Pilu Crescenzi

Informatica Teorica

Firenze, Settembre 2011

A Paola, Nicole e Giorgia

A uno a uno varcarono la soglia della porticina da fiaba
ed entrarono nella tana di It.
Stephen King

Sommario

Prefazione

QUESTE DISPENSE sono dedicate allo studio della teoria della calcolabilità, della teoria dei linguaggi formali e della teoria della complessità. In particolare, l'obiettivo principale delle dispense è quello di cercare di rispondere alla seguente domanda.

> COSA PUÒ ESSERE CALCOLATO DA UN CALCOLATORE E QUANTO COSTA FARLO?

Tale domanda sorse ben prima che fosse disponibile un calcolatore reale (almeno come viene inteso ai giorni nostri): più precisamente, essa risale agli anni trenta, quando diversi ricercatori nel campo della logica matematica si chiesero quale fosse il vero significato della parola "calcolabile" e quali fossero gli eventuali limiti del corrispondente modello di calcolo. Fu così che, in quegli anni, furono proposti e analizzati diversi modelli di calcolo, quali il λ-calcolo, le **macchine di Turing**, gli algoritmi di Markov, le macchine di Post e le funzioni ricorsive, e nacque la **teoria della calcolabilità**. Per quanto tutti questi modelli furono proposti ciascuno più o meno indipendentemente dall'altro, fu quasi immediatamente dimostrato che essi erano tutti equivalenti tra di loro, nel senso che tutto ciò che poteva essere calcolato in un dato modello, poteva essere calcolato in un qualunque altro modello. Questi risultati, oltre a confermare il fatto di muoversi nella giusta direzione, portarono a formulare quella che oggi è universalmente nota come la **tesi di Church-Turing** che sostanzialmente afferma quanto segue.

> È CALCOLABILE TUTTO CIÒ CHE PUÒ ESSERE CALCOLATO DA UNA MACCHINA DI TURING.

Per quanto detto in precedenza, nella formulazione della tesi di Church-Turing avremmo potuto sostituire "una macchina di Turing" con, ad esempio, "un algoritmo di Markov": abbiamo preferito formulare la tesi nel modo sopra esposto, in quanto

in queste dispense faremo principalmente riferimento alle macchine di Turing come modello di calcolo. Il motivo di questa scelta sta nel fatto che una macchina di Turing è probabilmente il modello di calcolo più simile agli odierni calcolatori e, quindi, più naturale da utilizzare per chi conosce l'architettura interna di un calcolatore reale (composto, essenzialmente, da un'unità di elaborazione centrale, da una memoria e da alcuni dispositivi di ingresso e uscita), e per chi è abituato a considerare un calcolatore come uno strumento di manipolazione di simboli (ovvero, di dati) mediante programmi, che a loro volta non sono altro che sequenze di simboli. A dispetto della sua semplicità, non è dunque sorprendente che il modello delle macchine di Turing sia estremamente potente dal punto di vista computazionale. A dimostrazione di ciò, nella prima parte delle dispense vedremo come sia possibile definire una **macchina di Turing universale**, ovvero una macchina di Turing in grado di simulare ogni altra macchina di Turing (in altre parole, quello che oggi viene comunemente chiamato un calcolatore *general-purpose*). Una volta stabilito il concetto di calcolabilità, è stato naturale chiedersi, allo stesso tempo, che cosa non fosse calcolabile. In altre parole, ci si è chiesti quali fossero i limiti di un modello di calcolo quale le macchine di Turing (a dire il vero, non sembra esagerato affermare che questa domanda fosse il vero motivo della definizione dei diversi modelli di calcolo). Sulla scia di uno dei più famosi risultati della logica matematica, ovvero il teorema di Gödel, è stato possibile stabilire quanto segue.

> ESISTONO PROBLEMI CHE NON POSSONO ESSERE RISOLTI IN MODO ALGORITMICO.

Ad esempio, vedremo come ciò sia vero nel caso del **problema della terminazione**, che consiste nel decidere se una specifica macchina di Turing con input una specifica sequenza di simboli termini la sua computazione oppure no. A partire da tale problema, potremo dimostrare la non calcolabilità di tanti altri problemi facendo uso della tecnica di **riduzione** e della tecnica di **diagonalizzazione**, la cui applicabilità sarà semplificata dal **teorema della ricorsione**, il quale, come vedremo, può essere interpretato come la dimostrazione formale e costruttiva dell'esistenza dei virus informatici.

Allo scopo di capire il modo in cui l'essere umano apprende e usa le proprie capacità linguistiche, altri ricercatori diedero il via negli anni cinquanta alla classificazione dei linguaggi in termini di grammatiche generative e di modelli di calcolo, mostrando come esistano fondamentalmente quattro classi di linguaggi (regolari, contestuali, liberi da contesto e di tipo 0) a ciascuna delle quali corrisponde uno specifico modello di calcolo (automi a stati finiti, automi a pila non deterministici, macchine di Turing lineari e macchine di Turing). Questi sono gli argomenti principali della **teoria dei linguaggi formali**, di cui tratteremo nella seconda parte di

queste dispense in cui introdurremo la **gerarchia di Chomski** per poi dimostrare la sua caratterizzazione in termini di modelli di calcolo.

Oltre a chiedersi che cosa potesse essere calcolato, i ricercatori iniziarono anche a domandarsi quanto fosse costoso (in termini di tempo e di memoria) eseguire determinati calcoli: recentemente, si è scoperto che una tale domanda fu posta già negli anni quaranta da Gödel in una lettera indirizzata a von Neumann. In effetti, nessun utente di un calcolatore è disposto ad aspettare un secolo per ottenere la risposta a una domanda posta al calcolatore, né è disposto a comprare una memoria da 1000 Gigabit per permettere al calcolatore di essere in grado di formulare tale risposta. Per questo motivo, nacque la **teoria della complessità** di cui tratteremo nella terza e ultima parte delle dispense. Tale teoria vide la sua forma attuale verso la metà degli anni settanta, quando fu implicitamente formulata quella che è probabilmente la questione aperta più importante nel campo dell'informatica teorica, ovvero la **congettura** $\text{P} \neq \text{NP}$, la quale afferma quanto segue.

> ESISTONO PROBLEMI CHE NON POSSONO ESSERE RISOLTI IN TEMPO POLINOMIALE, MA PER I QUALI È POSSIBILE VERIFICARE IN TEMPO POLINOMIALE LA CORRETTEZZA DI UNA SOLUZIONE.

Nonostante centinaia di ricercatori abbiamo tentato di dimostrare tale congettura, quest'ultima è all'inizio del terzo millennio ancora irrisolta: per dare al lettore un'idea della sua significatività, basti pensare che una "taglia" di un milione di dollari è stata posta sulla sua dimostrazione e che essa appare tra i dieci più importanti problemi matematici del secolo scorso.

Le dispense sono organizzate in tre parti corrispondenti, rispettivamente, alla trattazione della teoria della calcolabilità (Capitoli 1, 2, 3 e 4), della teoria dei linguaggi formali (Capitoli 5, 6, 7 e 8) e della teoria della complessità computazionale (Capitoli 9, 10 e 11). La prima e la seconda parte faranno riferimento all'applicazione JFLAP, che consente di costruire, eseguire e animare automi a stati finiti, automi a pila e macchine di Turing. Ogni capitolo è completato da diversi esercizi.

