mossa sia descrivibile in modo sufficientemente succinto rispetto alla descrizione di una posizione.

Se ora consideriamo l'insieme $Y_{\mathcal{P}}$ di un qualche problema di decisione $\mathcal{P} \in$ CO-NP (istanze negative di un problema in NP), vediamo facilmente che esso potrà essere descritto mediante una formula del tipo

$$\nexists y, \mid y \mid \leq p(\mid x \mid) : P(x, y) \equiv \forall y, \mid y \mid \leq p(\mid x \mid) : \neg P(x, y) \equiv$$
$$\equiv \forall y, \mid y \mid \leq p(\mid x \mid) : P'(x, y)$$

dove $P'(x, y)$ è il predicato negato di $P(x, y)$, anch'esso, per definizione, verificabile in tempo deterministico polinomiale. Quindi, possiamo vedere i problemi in CO-NP come giochi di una mossa in cui ci si chiede se l'unico giocatore che muove (A), facendo la sua mossa, non può raggiungere in nessun modo una posizione vincente.

Al tempo stesso, come caso limite, un problema in P può chiaramente essere visto come la verifica di una formula del tipo $P(x)$, priva cioè di quantificatori, e quindi come giochi in cui ci si chiede se la posizione raggiunta è una posizione vincente per A.

A partire dalle considerazioni precedenti, possiamo allora definire due nuove gerarchie di classi di complessità, di cui P ed NP rappresentano i livelli di ordine 0 ed 1 nella prima, e P e CO-NP i livelli di ordine 0 ed 1 nella seconda..

[6] D'ora in poi, assumeremo sempre che il predicato P sia valutabile in tempo deterministico polinomiale.

Tale gerarchia è definita dalle classi di problemi i cui insiemi di istanze positive sono descrivibili mediante formule del tipo

$$\exists y_1 \forall y_2 \exists y_3 \forall y_4 \ldots Q y_k P(x, y_1, y_2, y_3, y_4, \ldots, y_k)$$

dove k è un intero non negativo, e Q è $\exists$ se k è dispari e $\forall$ se k è pari, ed in ogni caso $|\, y_i \,| \leq p(|\, x \,|)$ per un opportuno polinomio p.

In generale, una formula composta da un predicato preceduto da una sequenza di quantificatori esistenziali ed universali alternati come

$$\exists y_1 \forall y_2 \exists y_3 \forall y_4 \exists y_5 \ldots P(x, y_1, y_2, y_3, y_4, y_5, \ldots)$$

può essere interpretata come un gioco ad informazione perfetta (ad esempio come scacchi o dama) tra due giocatori A e B, in cui le variabili quantificate rappresentano le mosse eseguite a turno, la variabile non quantificata x rappresenta la posizione di partenza ed il predicato P rappresenta la condizione di vittoria per A, che assumiamo giochi per primo. Dire che A ha una strategia vincente significa affermare che:

esiste una mossa y_1 *di* A *tale che*
per ogni mossa y_2 *di* B
esiste una mossa y_3 *di* A *tale che*
per ogni mossa y_4 *di* B
esiste una mossa y_5 *di* A *tale che*
$\vdots$
il giocatore A *vince.*

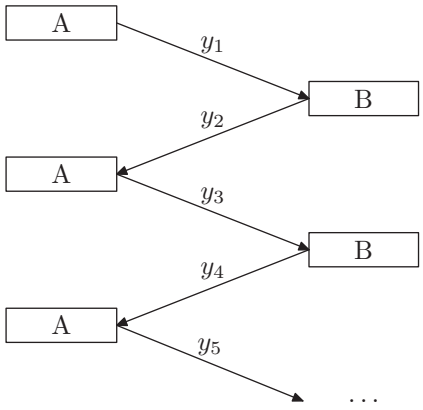


FIGURA 9.11 Gioco tra A e B

Quindi, la formula è verificata se e solo se A ha una strategia vincente nel gioco.

Si noti che se la formula è del tipo

$$\forall y_1 \exists y_2 \forall y_3 \exists y_4 \exists y_5 \ldots P(x, y_1, y_2, y_3, y_4, y_5, \ldots)$$

allora la formula è vera se e solo se B ha una strategia vincente nel gioco (tutte le strategie sono perdenti per A).

Definizione 9.14 *Per ogni problema di decisione $\mathcal{P}$, $\mathcal{P} \in \Sigma_k^p$ ($k \geq 0$) se e solo se esiste un predicato $P(x, y_1, \ldots, y_k) \in$ P ed un polinomio $p(\cdot)$ tali che, per ogni istanza x di $\mathcal{P}$, $x \in Y_{\mathcal{P}}$ se e solo se $\exists y_1 \forall y_2 \ldots Q y_k \; P(x, y_1, \ldots, y_k)$ è vera, dove $|y_i| \leq p(|x|)$, $i = 1, \ldots, k$ e $Q = \exists$ per k dispari, $Q = \forall$ per k pari.*

La seconda gerarchia che introduciamo è definita dalle classi di problemi i cui insiemi di istanze positive sono descrivibili mediante formule del tipo

$$\forall y_1 \exists y_2 \forall y_3 \exists y_4 \ldots Q s y_k P(x, y_1, y_2, y_3, y_4, \ldots, y_k)$$

dove k è un intero non negativo, e Q è $\forall$ se k è dispari e $\exists$ se k è pari.

Definizione 9.15 *Per ogni problema di decisione $\mathcal{P}$, $\mathcal{P} \in \Pi_k^p$ ($k \geq 0$) se e solo se esiste un predicato $P(x, y_1, \ldots, y_k) \in$ P ed un polinomio $p(\cdot)$ tali che, per ogni istanza x di $\mathcal{P}$, $x \in Y_{\mathcal{P}}$ se e solo se $\forall y_1 \exists y_2 \ldots Q y_k \; P(x, y_1, \ldots, y_k)$ è vera, dove $|y_i| \leq p(|x|)$, $i = 1, \ldots, k$ e $Q = \forall$ per k dispari, $Q = \exists$ per k pari.*

Si può banalmente verificare che per $k = 0$ si ha effettivamente $\Sigma_0^p = \Pi_0^p =$ P. Allo stesso modo, è immediato che $\Sigma_1^p =$ NP e $\Pi_1^p =$ co-NP.

Inoltre, vale il seguente teorema, generalizzazione di quanto già mostrato per NP e co-NP.