Le dispense non presuppongono particolari conoscenze da parte dello studente, a parte quelle relative a nozioni matematiche e logiche di base. A differenza di molti altri volumi, che relegano i necessari requisiti matematici nel capitolo iniziale oppure in un'appendice, in queste dispense si è scelto di includere i requisiti matematici, come quello relativo agli insiemi incluso in questa prefazione, al momento in cui per la prima volta essi si rendano necessari.

Esistono numerosi libri di testo dedicati agli argomenti trattati in queste dispense: in particolare, i seguenti due volumi hanno ispirato in parte la stesura delle dispense stesse e sono fortemente consigliati come letture aggiuntive.

Insiemi

Un **insieme** è una qualunque collezione di elementi. Se a, b e c sono tre elementi generici, allora l'insieme A formato dagli elementi a, b e c è rappresentato come $A = \{a, b, c\}$. Un insieme non può contenere più di una copia dello stesso elemento; inoltre, l'ordine con cui gli elementi sono elencati è irrilevante. Diremo che due insiemi A e B sono **uguali** (in simboli, $A = B$) se ogni elemento di A è anche un elemento di B e viceversa. Due insiemi A e B sono **diversi** (in simboli, $A \neq B$) se non è vero che $A = B$. I simboli $\in$ e $\notin$ denotano, rispettivamente, il fatto che un elemento appartiene o non appartiene a un insieme. Facendo riferimento all'esempio precedente, è chiaro che $a \in A$ e che $d \notin A$. Un insieme X è **vuoto** (in simboli, $X = \emptyset$) se non include alcun elemento, altrimenti è **non vuoto**. Un insieme X è **finito** se include un numero finito di elementi, altrimenti è **infinito**. Tipici esempi di insiemi infiniti sono gli insiemi dei **numeri naturali** $\mathbb{N}$, dei **numeri interi** $\mathbb{Z}$ e dei **numeri reali** $\mathbb{R}$. Poiché un insieme infinito non può essere descritto elencando tutti i suoi elementi, a tale scopo si usano altri approcci: uno di questi consiste nel definire nuovi insiemi sulla base di altri insiemi che si assume siano noti. Più precisamente, dato un insieme X e una proprietà π, indichiamo con $Y = \{y : y \in X \text{ e } y \text{ soddisfa } \pi\}$ l'insieme formato da tutti gli elementi y che appartengono a X e che soddisfano la proprietà π: ad esempio, $A = \{x : x \in \mathbb{N} \text{ e } (x \equiv 0 \pmod 3)\}$ denota l'insiemi dei numeri naturali che sono multipli di 3. Un insieme A è un **sottoinsieme** di un insieme B (in simboli, $A \subseteq B$) se ogni elemento di A è anche un elemento di B. Un insieme A è un **sottoinsieme proprio** di un insieme B (in simboli, $A \subset B$) se $A \subseteq B$ e $A \neq B$. Se $A = B$, allora è vero sia $A \subseteq B$ che $B \subseteq A$. Dati due insiemi A e B, la loro **unione**, indicata con $A \cup B$, è l'insieme di tutti gli elementi che appartengono ad A oppure a B; la loro **intersezione**, indicata con $A \cap B$, è l'insieme di tutti gli elementi che appartengono sia ad A che a B. Due insiemi sono **disgiunti** se non hanno alcun elemento in comune, ovvero se $A \cap B = \emptyset$. Dati due insiemi A e B, la loro **differenza**, indicata con $A - B$, è l'insieme di tutti gli elementi che appartengono ad A ma non appartengono a B, ovvero $A - B = \{x : x \in A \text{ e } x \notin B\}$. Dati n insiemi $A_1, A_2, \ldots, A_n$, il loro **prodotto cartesiano**, indicato con $A_1 \times A_2 \times \cdots \times A_n$, è l'insieme di tutte le n-tuple $(a_1, \ldots, a_n)$ per cui $a_i \in A_i$ per ogni i con $1 \leqslant i \leqslant n$ (una 2-tupla è anche detta coppia mentre una 3-tupla è anche detta tripla).

- G. Ausiello, F. D'Amore, G. Gambosi. *Linguaggi, modelli, complessità*. Edizioni Franco Angeli, 2003.
- M. Sipser. *Introduction to the Theory of Computation*. Thomson CT, 2005.

A chi poi volesse approfondire le nozioni relative alla teoria della complessità, suggeriamo la lettura del libro *Introduction to the Theory of Complexity*, scritto da D. P. Bovet e P. Crescenzi, che è disponibile gratuitamente a partire dal sito web del secondo autore.

Ringraziamenti Desidero ringraziare Marco Bartolozzi, Leonardo Cecchi, Simone Falcini, Lorenzo Frilli, Pierluigi Lai, Thomas Linford, Alessandro Meoni, Lorenzo Motti e Michela Picinotti per aver segnalato alcuni errori presenti nelle precedenti versioni di queste dispense.

Parte I

Teoria della calcolabilità

Capitolo 1

Macchine di Turing

SOMMARIO

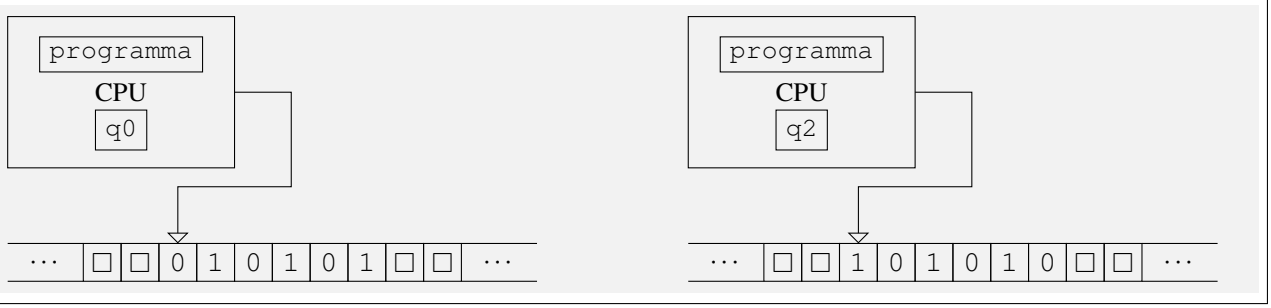
In questo capitolo introdurremo il modello di calcolo proposto dal logico matematico inglese Alan Turing. Una volta definita la macchina di Turing, mostreremo diversi esempi di tali macchine introducendo due tecniche di programmazione comunemente utilizzate per sviluppare macchine di Turing. Definiremo, quindi, le macchine di Turing multi-nastro dimostrando che non sono computazionalmente più potenti di quelle con un singolo nastro e concluderemo il capitolo introducendo il concetto di configurazione di una macchina di Turing e quello di sotto-macchina.

1.1 Definizione di macchina di Turing

UNA MACCHINA di Turing è un modello di calcolo abbastanza simile agli odierni calcolatori. Analogamente a questi ultimi, infatti, essa possiede un'unità di elaborazione centrale (CPU, dall'inglese *Central Processing Unit*) e una memoria su cui poter leggere e scrivere. In particolare, la CPU di una macchina di Turing è composta da un **registro di stato**, contenente lo stato attuale della macchina, e da un **programma** contenente le istruzioni che essa deve eseguire. La memoria di una macchina di Turing è composta da un **nastro** infinito, suddiviso in **celle** e al quale la CPU può accedere attraverso una **testina** di lettura/scrittura (si veda la Figura 1.1).

Inizialmente, il nastro contiene la stringa di **input** preceduta e seguita da una serie infinita di simboli vuoti (in queste dispense, il **simbolo vuoto** è indicato con □), la testina è posizionata sul primo simbolo della stringa di input e la CPU si trova in uno stato speciale, detto **stato iniziale** (si veda la parte sinistra della Figura 1.1). Sulla base dello stato in cui si trova la CPU e del simbolo letto dalla testina, la macchina esegue un'istruzione del programma che può modificare il simbolo attualmente scandito dalla testina, spostare la testina a destra oppure a sinistra e cambiare lo stato della

Figura 1.1: rappresentazione schematica di una macchina di Turing.



CPU. La macchina prosegue nell'esecuzione del programma fino a quando la CPU non viene a trovarsi in uno di un insieme di stati particolari, detti **stati finali**, oppure non esistono istruzioni del programma che sia possibile eseguire. Nel caso in cui il programma termini perché la CPU ha raggiunto uno stato finale, il contenuto della porzione di nastro racchiusa tra la posizione della testina ed il primo simbolo □ alla sua destra rappresenta la stringa di **output** (si veda la parte destra della Figura 1.1).

Esempio 1.1: una macchina di Turing per il complemento bit a bit

Consideriamo una macchina di Turing la cui CPU può assumere tre possibili stati `q0`, `q1` e `q2`, di cui il primo è lo stato iniziale e l'ultimo è l'unico stato finale. L'obiettivo della macchina è quello di calcolare la stringa binaria ottenuta eseguendo il complemento bit a bit della stringa binaria ricevuta in input: il programma di tale macchina è il seguente.

1. Se lo stato è `q0` e il simbolo letto non è □, allora complementa il simbolo letto e sposta la testina a destra.
2. Se lo stato è `q0` e il simbolo letto è □, allora passa allo stato `q1` e sposta la testina a sinistra.
3. Se lo stato è `q1` e il simbolo letto non è □, allora sposta la testina a sinistra.
4. Se lo stato è `q1` e il simbolo letto è □, allora passa allo stato `q2` e sposta la testina a destra.

La prima istruzione fa sì che la macchina scorra l'intera stringa di input, complementando ciascun bit incontrato, mentre le successive tre istruzioni permettono di riportare la testina all'inizio della stringa modificata. Ad esempio, tale macchina, con input la stringa binaria `010101`, inizia la computazione nella configurazione mostrata nella parte sinistra della Figura 1.1 e termina la sua esecuzione nella configurazione mostrata nella parte destra della figura.

Alfabeti, stringhe e linguaggi

Un **alfabeto** è un qualunque insieme finito non vuoto $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$: un **simbolo** è un elemento di un alfabeto. Una **stringa** su un alfabeto Σ è una sequenza finita $x = \sigma_{i_1}, \ldots, \sigma_{i_n}$ di simboli in Σ (per semplicità, la stringa $\sigma_{i_1}, \ldots, \sigma_{i_n}$ sarà indicata con $\sigma_{i_1} \cdots \sigma_{i_n}$): la **stringa vuota** è indicata con λ. L'insieme infinito di tutte le possibili stringhe su un alfabeto Σ è indicato con Σ^*. La **lunghezza** $|x|$ di una stringa $\sigma_{i_1} \cdots \sigma_{i_n}$ è il numero n di simboli contenuti in x: la stringa vuota ha lunghezza 0. Chiaramente, il numero di stringhe di lunghezza n su un alfabeto di k simboli è uguale a k^n. Date due stringhe x e y, la **concatenazione** di x e y (in simboli, xy) è definita come la stringa z formata da tutti i simboli di x seguiti da tutti i simboli di y: quindi, $|z| = |x| + |y|$. In particolare, la concatenazione di una stringa x con se stessa h volte è indicata con x^h. Date due stringhe x e y, diremo che x è un **prefisso** di y se esiste una stringa z per cui $y = xz$. Dato un alfabeto Σ, un **linguaggio** L su Σ è un sottoinsieme di Σ^*: il complemento di L, in simboli L^c, è definito come $L^c = \Sigma^* - L$. Dato un alfabeto Σ, un ordinamento dei simboli di Σ induce un ordinamento delle stringhe su Σ nel modo seguente: (a) per ogni $n \geqslant 0$, le stringhe di lunghezza n precedono quelle di lunghezza $n+1$, e (b) per ogni lunghezza, l'ordine è quello alfabetico. Tale ordinamento è detto **ordinamento lessicografico**. Ad esempio, dato l'alfabeto binario $\Sigma = \{0, 1\}$, le prime dieci stringhe nell'ordinamento lessicografico di Σ^* sono λ, 0, 1, 00, 01, 10, 11, 000, 001 e 010.

Formalmente, una macchina di Turing è definita specificando l'insieme degli stati (indicando quale tra di essi è quello iniziale e quali sono quelli finali) e l'insieme delle transizioni da uno stato a un altro: questo ci conduce alla seguente definizione.

Definizione 1.1: macchina di Turing

Una *macchina di Turing* è costituita da un **alfabeto di lavoro** Σ contenente il simbolo $\square$ e da un **grafo delle transizioni**, ovvero un grafo etichettato $G = (V, E)$ tale che:

- $V = \{q_0\} \cup F \cup Q$ è l'insieme degli **stati** (q_0 è lo stato **iniziale**, F è l'insieme degli stati **finali** e Q è l'insieme degli stati che non sono iniziali né finali);
- E è l'insieme delle **transizioni** e, a ogni transizione, è associata un'etichetta formata da una lista l di triple (σ, τ, m), in cui σ e τ sono simboli appartenenti a Σ e $m \in \{R, L, S\}$, tale che non esistono due triple in l con lo stesso primo elemento.

I simboli σ e τ che appaiono all'interno di una tripla dell'etichetta di una transizione indicano, rispettivamente, il simbolo attualmente letto dalla testina e il simbolo da scrivere, mentre il valore m specifica il movimento della testina: in particolare, R corrisponde allo spostamento a destra, L allo spostamento a sinistra e S a nessun spostamento. Nel seguito di queste dispense, per evitare di dover specificare ogni volta quale sia lo stato iniziale e quali siano gli stati finali di una macchina di Turing e in

Grafi

Un **grafo** G è una coppia di insiemi finiti (V, E) tale che $E \subseteq V \times V$: V è l'insieme dei **nodi**, mentre E è l'insieme degli **archi**. Se $(u, v) \in E$, allora diremo che u è **collegato** a v: il numero di nodi a cui un nodo x è collegato è detto **grado in uscita** di x, mentre il numero di nodi collegati a x è detto **grado in ingresso** di x. Un grafo $G' = (V', E')$ è un **sotto-grafo** di un grafo $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$. Un grafo **etichettato** è un grafo $G = (V, E)$ con associata una funzione $l : E \to L$ che fa corrispondere a ogni arco un elemento dell'insieme L delle etichette: tale insieme può essere, ad esempio, un insieme di valori numerici oppure un linguaggio. Un grafo $G = (V, E)$ è detto **completo** se $E = V \times V$, ovvero se ogni nodo è collegato a ogni altro nodo. Dato un grafo G e due nodi v_0 e v_k, un **cammino** da v_0 a v_k di lunghezza k è una sequenza di archi (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k) tale che, per ogni i e j con $0 \leqslant i < j \leqslant k$, $v_i \neq v_j$ se $i \neq 0$ o $j \neq k$: se $v_0 = v_k$ allora la sequenza di archi è detta essere un **ciclo**. Un **albero** è un grafo senza cicli.

accordo con l'applicativo JFLAP, adotteremo la convenzione di specificare lo stato iniziale affiancando a esso una freccia rivolta verso destra e di rappresentare gli stati finali con un doppio cerchio.