Teorema 9.13 *Per ogni $k \geq 0$, $\Pi_k^p =$ co-Σ_k^p.*

Dimostrazione. Per ogni problema $\mathcal{P} \in \Sigma_k^p$ ogni istanza $x \in N_{\mathcal{P}} = Y_{\overline{\mathcal{P}}}$ verifica la formula

$$\neg(\exists y_1 \forall y_2 \ldots Q y_k \; P(x, y_1, \ldots, y_k)) \equiv \forall y_1 \exists y_2 \ldots Q' y_k \; P'(x, y_1, \ldots, y_k)$$

dove $P'(\cdot)$ è il predicato negato di $P(\cdot)$ e $Q' = \exists$ se $Q = \forall$, $Q' = \forall$ se $Q = \exists$.

Da ciò deriva che $\forall \mathcal{P} \in \Sigma_k^p : \overline{\mathcal{P}} \in \Pi_k^p$, e quindi che co-$\Sigma_k^p \subseteq \Pi_k^p$.

Inoltre, per ogni problema $\mathcal{P} \in \Pi_k^p$ ogni istanza $x \in Y_{\mathcal{P}}$ verifica la formula

$$\forall y_1 \exists y_2 \ldots Q y_k \; P(x, y_1, \ldots, y_k) \equiv \neg(\exists y_1 \forall y_2 \ldots Q' y_k) \; P'(x, y_1, \ldots, y_k)$$

dove, nuovamente, P' è il predicato negato di P e $Q' = \exists$ se $Q = \forall$, $Q' = \forall$ se $Q = \exists$. Ma l'insieme delle x tali che $\neg(\exists y_1 \forall y_2 \ldots Q' y_k) \; P'(x, y_1, \ldots, y_k)$ rappresenta, per definizione, le istanze negative di un qualche problema $\mathcal{P}' \in \Sigma_k^p$.

Da ciò consegue che $\forall \mathcal{P} \in \Pi_k^p \; \exists \mathcal{P}' \in \Sigma_k^p : \mathcal{P} = \overline{\mathcal{P}}'$, e quindi che $\Pi_k^p \subseteq$ co-Σ_k^p. □

Il seguente corollario deriva immediatamente notando che, per due qualunque classi di complessità $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A} = \mathcal{B} \Rightarrow$ co-$\mathcal{A}$ = co-$\mathcal{B}$.

Corollario 9.14 *Per ogni* $k \geq 0$, $\Sigma_k^p = co\text{-}\Pi_k^p$.

Ai fini di quanto seguirà introduciamo ora la seguente notazione.

Definizione 9.16 *Data una classe di complessità* $\mathcal{C}$, *indichiamo con* $\exists\mathcal{C}$ *l'insieme dei problemi di decisione tale che* $\Pi \in \exists\mathcal{C}$ *se e solo se esistono un polinomio* $p(\cdot)$ *ed un predicato binario* P *in* $\mathcal{C}$ *tali che* $Y_\Pi = \{x \mid \exists y\, P(x,y)\}$, *dove* $|y| \leq p(|x|)$.

Allo stesso modo, possiamo definire $\forall\mathcal{C}$.

Proposizione 9.15 *Le relazioni seguenti derivano immediatamente dalle definizioni.*

1. $\Sigma_k^p = \underbrace{\exists\forall\ldots Q}_{k}\,\mathrm{P}$
2. $\Pi_k^p = \underbrace{\forall\exists\ldots Q}_{k}\,\mathrm{P}$
3. $\exists\Sigma_k^p = \Sigma_k^p$
4. $\forall\Pi_k^p = \Pi_k^p$

Teorema 9.16 *Per ogni* $k \geq 0$ $\Sigma_k^p \cup \Pi_k^p \subseteq \Sigma_{k+1}^p \cap \Pi_{k+1}^p$.

Dimostrazione. Sia $\mathcal{P} \in \Sigma_k^p$. Allora $Y_\mathcal{P} = \{x \mid \exists y_1 \ldots Q y_k\, P(y_1, \ldots, y_k, x)\}$, ma anche $Y_\Pi = \{x \mid \exists y_1 \ldots Q y_k Q' y_{k+1}\, P(y_1, \ldots, y_k, x)\}$ (dove $Q' = \exists$ se e solo se $Q = \forall$) e quindi $\mathcal{P} \in \Sigma_{k+1}^p$. Inoltre, $Y_\mathcal{P} = \{x \mid \forall y_{k+1}\exists y_1 \ldots Q y_k\, P(y_1, \ldots, y_k, x)\}$ e quindi $\mathcal{P} \in \Pi_{k+1}^p$.

Le stesse considerazioni possono essere applicate se $\mathcal{P} \in \Pi_k^p$. □

Il diagramma in Figura 9.12 mostra la struttura delle inclusioni per i primi livelli delle classi Σ_k^p e Π_k^p.

Definiamo come *gerarchia polinomiale* PH l'unione di tutte le classi Σ_k^p, $\mathrm{PH} = \cup_k \Sigma_k^p$: dalle relazioni di inclusione mostrate in Figura 9.12 risulta anche chiaro che $\mathrm{PH} = \cup_k \Pi_k^p$.

Per nessuna delle inclusioni si sa se è propria o meno, anche se delle dipendenze tra le diverse relazioni di inclusioni possono essere dimostrate.

Teorema 9.17 *Per ogni* $k \geq 1$ *le seguenti relazioni sono equivalenti:*

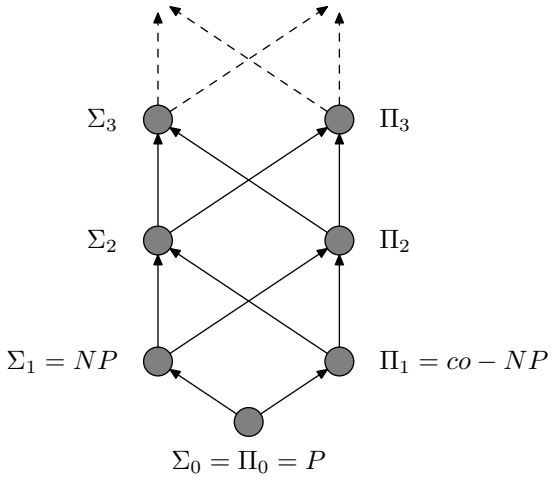


FIGURA 9.12 Relazioni di inclusione in PH

---

1. $\Sigma_k^p = \Sigma_{k+1}^p$
2. $\Pi_k^p = \Pi_{k+1}^p$
3. $\Sigma_k^p = \Pi_k^p$
4. $\Sigma_k^p = \Pi_{k+1}^p$
5. $\Pi_k^p = \Sigma_{k+1}^p$

Dimostrazione. 1 e 2 sono equivalenti per complementazione. Lo stesso vale per 4 e 5.

Mostriamo ora che 1, 3 e 4 sono equivalenti tra loro.

- 1⇒3 deriva da $\Sigma_k^p \subseteq \Pi_{k+1}^p$co-$\Sigma_{k+1}^p$ (dal Teorema 9.16) e co-Σ_{k+1}^p = co-$\Sigma_k^p = \Pi_k^p$ (da 1), che implicano $\Sigma_k^p \subseteq \Pi_k^p$, e da $\Pi_k^p \subseteq \Sigma_{k+1}^p =$ co-Π_{k+1}^p (dal Teorema 9.16) e co-$\Pi_{k+1}^p = \Sigma_{k+1}^p = \Sigma_k^p$ (da 1), che implicano $\Pi_k^p \subseteq \Sigma_k^p$;
- 3⇒4 deriva da $\Sigma_k^p = \Pi_k^p$ (da 3), $\Pi_k^p = \forall\Pi_k^p$ (dalla Proposizione 9.15), da $\forall\Pi_k^p = \forall\Sigma_k^p$ (da 3) e da $\forall\Sigma_k^p = \Pi_{k+1}^p$ per definizione;
- 4⇒1 deriva da $\Sigma_{k+1}^p =$ co-Π_{k+1}^p (per definizione), co-Π_{k+1}^p = co-Σ_k^p (da 4), co-$\Sigma_k^p = \Pi_k^p \subseteq \Pi_{k+1}^p$ (per definizione) e $\Pi_{k+1}^p = \Sigma_k^p$ (da 4).

□

Teorema 9.18 *Per ogni $k \geq 1$ le condizioni 1, 2, 3, 4, 5 del teorema 9.17 sono equivalenti a* $\mathrm{PH} = \Sigma_k^p$.

Dimostrazione. Data l'equivalenza, già mostrata, tra le condizioni nel Teorema 9.17, considereremo soltanto l'equivalenza tra $\Sigma_k^p = \Sigma_{k+1}^p$ e $\mathrm{PH} = \Sigma_k^p$.

È immediato che da $\mathrm{PH} = \Sigma_k^p$ deriva necessariamente $\Sigma_k^p = \Sigma_{k+1}^p$. L'implicazione opposta deriva osservando che, se $\Sigma_k^p = \Sigma_{k+1}^p$, allora $\Pi_k^p = \Pi_{k+1}^p$ (dal Teorema 9.17) e $\Sigma_{k+2}^p = \exists\Pi_{k+1}^p = \exists\Pi_k^p = \Sigma_{k+1}^p = \Sigma_k^p$ e, per induzione, si ha che per ogni $i \geq k$ $\Sigma_i^p = \Sigma_k^p$ e, dato che $\cup_{i \leq k}\Sigma_i^p = \Sigma_k^p$ per le relazioni di inclusione già mostrate, ne deriva $\mathrm{PH} = \Sigma_k^p$. □

È quindi sufficiente che, per un qualunque livello k della gerarchia si abbia $\Sigma_k^p = \Sigma_{k+1}^p$ (o una qualunque delle relazioni equivalenti del Teorema 9.17) perché risulti che tutta la gerarchia polinomiale sia uguale a Σ_k^p: tale situazione viene detta *collasso* della gerarchia polinomiale al livello k.

Si noti anche che, al contrario, se, per qualche k, $\Sigma_k^p \neq \Sigma_{k+1}^p$ (la gerarchia polinomiale non collassa) allora $\Sigma_0^p = \mathrm{P} \neq \Sigma_1^p = \mathrm{NP} \neq \ldots \neq \Sigma_k^p \neq \Sigma_{k+1}^p$. Ne deriva, tra l'altro, che ogni condizione $\Sigma_k^p \neq \Sigma_{k+1}^p$ (o, ad esempio, $\Sigma_k^p \neq \Pi_k^p$) con $k \geq 1$ implica $\mathrm{P} \neq \mathrm{NP}$: si osservi che, come caso particolare, ciò implica il già noto risultato che $\mathrm{NP} \neq \text{co-NP} \Rightarrow \mathrm{P} \neq \mathrm{NP}$. Inoltre, da quanto mostrato deriva anche che ogni condizione $\Sigma_k^p \neq \Sigma_{k+1}^p$ (o $\Sigma_k^p \neq \Pi_k^p$) con $k \geq 2$ implica $\mathrm{NP} \neq \text{co-NP}$.

Una definizione alternativa della gerarchia polinomiale può essere fornita facendo riferimento alla *relativizzazione* di classi di complessità.

Definizione 9.17 *Dato un problema $\mathcal{P}$, definiamo la classe relativizzata* $\mathrm{P}^{\mathcal{P}}$ *come l'insieme dei problemi risolubili in tempo polinomiale da un macchina di Turing deterministica con oracolo per $\mathcal{P}$.*

Si noti che $\mathcal{P}' \in \mathrm{P}^{\mathcal{P}}$ è equivalente a $\mathcal{P}' \leq_T^p \mathcal{P}$.

Definizione 9.18 *Dato un problema $\mathcal{P}$, definiamo la classe relativizzata* $\mathrm{NP}^{\mathcal{P}}$ *come l'insieme dei problemi risolubili in tempo polinomiale da un macchina di Turing non deterministica con oracolo per $\mathcal{P}$.*

Le definizioni precedenti possono essere estese a classi relativizzate rispetto ad altre classi.

Definizione 9.19 *Data una classe A, un problema $\mathcal{P}'$ appartiene alla classe relativizzata* P^A *se esiste un problema $\mathcal{P} \in A$ tale che $\mathcal{P} \in \mathrm{P}^{\mathcal{P}}$. Cioè,* P^A *è l'insieme dei problemi risolubili in tempo polinomiale da un macchina di Turing deterministica con oracolo per un opportuno problema $\mathcal{P} \in A$.*

Definizione 9.20 *Data una classe A, un problema $\mathcal{P}'$ appartiene alla classe relativizzata NP^A se esiste un problema $\mathcal{P} \in A$ tale che $\mathcal{P} \in \mathrm{NP}^{\mathcal{P}}$. Cioè, NP^A è l'insieme dei problemi risolubili in tempo polinomiale da un macchina di Turing non deterministica con oracolo per un opportuno problema $\mathcal{P} \in A$.*

Si noti che alcune classi già definite possono essere interpretate anche come classi relativizzate: ad esempio, $\mathrm{P} = \mathrm{P}^{\mathrm{P}}$ e $\mathrm{NP} = \mathrm{NP}^{\mathrm{P}}$.