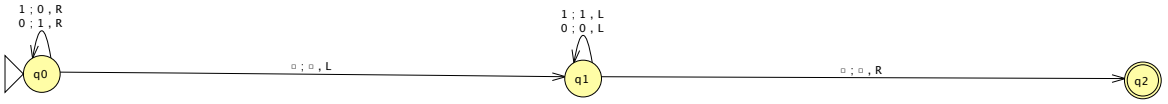
Esempio 1.2: il grafo delle transizioni della macchina per il complemento bit a bit

Il grafo corrispondente alla macchina di Turing introdotta nell'Esempio 1.1 è mostrato nella Figura 1.2. Come si può vedere, ciascuna transizione corrisponde a un'istruzione del programma descritto nell'Esempio 1.1. Ad esempio, l'arco che congiunge lo stato `q0` allo stato `q1` corrisponde alla seconda istruzione: in effetti, nel caso in cui lo stato attuale sia `q0` e il simbolo letto sia □, allora il simbolo non deve essere modificato, la testina si deve spostare a sinistra e la CPU deve passare nello stato `q1`.

1.1.1 Rappresentazione tabellare di una macchina di Turing

Oltre alla rappresentazione basata sul grafo delle transizioni a cui abbiamo fatto riferimento nella Definizione 1.1, una macchina di Turing viene spesso specificata facendo uso di una rappresentazione tabellare. In base a tale rappresentazione, a una macchina di Turing viene associata una tabella di tante righe quante sono le triple contenute nelle etichette degli archi del grafo delle transizioni corrispondente alla macchina. Se (σ, τ, m) è una tripla contenuta nell'etichetta dell'arco che collega lo stato q allo stato p, la corrispondente riga della tabella sarà formata da cinque colonne, contenenti, rispettivamente, q, σ, p, τ e m. In altre parole, ogni riga della tabella corrisponde a un'istruzione del programma della macchina di Turing: la prima e la seconda colonna della riga specificano, rispettivamente, lo stato in cui si deve trovare la CPU e il simbolo che deve essere letto dalla testina affinché l'istruzione sia eseguita, mentre la terza, la quarta e la quinta colonna specificano, rispettivamente,

Figura 1.2: macchina di Turing per il complemento bit a bit.



il nuovo stato in cui si troverà la CPU, il simbolo che sostituirà quello appena letto e il movimento della testina che sarà effettuato eseguendo l'istruzione. Osserviamo che, in generale, la tabella non contiene tante righe quante sono le possibili coppie formate da uno stato e un simbolo dell'alfabeto di lavoro della macchina, in quanto per alcune (generalmente molte) di queste coppie il comportamento della macchina può non essere stato specificato.

Esempio 1.3: rappresentazione tabellare della macchina per il complemento bit a bit

Facendo riferimento alla macchina di Turing introdotta nell'Esempio 1.1 e mostrata nella Figura 1.2, la corrispondente rappresentazione tabellare di tale macchina è la seguente.

stato	simbolo	stato	simbolo	movimento
q0	0	q0	1	R
q0	1	q0	0	R
q0	□	q1	□	L
q1	0	q1	0	L
q1	1	q1	1	L
q1	□	q2	□	R

1.1.2 Macchine di Turing e JFLAP

Nel seguito faremo spesso riferimento all'applicativo JFLAP, che consente di creare macchine di Turing facendo uso di una semplice interfaccia grafica (la macchina di Turing mostrata nella Figura 1.2 è stata prodotta facendo uso di tale applicativo), e inviteremo il lettore a sperimentare una determinata macchina di Turing attraverso un riquadro come quello seguente, nella cui parte centrale viene specificato il nome del file XML contenente la descrizione della macchina secondo le regole di JFLAP: tale file può essere ottenuto a partire dal sito web del corso *Informatica teorica: linguaggi, computabiltià, complessità*.

Rappresentazione binaria di numeri interi

I principi alla base del sistema di **numerazione binaria** sono gli stessi di quelli usati dal sistema decimale: infatti, entrambi sono sistemi di numerazione posizionale, nei quali la posizione di una cifra ne indica il valore relativo. In particolare, il sistema binario usa potenze crescenti di 2: per esempio, il numero binario 1001 rappresenta il numero decimale 9, in quanto $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 1 = 9$. La conversione di un numero decimale n in uno binario è invece leggermente più complicata e può essere realizzata nel modo seguente: si divide ripetutamente n per 2, memorizzando in ordine inverso i resti che vengono ottenuti, fino a ottenere un quoziente minore di 2 che rappresenta la cifra binaria più a sinistra. Ad esempio, per ottenere la rappresentazione binaria del numero decimale 9 vengono effettuate le seguenti divisioni: (a) $9 \div 2 = 4$ con resto $\underline{1}$, (b) $4 \div 2 = 2$ con resto $\underline{0}$, (c) $2 \div 2 = \underline{1}$ con resto $\underline{0}$. Pertanto, al numero decimale 9 corrisponde il numero binario 1001. Osserviamo che, come nel sistema decimale con k cifre possiamo rappresentare al massimo 10^k numeri (da 0 a $10^k - 1$), anche nel sistema binario avendo a disposizione k cifre (binarie) possiamo rappresentare al massimo 2^k numeri (da 0 a $2^k - 1$). La maggior parte dei calcolatori, in realtà, rappresenta i numeri interi facendo uso della rappresentazione **complemento a due**. Dati k bit, un numero intero non negativo viene rappresentato normalmente facendo uso di $k-1$ bit preceduti da uno 0, mentre un numero intero negativo x viene rappresentato mediante la rappresentazione binaria di $2^k - x$ che faccia uso di k bit. Ad esempio, avendo a disposizione 4 bit, la rappresentazione complemento a due di 6 è 0110, mentre quella di -6 è 1010 (ovvero la rappresentazione binaria di $2^4 - 6 = 10$).

JFLAP: macchina per il complemento bit a bit

Osserva, sperimenta e verifica la macchina
`bitwisecomplement.jff`

1.2 Esempi di macchine di Turing

IN QUESTO paragrafo mostriamo alcuni esempi di macchine di Turing. Vedremo molti altri esempi nel resto delle dispense, ma già alcuni di quelli che verranno presentati in questo paragrafo evidenziano il tipico comportamento di una macchina di Turing: in generale, infatti, queste macchine implementano una strategia a *zig-zag* che consente loro di attraversare ripetutamente il contenuto del nastro, combinata con una strategia di *piggy-backing* che consiste nel "memorizzare" nello stato il simbolo o i simboli che sono stati letti.

Sebbene ciò sia l'argomento principale della terza parte di queste dispense, nel seguito valuteremo sempre il numero di passi che ciascuna macchina esegue in funzione della lunghezza della stringa ricevuta in input: a tale scopo faremo uso della notazione O, evitando quindi di dover sempre specificare esattamente il numero di

passi eseguiti e accontentandoci di valutarne l'ordine di grandezza. In particolare, data una macchina di Turing T, indicheremo con $t_T(n)$ la funzione che rappresenta il massimo numero di passi eseguito da T con input una stringa di lunghezza n.