A partire da operazioni di relativizzazione operate a partire dalle classi P e NP, possiamo definire la seguente gerarchia di classi.

Definizione 9.21 *Per ogni intero $k \geq 0$, la classe $\overline{\Sigma}_k^p$ è definita come $\overline{\Sigma}_k^p = \mathrm{P}$ se $k = 0$ e $\overline{\Sigma}_k^p = \mathrm{NP}^{\overline{\Sigma}_{k-1}^p}$ se $k > 0$.*

Definizione 9.22 *Per ogni intero $k \geq 0$, la classe $\overline{\Pi}_k^p$ è definita come co-$\overline{\Sigma}_k^p$.*

Definizione 9.23 *Per ogni intero $k \geq 0$, la classe $\overline{\Delta}_k^p$ è definita come $\overline{\Delta}_k^p = \mathrm{P}$ se $k = 0$ e $\overline{\Delta}_k^p = \mathrm{P}^{\overline{\Sigma}_{k-1}^p}$ se $k > 0$.*

Esercizio 9.16 Mostrare che dalle precedenti definizioni deriva che $\overline{\Sigma}_0^p = \overline{\Pi}_0^p = \overline{\Delta}_0^p = \mathrm{P}$, $\overline{\Sigma}_1^p = \mathrm{NP}$, $\overline{\Pi}_1^p = \text{co-NP}$, $\overline{\Delta}_1^p = \mathrm{P}$, $\overline{\Sigma}_2^p = \mathrm{NP}^{\mathrm{NP}}$, $\overline{\Pi}_2^p = \text{co-NP}^{\mathrm{NP}}$, $\overline{\Delta}_2^p = \mathrm{P}^{\mathrm{NP}}$.

L'equivalenza tra le classi Σ_k^p, Π_k^p, che definiscono la gerarchia polinomiale, e le classi $\overline{\Sigma}_k^p$, $\overline{\Pi}_k^p$ deriva dal seguente teorema, la cui dimostrazione è lasciata per esercizio.

Teorema 9.19 *Dato un linguaggio L e dato un intero $k \geq 1$, $L \in \overline{\Sigma}_k^p$ se e solo se $L \in \Sigma_k^p$.*

Dimostrazione. La dimostrazione è lasciata per esercizio (vedi Esercizio 9.17). □

Esercizio 9.17 Dimostrare il Teorema 9.19.

La caratterizzazione mediante formule booleane quantificate con alternanza di quantificatori permette di definire anche una sequenza di problemi completi (rispetto alla Karp-riducibilità) per le classi Σ_k^p.

Per ogni intero $i \geq 1$, definiamo il problema Formula Booleana Quantificata i-limitata nel modo seguente:

FORMULA BOOLEANA QUANTIFICATA i-LIMITATA

ISTANZA: Una formula booleana $\mathcal{F}$ su una sequenza $X = \{x_1, x_2, \ldots, x_i\}$ di variabili booleane.

PREDICATO: La formula booleana quantificata

$$\forall x_1 \exists x_2 \forall x_3 \ldots Q x_i \mathcal{F}(x_1, x_2, \ldots, x_i)$$

è vera?

Teorema 9.20 *Per ogni $i \geq 1$,* FORMULA BOOLEANA QUANTIFICATA i-LIMITATA *è Σ_i^p-completo rispetto alla Karp-riducibilità polinomiale.*

Dimostrazione. La dimostrazione è lasciata come esercizio (vedi Esercizio 9.18). □

Esercizio 9.18 Dimostrare il Teorema 9.20.

9.7 La classe PSPACE

La classe PSPACE, definita come l'insieme dei problemi risolubili in spazio polinomiale nella dimensione dell'istanza, viene a costituire il primo e più naturale esempio di classe di complessità che non collassa a P nel caso in cui P=NP: vale a dire che P = NP non implica PSPACE = P. Inoltre, la caratterizzazione di problemi mediante spazio di risoluzione polinomiale presenta l'interessante caratteristica di essere invariante rispetto all'introduzione del non determinismo: infatti, in conseguenza del Teorema di Savitch (Teorema 8.26), ogni problema risolubile in spazio polinomiale da macchine di Turing non deterministiche è risolubile in spazio polinomiale anche da macchine di Turing deterministiche, vale a dire che NSPACE=DSPACE.

Come vedremo, la classe PSPACE rappresenta inoltre l'insieme di linguaggi cui tende la Gerarchia Polinomiale PH al crescere indefinito del numero di quantificatori alternati nella formula, o, corrispondentemente, della lunghezza della sequenza di relativizzazioni operate.

Possiamo verificare l'inclusione di PH in PSPACE come conseguenza del Teorema seguente, che identifica, corrispondentemente a quanto effettuato nei Teoremi 9.2 e 9.4 per le classi P ed NP, un problema completo in PSPACE iniziale.

Sia FORMULA BOOLEANA QUANTIFICATA il problema definito nel modo seguente.

FORMULA BOOLEANA QUANTIFICATA

ISTANZA: Una formula booleana $\mathcal{F}$ su una sequenza $X = \{x_1, x_2, \ldots, x_n\}$ di variabili booleane.

PREDICATO: La seguente formula booleana quantificata Φ:

$$\Phi = Q_1 x_1 Q_2 x_2 Q_3 x_3 \ldots Q_n x_n \mathcal{F}(x_1, x_2, \ldots, x_n),$$

dove, per ogni i, $Q_i \in \{\exists, \forall\}$, $Q_i = \exists \Rightarrow Q_{i+1} = \forall$ e $Q_i = \forall \Rightarrow Q_{i+1} = \exists$, è vera?

Come si può immediatamente vedere, FORMULA BOOLEANA QUANTIFICATA è una generalizzazione di tutti i problemi FORMULA BOOLEANA QUANTIFICATA i-LIMITATA per ogni $i > 0$. Mostriamo ora che vale il seguente teorema.

Teorema 9.21 *Il problema* FORMULA BOOLEANA QUANTIFICATA *è* PSPACE*-completo rispetto alla Karp–riducibilità log-space.*

Dimostrazione. Per mostrare che FORMULA BOOLEANA QUANTIFICATA $\in$ PSPACE introduciamo una procedura ricorsiva *Val* che risolve il problema su una istanza generica $\Phi = \exists x_1 \forall x_2\ \exists x_3\ \ldots Q x_m \mathcal{F}(x_1, x_2, \ldots, x_n)$.

La procedura *Val* ha la struttura seguente, e viene applicata inizialmente per $\phi = \Phi$:

- se $\phi =$ TRUE allora $Val(\phi) =$ TRUE;
- $\phi =$ FALSE allora $Val(\phi) =$ FALSE;
- $\phi = \overline{\psi}$ allora $Val(\phi) = \overline{Val(\psi)}$;
- $\phi = \psi \wedge \psi'$ allora $Val(\phi) = Val(\psi) \wedge Val(\psi')$;
- $\phi = \psi \vee \psi'$ allora $Val(\phi) = Val(\psi) \vee Val(\psi')$;
- $\phi = \forall x \psi$ allora $Val(\phi) = Val(\psi\ |_{x=\text{TRUE}}) \wedge Val(\psi\ |_{x=\text{FALSE}})$;
- $\phi = \exists x \psi$ allora $Val(\phi) = Val(\psi\ |_{x=\text{TRUE}}) \vee Val(\psi\ |_{x=\text{FALSE}})$.

Con la notazione $Val(\psi\ |_{x=k})$ intendiamo la applicazione della procedura *Val* sulla formula ψ, in cui la variabile x è posta uguale a k. Chiaramente, la formula Φ rappresenta un'istanza positiva di FORMULA BOOLEANA QUANTIFICATA se e solo se $Val(\Phi)$ restituisce il valore TRUE.

Si osservi che la profondità massima della ricursione è pari al numero m di variabili, ed è limitata superiormente da n, la dimensione della formula, per cui una pila di n elementi è sufficiente ad implementare la procedura. Si osservi inoltre che ogni elemento di tale pila è sufficiente che rappresenti una

copia della formula modificata per rappresentare le assegnazioni effettuate fino ad ora nell'ambito della valutazione, per una dimensione proporzionale ad n. In definitiva, uno spazio $O(n^2)$ è sufficiente per implementare la procedura *Val*, dal che deriva che FORMULA BOOLEANA QUANTIFICATA $\in$ PSPACE.

Per mostrare la completezza di FORMULA BOOLEANA QUANTIFICATA in PSPACE è ora necessario mostrare che per ogni problema $\mathcal{P} \in$ PSPACE vale la proprietà $\mathcal{P} \leq_m^{logsp}$FORMULA BOOLEANA QUANTIFICATA.

Come nel caso del Teorema di Cook (Teorema 9.4), costruiremo, a partire da una macchina di Turing $\mathcal{M}$[7] e da una stringa di input x, una formula booleana quantificata che è soddisfacibile se e solo se x viene accettata da $\mathcal{M}$ in spazio polinomiale.

Sia $p(n)$ il polinomio che definisce la limitazione di spazio utilizzabile da $\mathcal{M}$ e si osservi che, se x ha lunghezza n, il tableau relativo ad una computazione effettuata da $\mathcal{M}$ può avere $p(n)$ colonne e $c^{p(n)}$ righe per qualche costante c,[8] il che rende ad esempio la costruzione mostrata nella dimostrazione del Teorema 9.4 non applicabile, in quanto la formula CNF risultante avrebbe dimensione esponenziale.

Per brevità, utilizzeremo nel seguito delle variabili $X_1, X_2, \ldots$ che rappresentano codifiche di righe di tableau, effettuate secondo le modalità descritte nella dimostrazione del Teorema 9.4. Vale a dire che ognuna di tali variabili corisponde in realtà a $O(p(n))$ variabili booleane del tipo $S(i,j,k)$ e $C(i,j,k)$. Si assume inoltre che, in corrispondenza all'uso di una variabile X_i di tal tipo, nella formula sia presente una formula $\phi^M(X_i)$, della stessa struttura delle formule ϕ_i^M introdotte nella dimostrazione del Teorema 9.4, che specifica che l'assegnazione effettuata sull'insieme delle variabili $S(i,j,k)$ e $C(i,j,k)$ in X_i rappresenta una configurazione di $\mathcal{M}$.

$$\phi(x) = \exists X_1 \exists X_2 (\phi^M(X_1) \wedge \phi^M(X_2) \wedge \phi^I(X_1, x) \wedge \phi^A(X_2) \wedge \phi^T(X_1, X_2, c^{p(n)}))$$

dove valgono le proprietà seguenti:

- la formula $\phi^I(X_1, x)$ ha la stessa struttura della formula ϕ^I nella dimostrazione del Teorema 9.4, ed è verificata se e solo se si ha una assegnazione sulle variabili booleane in X_1 che codifica la prima riga del tableau con stringa x, e quindi la configurazione iniziale di $\mathcal{M}$ in cui x si trova sul nastro.

- la formula $\phi^A(X_2)$ ha la stessa struttura della formula ϕ^A nella dimostrazione del Teorema 9.4, ed è verificata se e solo se si ha una assegnazione sulle variabili booleane in X_2 che codifica una configurazione accettante di $\mathcal{M}$.

[7] Come nel Teorema 9.4 si assume che $\mathcal{M}$ abbia un solo nastro semi-infinito e che cicli indefinitamente su ogni stato finale.

[8] Nel seguito si assume, senza perdita di generalità, che $c^{p(n)}$ sia una potenza di due, vale a dire che esista un intero k tale che $c^{p(n)} = 2^k$.

- la formula $\phi^T(X_1, X_2, c^{p(n)})$ è soddisfatta da ogni assegnazione che rappresenta una computazione in cui $\mathcal{M}$ passa dalla configurazione X_1 alla configurazione X_2 in al più $c^{p(n)}$ passi.

  Applicando la costruzione utilizzata nella dimostrazione del Teorema 8.26, una formula $\phi^T(X_1, X_2, 2t)$ può essere definita nel modo seguente:

$$\phi^T(X_1, X_2, 2t) = \exists X_3(\phi^M(X_3) \wedge \phi^T(X_1, X_3, t) \wedge \phi^T(X_3, X_2, t)).$$

  Dove $\phi^T(X_1, X_2, 1)$ ha la struttura della formula ϕ^T nella dimostrazione del Teorema 9.4.

  Applicando la definizione precedente, la formula $\phi^T(X_1, X_2, c^{p(n)})$ risulta avere lunghezza esponenziale $O(2^{p(n)})$.