1.2.1 Una macchina per il complemento a due

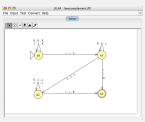
Definiamo una macchina di Turing T che, data la rappresentazione complemento a due di un numero intero non negativo x che faccia uso di k bit, produca la rappresentazione complemento a due di $-x$ che faccia anch'essa uso di k bit. Tale rappresentazione può essere calcolata individuando il simbolo `1` più a destra della rappresentazione di x e complementando tutti i simboli alla sua sinistra: quindi, T opera nel modo seguente.

1. Attraversa la stringa in input fino a raggiungere il primo $\square$ alla sua destra.
2. Si muove verso sinistra fino a trovare e superare un `1`: se tale `1` non esiste (ovvero, la stringa in input è costituita da soli simboli `0` oppure è la stringa vuota), la macchina può terminare la sua esecuzione posizionando la testina sul simbolo `0` più a sinistra (se tale simbolo esiste).
3. Si muove verso sinistra, complementando ogni simbolo incontrato, fino a raggiungere il primo $\square$ e posiziona la testina sul simbolo immediatamente a destra.

Chiaramente, $t_T(n) \in O(n)$: infatti, per ogni stringa x di lunghezza n, n passi sono eseguiti per raggiungere il primo $\square$ alla destra di x e altri n passi sono eseguiti per effettuare la ricerca del simbolo `1` e la complementazione dei simboli alla sua sinistra.

JFLAP: macchina per il calcolo del complemento a due

Osserva, sperimenta e verifica la macchina `twocomplement.jff`



1.2.2 Una macchina per l'ordinamento di stringhe binarie

Data una stringa x sull'alfabeto $\Sigma = \{0, 1\}$, la stringa ordinata corrispondente a x è la stringa su Σ ottenuta a partire da x mettendo tutti i simboli `0` prima dei simboli `1`: ad esempio, la stringa ordinata corrispondente a `0101011` è la stringa `0001111`. Definiamo una macchina di Turing T che, data in input una stringa binaria x, termina producendo in output la sequenza ordinata corrispondente a x: T opera nel modo seguente.

Ordini di grandezza

Data una funzione $f : N \to R$, $O(f(n))$ denota l'insieme delle funzioni $g : N \to R$ per le quali esistono due costanti $c > 0$ e $n_0 > 0$ per cui vale $g(n) \leqslant cf(n)$, per ogni $n > n_0$. Inoltre, $\Omega(f(n))$ denota l'insieme delle funzioni $g : N \to R$ per le quali esistono due costanti $c > 0$ e $n_0 > 0$ e infiniti valori $n > n_0$ per cui vale $g(n) \geqslant cf(n)$. Infine, $o(f(n))$ denota l'insieme delle funzioni $g : N \to R$ tali che $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

1. Scorre il nastro verso destra alla ricerca di un simbolo `1`. Se non lo trova, vuol dire che la testina è posizionata sul primo simbolo □ successivo alla stringa che è ora ordinata: in tal caso, scorre il nastro verso sinistra fino a posizionare la testina sul primo simbolo diverso da □ e termina l'esecuzione.

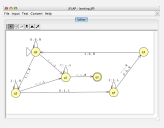
2. Scorre il nastro verso destra alla ricerca di un simbolo `0`. Se non lo trova, vuol dire che la testina è posizionata sul primo simbolo □ successivo alla stringa che è ora ordinata: in tal caso, scorre il nastro verso sinistra fino a posizionare la testina sul primo simbolo diverso da □ e termina l'esecuzione.

3. Complementa il simbolo `0` e scorre il nastro verso sinistra alla ricerca di un simbolo `0` oppure di un simbolo □, posizionando la testina immediatamente dopo tale simbolo, ovvero, sul simbolo `1` trovato in precedenza: complementa tale simbolo e ricomincia dal primo passo.

Per ogni stringa x di lunghezza n, la ricerca di una coppia di simboli `1` e `0` posizionati in ordine sbagliato richiede al più n passi: il successivo scambio di questi due simboli richiede anch'esso un numero lineare di passi. Poiché vi possono essere al più $\frac{n}{2}$ coppie di simboli invertiti, abbiamo che $t_T(n) \in O(n^2)$.

JFLAP: macchina per l'ordinamento di stringhe binarie

Osserva, sperimenta e verifica la macchina `sorting.jff`



1.2.3 Una macchina per un linguaggio non regolare

Consideriamo il seguente linguaggio L sull'alfabeto $\Sigma = \{0, 1\}$.

$$L = \{0^n 1^n : n \geqslant 0\}$$

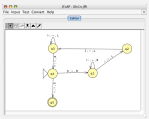
Ad esempio, la stringa `00001111` appartiene a L, mentre la stringa `0001111` non vi appartiene. Tale linguaggio, come vedremo nella seconda parte di queste dispense, svolge un ruolo molto importante nell'ambito della teoria dei linguaggi formali, in quanto rappresenta il classico esempio di linguaggio generabile da una grammatica contestuale, ma non generabile da una grammatica regolare. Definiamo una macchina di Turing che, data in input una sequenza di simboli `0` seguita da una sequenza di simboli `1`, termina in uno stato finale se il numero di simboli `0` è uguale a quello dei simboli `1`, altrimenti termina in uno stato non finale non avendo istruzioni da poter eseguire. In questo caso, la posizione finale della testina è ininfluente, in quanto l'output prodotto dalla macchina viene codificato attraverso il fatto che la computazione sia terminata o meno in uno stato finale. La macchina opera nel modo seguente.

1. Partendo dall'estremità sinistra della stringa in input, sostituisce un simbolo `0` con il simbolo $\square$ e si muove verso destra, ignorando tutti i simboli `0` e `1`, fino a incontrare il simbolo $\square$.
2. Verifica che il simbolo immediatamente a sinistra del simbolo $\square$ sia un `1`: se così non è, allora termina la computazione in uno stato non finale.
3. Sostituisce il simbolo `1` con il simbolo $\square$ e scorre il nastro verso sinistra fino a incontrare un simbolo $\square$.
4. Se il simbolo immediatamente a destra del $\square$ è anch'esso un simbolo $\square$, allora termina in uno stato finale. Se, invece, è un `1`, allora termina in uno stato non finale. Altrimenti, ripete l'intero procedimento a partire dal primo passo.

Anche in questo caso, per ogni stringa x di lunghezza n, la ricerca di una coppia di simboli `0` e `1` posizionati, rispettivamente, all'estremità sinistra e destra della stringa binaria rimanente richiede un numero lineare di passi. Poiché vi possono essere al più $\frac{n}{2}$ di tali coppie, abbiamo che $t_T(n) \in O(n^2)$. Osserviamo che, in base alla descrizione precedente, la stringa in ingresso viene modificata dalla macchina in modo tale da non poter essere successivamente ricostruita: non è difficile, comunque, modificare la macchina in modo che, facendo uso di simboli ausiliari, sia possibile ricostruire in un numero di passi lineare, al termine del procedimento sopra descritto, la stringa ricevuta in ingresso.

JFLAP: macchina per 0^n1^n

Osserva, sperimenta e verifica la macchina
`0n1n.jff`



Logaritmi

Dato un numero intero positivo b, il **logaritmo** in base b di un numero reale positivo x, indicato con $\log_b x$, è il numero reale y tale che $x = b^y$. Se $b = 2$, generalmente la base viene omessa mentre, se $b = e$ (dove $e = 2{,}71828\ldots$ indica la costante di Nepero), allora il logaritmo viene indicato con $\ln x$. Per ogni coppia di basi b e b' e per ogni numero reale positivo x, vale la seguente relazione: $\log_{b'} x = \frac{\log_b x}{\log_b b'}$.