  Consideriamo ora una diversa definizione di $\phi^T(X_1, X_2, 2t)$, che utilizza anche quantificatori universali.

$$\phi^T(X_1, X_2, 2t) = \forall X \forall Y \exists X_3 \left(\phi^M(X_3) \wedge \right.$$

$$\left. \wedge \left[((X = X_1 \wedge Y = X_3) \vee (X = X_3 \wedge Y = X_2)) \Rightarrow \phi^T(X, Y, t)\right]\right).$$

  Come si può vedere, data questa definizione, la lunghezza della formula $\phi^T(X_1, X_2, 2t)$ è pari alla lunghezza della formula $\phi^T(X_1, X_2, t)$ più $O(p(n) \log n)$. Da ciò deriva che la lunghezza di $\phi^T(X_1, X_2, c^{p(n)})$ è $O(p(n) \log n \log c^{p(n)}) = O((p(n))^2 \log n)$, e quindi che la lunghezza della formula $\phi(x)$ è anch'essa $O((p(n))^2 \log n)$.

Si può verificare (vedi Esercizio 9.19) che la formula $\phi(x)$, oltre ad avere lunghezza polinomiale, può essere costruita in spazio logaritmico, e che esiste un'assegnazione di verità che la soddisfa se e solo se $\mathcal{M}$ accetta la stringa di input x. □

Esercizio 9.19 Dimostrare che la formula $\phi(x)$ nella dimostrazione del Teorema 9.21 può essere costruita in spazio logaritmico, e che esiste una assegnazione di verità che la soddisfa se e solo se la macchina di Turing $\mathcal{M}$ accetta la stringa x.

Dal Teorema 9.21 deriva che per ogni $i \geq 1$ $\Sigma_i^p \subseteq$ PSPACE, in quanto ogni problema Formula Booleana Quantificata i-limitata (per ogni $i \geq 1$) è immediatamente Karp-riducibile in spazio logaritmico a Formula Booleana Quantificata, e le classi Σ_i^p ($i \geq 1$) sono chiuse rispetto a tale riducibilità. Da ciò evidentemente deriva che NP = PSPACE implica il collasso dell'intera gerarchia polinomiale.

Esercizio 9.20 Dimostrare che ogni classe Σ_i^p è chiusa rispetto alla Karp-riducibilità *log*-space.

Non è difficile rendersi conto che la classe PSPACE è chiusa rispetto alla complementazione, vale a dire che PSPACE=co-PSPACE. Ciò risulta evidente applicando ad esempio le stesse considerazioni effettuate per mostrare che P=co-P, vale a dire osservando che per risolvere un problema complementare di un problema $\mathcal{P} \in$ PSPACE basta cambiare l'algoritmo per $\mathcal{P}$ in modo da scambiare risposte VERO con risposte FALSO.

Numerosi problemi interessanti risultano PSPACE-completi: la loro completezza può essere mostrata, come nel caso dei problemi NP-completi considerati in precedenza, mostrando che essi sono in PSPACE e individuando una Karp-riduzione *log*-space da un problema PSPACE-completo ad essi. Tra questi possiamo citare il problema GEOGRAFIA introdotto qui di seguito.

Definiamo il seguente gioco *Geografia* da effettuarsi tra due giocatori A e B su un grafo orientato $G = (V, A)$. I giocatori scelgono alternativamente archi da A, con il giocatore A che sceglie inizialmente un arco uscente dal nodo predefinito v_0: ad ogni turno, un giocatore deve scegliere un arco $\langle V_i, V_j \rangle$, se l'avversario al turno precedente ha scelto un arco $\langle V_k, V_i \rangle$. Il primo giocatore che non ha archi da scegliere perde.

GEOGRAFIA

ISTANZA: Grafo orientato $G = (V, A)$, vertice $v_0 \in V$

PREDICATO: Il giocatore A ha una strategia vincente nel gioco *Geografia* eseguito sul grafo G a partire dal nodo v_0?

Al fine di mostrare la PSPACE-completezza di GEOGRAFIA dimostriamo prima il seguente lemma.

Lemma 9.22 *Per ogni istanza* Φ *di* FORMULA BOOLEANA QUANTIFICATA *esiste una istanza equivalente* $\overline{\Phi}$ *con le seguenti caratteristiche:*

1. *la formula* $\overline{\mathcal{F}}$ *di* $\overline{\Phi}$ *è in CNF;*
2. *il primo quantificatore di* $\overline{\Phi}$ *è un quantificatore esistenziale;*
3. *l'ultimo quantificatore di* $\overline{\Phi}$ *è anch'esso un quantificatore esistenziale;*
4. *i quantificatori si alternano, nel senso che , per ogni* $i < n$, *se* $Q_i = \exists$ *allora* $Q_{i+1} = \forall$ *e se* $Q_i = \forall$ *allora* $Q_{i+1} = \exists$*;*
5. *l'istanza* $\overline{\Phi}$ *ha dimensione polinomiale nella dimensione dell'istanza* Φ*.*

Dimostrazione. Sia $\Phi(x_1, \ldots, x_n) = Q_1 x_1 \ldots Q_n x_n \mathcal{F}(x_1, \ldots, x_n)$ una istanza di FORMULA BOOLEANA QUANTIFICATA. A partire da Φ possiamo costruire una istanza Φ' avente la proprietà che la formula $\mathcal{F}'$ è in CNF nel modo

seguente: introduciamo una nuova variabile z_i, quantificata in modo esistenziale, per rappresentare ogni sottoespressione in $\mathcal{F}$, e definiamo $\mathcal{F}'$ per mezzo della composizione delle variabili z_i (vedi Esempio 9.8).

A partire da

$$\Phi'(z_1, \ldots, z_m, x_1, \ldots, x_n) = Q_1 z_1 \ldots Q_{m+n} x_n \mathcal{F}'(z_1, \ldots, z_m, x_1, \ldots, x_n)$$

costruiamo una istanza Φ'', avente le prime tre caratteristiche nell'enunciato, introducendo opportune variabili *dummy*, che non sono presenti nella formula $\mathcal{F}$, nel modo seguente:

- se $Q_1 = \exists$ e $Q_{m+n} = \exists$ allora $\Phi'' = \Phi'$;
- se $Q_1 = \forall$ e $Q_m = \exists$ allora poniamo

$$\Phi'' = \exists y_0 Q_1 z_1 \ldots Q_{m+n} x_n \mathcal{F}'(z_1, \ldots, z_m, x_1, \ldots, x_n);$$

- se $Q_1 = \exists$ e $Q_m = \forall$ allora poniamo

$$\Phi'' = Q_1 z_1 \ldots Q_{m+n} x_n \exists y_1 \mathcal{F}'(z_1, \ldots, z_m, x_1, \ldots, x_n);$$