1.2.4 Una macchina per la conversione da unario in binario

Definiamo una macchina che calcola la rappresentazione binaria di un numero intero positivo a partire dalla sua rappresentazione unaria: a tale scopo possiamo assumere che un numero intero positivo x sia rappresentato in unario da x simboli `U` (quindi, 1 è rappresentato da `U`, 2 è rappresentato da `UU`, 3 è rappresentato da `UUU` e così via). Tale macchina può fare uso del seguente semplice algoritmo per aumentare di un'unità un numero intero rappresentato in forma binaria: scandisce i bit da destra verso sinistra, complementando ciascun simbolo `1`, fino a trovare uno `0` oppure un $\square$ che viene cambiato in un simbolo `1`. A questo punto la macchina per la conversione della rappresentazione unaria di un numero intero positivo in quella binaria può operare nel modo seguente.

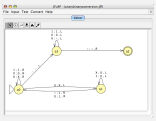
1. Cerca un simbolo `U` spostandosi verso destra e lo sostituisce con il simbolo `X`. Se il simbolo `U` non viene trovato (ovvero, si incontra un simbolo $\square$), allora cancella tutti i simboli `X` e posiziona la testina sul primo simbolo della rappresentazione binaria.

2. Si sposta verso sinistra fino a trovare il primo simbolo della rappresentazione binaria (eventualmente vuota) e, applicando l'algoritmo descritto in precedenza, aumenta di un'unità il numero intero da essa rappresentato. Quindi, torna al passo precedente.

Per ogni stringa x di n simboli `U`, la ricerca del prossimo simbolo `U` da aggiungere al contatore binario richiede, a regime, un numero di passi proporzionale alla lunghezza del contatore più quella della stringa in ingresso: poiché la rappresentazione binaria di un numero intero m richiede $O(\log m)$ cifre, abbiamo che la ricerca del simbolo `U` richiede un numero di passi proporzionale a $n + \log n$. Un numero di passi logaritmico è invece richiesto per il successivo incremento del contatore binario: pertanto, abbiamo che $t_T(n) \in O(n(n + \log n)) = O(n^2)$.

JFLAP: macchina per la conversione da unario in binario

Osserva, sperimenta e verifica la macchina
`unarybinaryconversion.jff`



1.2.5 Una macchina per il decremento di un numero intero binario

Nell'esempio precedente abbiamo visto come sia possibile aumentare di un'unità un numero intero rappresentato in forma binaria: in questo paragrafo definiamo, invece, una macchina di Turing T che decrementa di un'unità un numero intero positivo rappresentato in forma binaria. Tale macchina opera nel modo seguente.

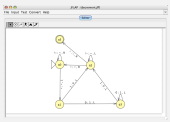
1. Si sposta verso destra fino a trovare l'ultimo simbolo della rappresentazione binaria del numero intero: se tale simbolo è un `1`, lo complementa e posiziona la testina sul primo simbolo della stringa binaria.

2. Se l'ultimo simbolo è uno `0`, lo complementa e si sposta verso sinistra fino a trovare un simbolo `1` e complementando ciascun simbolo `0` incontrato: complementa il simbolo `1` e posiziona la testina sul primo simbolo della stringa binaria.

Per ogni stringa binaria x di lunghezza n, la ricerca del suo ultimo simbolo richiede un numero lineare di passi e un analogo numero di passi è richiesto per la ricerca della prima posizione a cui "chiedere in prestito" un simbolo `1` (se tale posizione esiste): pertanto, abbiamo che $t_T(n) \in O(n)$.

JFLAP: macchina per il decremento di un numero intero binario

Osserva, sperimenta e verifica la macchina
`decrement.jff`



Osserviamo che la macchina appena descritta termina in modo anomalo (ovvero, a causa della non esistenza di operazioni da eseguire) nel caso in cui la stringa binaria sia vuota oppure nel caso in cui sia la rappresentazione del numero 0. In generale, in questo e in tutti gli esempi precedenti, abbiamo sempre assunto che l'input sia fornito in modo corretto. Ad esempio, la descrizione della macchina di Turing per la conversione da unario in binario, assume che l'input sia sempre costituito da una

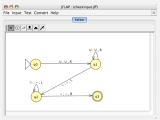
sequenza non vuota di simboli `U`. Se così non fosse, potrebbero verificarsi diverse situazioni di errore. Ad esempio, se la stringa in ingresso fosse `UUAUU`, si avrebbe che la macchina, dopo aver contato fino a 2, nello stato iniziale leggerebbe il simbolo `A`: in tal caso, la macchina si arresterebbe in modo anomalo, in quanto non sarebbe definita una transizione in corrispondenza di tale situazione. Se, invece, la stringa in ingresso fosse `XXUUXUX`, allora la macchina, invece di segnalare in qualche modo il fatto che l'input non è una corretta rappresentazione unaria di un un numero intero positivo, si arresterebbe nello stato finale producendo come output la stringa `11` (ovvero, la rappresentazione binaria del numero di simboli `U` presenti nella stringa in ingresso). In tutti gli esempi mostrati sinora, comunque, è sempre facile verificare l'esistenza di una macchina di Turing in grado di controllare che l'input sia stato fornito in modo corretto. Ad esempio, una macchina di Turing che verifichi che l'input sia costituito da una sequenza non vuota di simboli `U`, può facilmente essere definita nel modo seguente.

1. Controlla che il simbolo attualmente letto sia `U`: se così non fosse, termina segnalando un errore (ad esempio, non definendo le transizioni corrispondenti a simboli diversi da `U`).

2. Sposta la testina a destra: se il simbolo attualmente letto è □, posiziona nuovamente la testina sul primo simbolo della stringa in input e termina. Altrimenti, torna al passo precedente.

JFLAP: macchina per la verifica della correttezza dell'input

Osserva, sperimenta e verifica la macchina
`checkinput.jff`



In generale, esplicitare una macchina di Turing che verifichi la correttezza dell'input all'interno di un esempio, complicherebbe solamente l'esposizione senza aggiungere nulla di particolarmente interessante alla comprensione dell'esempio stesso. Per questo motivo, nel seguito continueremo ad assumere che l'input fornito in ingresso a una macchina di Turing sia sempre corretto.

1.2.6 Una macchina per la somma di due numeri interi binari

Definiamo, in quest'ultimo esempio, una macchina che calcola la somma di due numeri interi non negativi rappresentati in forma binaria, assumendo che i due numeri x

e y siano inizialmente presenti sul nastro uno di seguito all'altro e separati dal simbolo +. Una tale macchina potrebbe essere definita facendo riferimento ai due algoritmi, visti in precedenza, che, rispettivamente, aumentano e diminuiscono di un'unità un numero intero rappresentato in forma binaria: in particolare, potremmo ripetutamente aumentare di un'unità x e diminuire di un'unità y, fino a quando quest'ultimo non diventa nullo. Seguendo tale approccio, tuttavia, il numero di passi richiesto sarebbe proporzionale al *valore* di y moltiplicato per il numero di passi richiesti dall'aumento e dal decremento dei due numeri: poiché la *lunghezza* di y è proporzionale a $\log y$, avremmo che il numero di passi eseguito da tale macchina sarebbe esponenziale rispetto alla lunghezza della stringa in ingresso. Una macchina di Turing T più efficiente può essere definita simulando, invece, il normale procedimento utilizzato per sommare due numeri, che consiste nello scorrerli da sinistra verso destra, sommando le corrispondenti cifre binarie e l'eventuale riporto: una tale macchina può operare nel modo seguente.

1. Inizializza il risultato della somma, scrivendo alla sinistra del primo numero il simbolo `0` (attuale riporto) seguito dal simbolo = (poiché il risultato crescerà di lunghezza durante l'esecuzione, conviene memorizzarlo alla sinistra dell'input).

2. Posiziona la testina sul primo simbolo □ che segue il simbolo +.

3. Scorre il nastro da destra verso sinistra memorizzando e cancellando gli ultimi simboli del secondo e del primo numero: se uno di tali numeri non è più presente, considera come suo ultimo simbolo il simbolo `0`.

4. Sposta la testina all'estremità sinistra del risultato e, a seconda dei due simboli precedentemente letti e del simbolo alla sinistra del risultato (ovvero del riporto), determina e scrive il prossimo simbolo del risultato e il nuovo riporto.

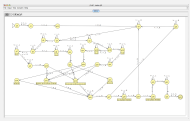
5. Posiziona la testina sul primo simbolo □ che segue il simbolo +, verificando se almeno uno dei due numeri è ancora presente: in tal caso, torna al Passo 3. Altrimenti, cancella i simboli + e =, cancella il riporto attuale se è pari a `0` e termina con la testina posizionata sul primo simbolo del risultato.

Osserviamo che, in base alla descrizione precedente, la rappresentazione binaria della somma dei due numeri interi userà un numero di simboli pari al massimo tra il numero di simboli della rappresentazione del primo numero e quello della rappresentazione del secondo numero, eventualmente incrementato di 1. Quindi, per ogni coppia di numeri rappresentati in forma binaria la cui lunghezza sia, rispettivamente, n_1 e n_2 con $n_1 \geqslant n_2$, la macchina di Turing T appena descritta esegue un numero di

passi proporzionale al prodotto tra n_1 e $2n_1 + n_2$: pertanto, $t_{T(n)} \in O(n^2)$. Abbiamo quindi ottenuto una macchina di Turing significativamente più veloce di quella basata sull'aumento e sul decremento iterativo dei due numeri.

JFLAP: macchina per la somma di due numeri interi binari

Osserva, sperimenta e verifica la macchina
`adder.jff`



1.3 Macchine di Turing multi-nastro

NEL PRIMO paragrafo di questo capitolo abbiamo definito la macchina di Turing, assumendo che essa possa utilizzare un *solo* nastro di lettura e scrittura: in alcuni degli esempi del secondo paragrafo, questo vincolo ci ha costretto a dover scorrere ripetutamente il contenuto del nastro e, quindi, a eseguire un elevato numero di passi (tipicamente quadratico rispetto alla lunghezza della stringa in ingresso). In questo paragrafo, vedremo come questa restrizione non influenza in alcun modo il potere di calcolo di una macchina di Turing, anche se l'utilizzo di un numero maggiore di nastri può consentire di sviluppare macchine di Turing più veloci.

Una **macchina di Turing multi-nastro** è del tutto simile a una macchina di Turing ordinaria, con l'unica differenza che essa ha a disposizione un numero fissato k di nastri con $k \geqslant 1$. Ciascun nastro è dotato di una testina di lettura e scrittura: ogni istruzione del programma della macchina di Turing deve, quindi, specificare, oltre al nuovo stato, i k simboli letti dalle k testine, affinché l'istruzione sia eseguita, i k simboli che devono essere scritti al loro posto, e i k movimenti che devono essere eseguiti dalle k testine.

Formalmente, quindi, a ogni transizione è associata un'etichetta formata da una lista di triple

$$((\sigma_1,\ldots,\sigma_k),(\tau_1,\ldots,\tau_k),(m_1,\ldots,m_k))$$

I simboli $\sigma_1,\ldots,\sigma_k$ e $\tau_1,\ldots,\tau_k$ che appaiono all'interno di una tripla dell'etichetta di una transizione indicano, rispettivamente, i k simboli attualmente letti dalle k testine e i k simboli da scrivere, mentre i valori $m_1,\ldots,m_k$ specificano i movimenti che devono essere effettuati dalle k testine.

Inizialmente, una macchina di Turing con k nastri viene avviata con la stringa di input posizionata sul primo nastro, con i rimanenti $k-1$ nastri vuoti (ovvero, contenenti solo simboli $\square$), con la testina del primo nastro posizionata sul primo simbolo della stringa di input e con la CPU configurata nello stato iniziale. Nel caso in cui la

macchina produca un output, assumiamo anche che essa termini con la CPU configurata in uno stato finale, con la stringa di output contenuta nel primo nastro e seguita da un simbolo □ e con la testina del primo nastro posizionata sul primo simbolo della stringa di output.

1.3.1 Un esempio di macchina di Turing multi-nastro

Consideriamo nuovamente il linguaggio $L = \{0^n 1^n : n \geqslant 0\}$ e definiamo una macchina di Turing T con 2 nastri che, data in input una sequenza di simboli `0` seguita da una sequenza di simboli `1`, termina in uno stato finale se il numero di simboli `0` è uguale a quello dei simboli `1`, altrimenti termina in uno stato non finale non avendo istruzioni da poter eseguire. Tale macchina opera nel modo seguente.

1. Scorre il primo nastro verso destra fino a incontrare il primo simbolo `1`: per ogni simbolo `0` letto viene scritto un simbolo `1` sul secondo nastro, spostando la testina a destra dopo aver eseguito la scrittura.

2. Scorre il primo nastro da sinistra verso destra e il secondo nastro da destra verso sinistra, controllando che i simboli letti sui due nastri siano uguali: se così non fosse, termina senza avere alcuna istruzione da poter eseguire.

3. Una volta incontrato il simbolo □ su entrambi i nastri, termina nello stato finale.

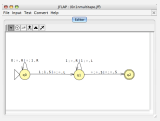
La rappresentazione tabellare di T è, dunque, la seguente.

stato	simboli	stato	simboli	movimenti
q0	(0,□)	q0	(□,1)	(R,R)
q0	(1,□)	q1	(1,□)	(S,L)
q1	(1,1)	q1	(□,□)	(R,L)
q1	(□,□)	q2	(□,□)	(S,S)

Per ogni stringa x di lunghezza n, il numero di passi eseguito da T prima di terminare è chiaramente lineare in n: in effetti, T legge una e una sola volta ciascun simbolo di x presente sul primo nastro (a eccezione del primo simbolo `1`). Pertanto, tale macchina di Turing multi-nastro è significativamente più veloce di quella con un solo nastro, che abbiamo descritto nel secondo paragrafo di questo capitolo e che eseguiva un numero di passi quadratico rispetto alla lunghezza della stringa in ingresso.

JFLAP: macchina multi-nastro per 0^n1^n

Osserva, sperimenta e verifica la macchina
`0n1nmultitape.jff`



1.3.2 Simulazione di macchine di Turing multi-nastro

Chiaramente, tutto ciò che può essere fatto da una macchina di Turing ordinaria risulta essere realizzabile da una macchina di Turing multi-nastro. In effetti, una macchina di Turing ordinaria T corrisponde al caso particolare in cui il numero di nastri k sia uguale a 1: pertanto, una macchina di Turing con $k > 1$ nastri può simulare T sul primo nastro non modificando mai il contenuto dei rimanenti $k-1$ nastri. A prima vista, potrebbe invece sembrare che le macchine multi-nastro siano computazionalmente più potenti di quelle ordinarie. Tuttavia, facendo uso della tecnica di zig-zag combinata con quella di piggy-backing viste nel secondo paragrafo di questo capitolo, è possibile dimostrare che i due modelli di calcolo sono in realtà equivalenti. In particolare, possiamo mostrare che ogni macchina di Turing T con k nastri, dove $k > 1$, può essere simulata da una macchina di Turing T' con un solo nastro.