- se $Q_1 = \forall$ e $Q_m = \forall$ allora poniamo

$$\Phi'' = \exists y_0 Q_1 z_1 \ldots Q_{m+n} x_n \exists y_1 \mathcal{F}'(z_1, \ldots, z_m, x_1, \ldots, x_n).$$

Al fine di ottenere, a partire da Φ'', l'istanza $\overline{\Phi}$ avente i quantificatori alternanti, possiamo procedere nel modo seguente: per ogni coppia Q_i, Q_{i+1} di quantificatori successivi dello stesso tipo introduciamo ua variabile *dummy* ed un quantificatore di tipo opposto da inserire tra Q_i e Q_{i+1}. Se ad esempio $Q_i = \forall$ e $Q_{i+1} = \forall$, la sequenza di quantificatori in Φ'' sarà $(\ldots Q_i x_i \exists y_i Q_{i+1} x_{i+1} \ldots)$: in questo modo, come si può osservare, $\mathcal{F}'' = \mathcal{F}$ e la sequenza dei quantificatori di Φ'' ha lunghezza al più doppia rispetto a quella di Φ.

È possibile verificare facilmente (vedi Esercizio 9.21) che l'istanza $\overline{\Phi}$ ha dimensione polinomiale nella dimensione di Φ. □

Esempio 9.8 Si consideri l'istanza di FORMULA BOOLEANA QUANTIFICATA rappresentata dalla formula

$$\forall x_1 \forall x_2 \exists x_3 [((x_1 \wedge \overline{x}_3) \vee x_2) \wedge x_3].$$

Definiamo la variabile z_1 e poniamo $\equiv (x_1 \wedge \overline{x}_3)$, ottenendo la formula

$$\exists z_1 \forall x_1 \forall x_2 \exists x_3 [(z_1 \equiv (x_1 \wedge \overline{x}_3)) \wedge (z_1 \vee x_2) \wedge x_3].$$

Definiamo ora la variabile z_2 e poniamo $z_2 \equiv (z_1 \vee x_2)$, ottenendo la formula

$$\exists z_1 \exists z_2 \forall x_1 \forall x_2 \exists x_3 [(z_1 \equiv (x_1 \wedge \overline{x}_3)) \wedge (z_2 \equiv (z_1 \vee x_2)) \wedge z_2 \wedge x_3].$$

Osserviamo ora che la formula $z_1 \equiv (x_1 \wedge \overline{x}_3))$ può essere scritta anche come

$$(\overline{x}_1 \vee x_3 \vee z_1) \wedge (x_1 \vee \overline{z}_1) \wedge (\overline{x}_3 \vee \overline{z}_1),$$

e che $z_2 \equiv (z_1 \vee x_2)$ può a sua volta essere scritta come

$$(\overline{z}_1 \vee \overline{z}_2 \vee x_2) \wedge (z_2 \vee \overline{z}_1) \wedge (\overline{x}_2 \vee z_2).$$

Da ciò otteniamo la formula Φ' definita come:

$$\exists z_1 \exists z_2 \forall x_1 \forall x_2 \exists x_3 [(\overline{x}_1 \vee x_3 \vee z_1) \wedge (x_1 \vee \overline{z}_1) \wedge (\overline{x}_3 \vee \overline{z}_1) \wedge$$
$$\wedge (\overline{z}_1 \vee \overline{z}_2 \vee x_2) \wedge (z_2 \vee \overline{z}_1) \wedge (\overline{x}_2 \vee z_2) \wedge z_2 \wedge x_3].$$

Applicando le altre trasformazioni descritte nella dimostrazione del Lemma 9.22, otteniamo quindi l'istanza $\overline{\Phi}$ voluta

$$\exists z_1 \forall y_1 \exists z_2 \forall x_1 \exists y_2 \forall x_2 \exists x_3 [(\overline{x}_1 \vee x_3 \vee z_1) \wedge (x_1 \vee \overline{z}_1) \wedge (\overline{x}_3 \vee \overline{z}_1) \wedge$$
$$\wedge (\overline{z}_1 \vee \overline{z}_2 \vee x_2) \wedge (z_2 \vee \overline{z}_1) \wedge (\overline{x}_2 \vee z_2) \wedge z_2 \wedge x_3].$$

Esercizio 9.21 Verificare che l'istanza $\overline{\Phi}$ ottenuta applicando il procedimento descritto nella dimostrazione del Lemma 9.22 ha in effetti dimensione polinomiale nella dimensione dell'istanza originaria Φ.

Passiamo ora a dimostrare che GEOGRAFIA è un problema completo in PSPACE.

Teorema 9.23 *Il problema* GEOGRAFIA *è* PSPACE*-completo rispetto alla Karp–riducibilità log-space.*

Dimostrazione. Mostriamo dapprima che GEOGRAFIA $\in$ PSPACE: per verificare che tale relazione è vera osserviamo che ogni partita di *Geografia* ha lunghezza polinomiale. Consideriamo inoltre l'albero di gioco associato ad una istanza di GEOGRAFIA: in tale albero ogni nodo è associato ad una posizione nella partita, con la radice associata alla posizione iniziale, ed ogni arco (u, v) corrisponde ad una mossa che viene effettuata nella posizione associata ad u e porta alla posizione associata a v. Gli archi uscenti dalla radice rappresentano possibili mosse di A, gli archi del livello sottostante possibili mosse di B, e così via. Dato che una partita corrisponde ad un cammino nell'albero, ed ha lunghezza polinomiale, ne deriva che l'altezza dell'albero è polinomiale.

Non è difficile rendersi conto (vedi Esercizio 9.22) che per verificare se il giocatore A ha una strategia vincente è sufficiente effettuare una opportuna visita in profondità dell'intero albero di gioco, e che l'effettuazione di tale visita richiede spazio polinomiale.

Per mostrare la completezza di GEOGRAFIA in PSPACE, dimostriamo ora che FORMULA BOOLEANA QUANTIFICATA $\leq_m^{logsp}$ GEOGRAFIA. A tal fine, assumiamo di avere una istanza di $\Phi = \exists x_1 \forall x_2 \ldots \exists x_n \mathcal{F}(x_1, x_2, \ldots, x_n)$ di FORMULA BOOLEANA QUANTIFICATA nella forma enunciata nel Lemma 9.22, vale a dire tale che:

1. il primo quantificatore Q_1 è un quantificatore esistenziale $\exists$;
2. l'ultimo quantificatore Q_n è anch'esso un quantificatore esistenziale $\exists$;
3. i quantificatori si alternano;
4. la formula non quantificata $\mathcal{F}$ è in CNF.

Per il Lemma 9.22, le ipotesi precedenti non limitano l'applicabilità della dimostrazione.

Sia m il numero di clausole in $\mathcal{F}$. A partire da Φ costruiamo una istanza di GEOGRAFIA nel modo seguente. Associamo ad ogni variabile x_i in Φ un grafo V_i di quattro nodi n_i, s_i, e_i, w_i e quattro archi $< n_i, e_i >$,$< n_i, w_i >$,$< e_i, s_i >$,$< w_i, s_i >$: tali grafi sono composti in una sequenza mediante archi $< s_i, n_{i+1} >$.

Alla j-esima clausola $t_{j,1} \vee t_{j,2} \vee \ldots \vee t_{j,n_j}$ di $\mathcal{F}$ associamo un grafo C_j di n_j+1 nodi $c_j, t_{j,1}, \ldots, t_{j,n_j}$ e n_j archi $< c_j, t_{j,1} >, < c_j, t_{j,2} >, \ldots, < c_j, t_{j,n_j} >$.

Nell'istanza di GEOGRAFIA il nodo s_n è collegato ad un nodo r, dal quale a sua volta esce un arco verso ogni nodo c_j, $1 \leq j \leq m$. Inoltre, se $t_{j,i} = x_k$ esiste un arco $< c_{j,i}, w_k >$, mentre se $t_{j,i} = \overline{x}_k$ esiste un arco $< c_{j,i}, e_k >$.

Sia $n_1 = v_0$ il nodo iniziale del gioco. Per la struttura del grafo risultante, la partita procede nel modo seguente: il giocatore A inizialmente sceglie uno tra i due archi $< n_0, e_1 >$, $< n_0, w_1 >$, scegliendo così un valore VERO (nel primo caso), FALSO (nel secondo caso) per la variabile x_1. La stessa scelta viene quindi effettuata da B per la seconda variabile, e così via, fino alla scelta effettuata da A per la variabile x_n. In tal modo, viene costruita una assegnazione di verità sulle variabili $x_1, \ldots, x_n$.

La scelta dell'arco uscente dal nodo r è effettuata dal giocatore B, il quale in tal modo può selezionare una possibile clausola c_j falsa secondo l'assegnazione costruita. A questo punto, il giocatore A può selezionare un termine $t_{j,i}$ di tale clausola. Per costruzione, se tale termine è vero nell'assegnazione di verità allora il giocatore B non ha archi da scegliere, ed A vince la partita. Al contrario, se il termine selezionato da A non è vero nell'assegnazione di verità, allora B può scegliere l'unico arco uscente dal nodo associato a tale termine, mentre A successivamente non ha possibilità di scelta, per cui B vince la partita.

Da quanto detto, risulta semplice verificare sia che A ha una strategia vincente per l'istanza di GEOGRAFIA se e solo se la formula nell'istanza di Φ è vera, sia che la riduzione mostrata può essere effettuata in spazio logaritmico. □

Esempio 9.9 Consideriamo l'istanza $\exists x_1 \forall x_2 \exists x_3 [(x_1 \vee \overline{x}_2) \wedge (x_1 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \vee \overline{x}_2]$.

Applicando la riduzione mostrata nella dimostrazione del Teorema 9.23 si ottiene l'istanza di GEOGRAFIA in Figura9.13.

Esercizio 9.22 Mostrare come sia possibile verificare se il giocatore A ha una strategia vincente in *Geografia* effettuando una visita in profondità del relativo albero di gioco.

Esercizio 9.23 Dimostrare che la riduzione illustrata nella dimostrazione del Teorema 9.23 richiede spazio logaritmico.

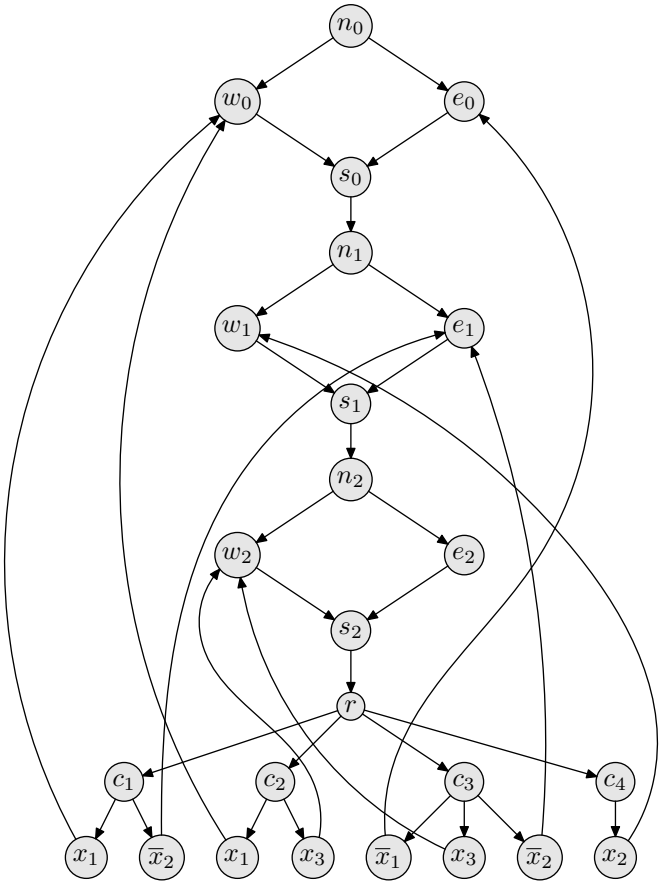


FIGURA 9.13 Istanza di GEOGRAFIA risultante dall'Esempio 9.23

Un ulteriore esempio di problema PSPACE-completo, problema originato nell'ambito della teoria dei linguaggi formali, è il seguente.

COMPLETEZZA DI ESPRESSIONI REGOLARI

ISTANZA: Espressione regolare ρ sugli operatori $+$, $\cdot$, $*$ e su un alfabeto Σ

PREDICATO: Denominato con $L(\rho)$ il linguaggio descritto da ρ, è vero che $L(\rho) = \Sigma^*$?

Teorema 9.24 *Il problema* COMPLETEZZA DI ESPRESSIONI REGOLARI *è* PSPACE*-completo rispetto alla Karp–riducibilità log-space.*

Dimostrazione. La dimostrazione è lasciata come esercizio (vedi Esercizio 9.24). □

Esercizio 9.24 Dimostrare la PSPACE-completezza del problema COMPLETEZZA DI ESPRESSIONI REGOLARI.

Indice analitico