L'idea della dimostrazione consiste nel concatenare uno dopo l'altro il contenuto dei k nastri, separati da un simbolo speciale, che non faccia parte dell'alfabeto di lavoro della macchina multi-nastro: indichiamo con $\#$ tale simbolo. La macchina T' simulerà ogni istruzione di T scorrendo i k nastri e "raccogliendo" le informazioni relative ai k simboli letti dalle k testine. Terminata questa prima scansione, T' sarà in grado di decidere quale transizione applicare, per cui scorrerà nuovamente i k nastri applicando su ciascuno di essi la corretta operazione di scrittura e di spostamento della testina. Al termine di questa seconda scansione, T' riposizionerà la testina all'inizio del primo nastro e sarà così pronta a simulare la successiva istruzione.

Formalmente, sia Σ l'alfabeto di lavoro di T. Allora, l'alfabeto di lavoro di T' sarà definito nel modo seguente.

$$\Sigma' = \Sigma \cup \{\sigma^{\text{head}} : \sigma \in \Sigma\} \cup \{\#, \top\}$$

Intuitivamente, ciascun simbolo σ^{head} consentirà di identificare, per ogni nastro, il simbolo attualmente scandito dalla sua testina mentre il simbolo $\top$ sarà utilizzato come delimitatore sinistro e destro del nastro di T'. Se q_0 è lo stato iniziale di T, F è l'insieme dei suoi stati finali e Q l'insieme dei rimanenti suoi stati, allora la macchina T' avrà lo stesso stato iniziale di T e gli stessi stati finali e, per ogni stato $q \in \{q_0\} \cup Q$,

avrà il seguente insieme di stati.

$$\bigcup_{h=0}^{k} \{q^{\sigma_1,\ldots,\sigma_h}, \bar{q}^{\sigma_1,\ldots,\sigma_h} : \sigma_i \in \Sigma \text{ per ogni } i \text{ con } 1 \leqslant i \leqslant h\}$$

Intuitivamente, tali stati consentiranno di memorizzare i simboli scanditi dalle k testine, man mano che questi vengono identificati. La macchina T', che avrà ulteriori stati ausiliari, opera nel modo seguente.

- "Crea" i k nastri separandoli con il simbolo $\#$. In particolare, ogni nastro (a eccezione del primo) sarà creato con un solo simbolo $\square^{\text{head}}$ al suo interno, mentre sul primo nastro il primo simbolo σ della stringa in ingresso in esso contenuta sarà sostituito con il suo corrispondente simbolo σ^{head}. Inoltre, per indicare l'estremo sinistro del primo nastro e quello destro dell'ultimo nastro, sostituisce il primo simbolo $\square$ alla sinistra del primo nastro e il primo simbolo $\square$ alla destra dell'ultimo nastro con il simbolo speciale $\top$. Infine, lascia la testina posizionata sul simbolo alla destra dell'istanza sinistra di $\top$ e entra nello stato q_0^{λ}.

- Essendo nello stato q^{λ} con $q \in \{q_0\} \cup Q$ ed essendo la testina posizionata sul primo simbolo del primo nastro, legge i k simboli attualmente scanditi dalle k testine e memorizza quest'informazione nel corrispondente stato. In particolare, scorre il nastro da sinistra verso destra e, ogni qualvolta incontra un simbolo σ^{head}, passa dallo stato $q^{\sigma_1,\ldots,\sigma_h}$ allo stato $q^{\sigma_1,\ldots,\sigma_h,\sigma}$, dove $0 \leqslant h < k$. Una volta raggiunto lo stato $q^{\sigma_1,\ldots,\sigma_k}$, scorre il nastro verso sinistra fino a incontrare l'istanza sinistra del simbolo $\top$, posiziona la testina sul simbolo immediatamente a destra e entra nello stato $\bar{q}^{\sigma_1,\ldots,\sigma_k}$.

- Supponiamo che, essendo nello stato $\bar{q}^{\sigma_1,\ldots,\sigma_k}$ con la testina posizionata sul primo simbolo del primo nastro, esista una transizione della macchina di Turing multi-nastro che parte dallo stato q e che termina in uno stato p, la cui etichetta contiene la tripla

  $$e = ((\sigma_1,\ldots,\sigma_k),(\tau_1,\ldots,\tau_k),(m_1,\ldots,m_k))$$

  Allora, T' scandisce i k nastri sostituendo ciascun simbolo σ_i^{head} che incontra con il corrispondente simbolo τ_i e simulando lo spostamento della testina corrispondente al movimento m_i. Per realizzare lo spostamento della testina, potrà essere necessario spostare il contenuto dell'intero nastro di una posizione a destra: ciò accade, per esempio, se la testina di uno dei k nastri si deve spostare a destra, ma il simbolo alla destra della testina della macchina con

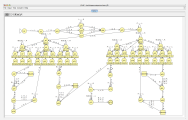
un solo nastro è il simbolo #. Al termine della simulazione della transizione, posiziona la testina sul simbolo alla destra dell'istanza sinistra di $\top$: se $p \notin F$, allora entra nello stato p^{λ} e torna al passo precedente. Altrimenti sostituisce il simbolo # più a sinistra con il simbolo $\square$ sostituisce il simbolo σ^{head} contenuto nel primo nastro con il simbolo σ e termina in p con la testina posizionata su tale simbolo.

- Se, essendo nello stato $\bar{q}^{\sigma_1,\ldots,\sigma_k}$ con la testina posizionata sul primo simbolo del primo nastro, non esiste una transizione della macchina di Turing multi-nastro che parte dallo stato q, la cui etichetta contenga una tripla il cui primo elemento è $(\sigma_1,\ldots,\sigma_k)$, allora termina in modo anomalo, ovvero non avendo alcuna transizione uscente dallo stato $\bar{q}^{\sigma_1,\ldots,\sigma_k}$.

Questa descrizione del comportamento di T' è per forza di cose non del tutto formale, in quanto una costruzione rigorosa di T' richiederebbe la specifica esatta del suo grafo delle transizioni: dovrebbe comunque risultare chiaro che, volendo, ciò può essere fatto.

JFLAP: simulazione di una macchina multi-nastro

Osserva, sperimenta e verifica la macchina
`multitapesimulator.jff`



In conclusione, abbiamo mostrato come una macchina di Turing con un solo nastro possa simulare in modo corretto il comportamento di una macchina di Turing multi-nastro e, quindi, che le macchine di Turing multi-nastro non sono computazionalmente più potenti di quelle con un singolo nastro. Osserviamo, però, che la simulazione descritta in precedenza causa un numero di passi eseguiti da T' significativamente maggiore di quello relativo alla macchina di Turing T con k nastri. In effetti, per ogni stringa x di lunghezza n, abbiamo che, per ogni passo di T, la macchina T' esegue anzitutto una doppia scansione del contenuto dei k nastri per determinare i simboli attualmente letti. Successivamente, un'ulteriore doppia scansione viene eseguita per eventualmente simulare la transizione: durante la prima scansione, inoltre, potrebbe essere richiesto uno spostamento a destra del contenuto di tutti i nastri che seguono quello su cui si sta operando la transizione. Poiché la lunghezza di ciascun nastro può essere pari a $t_T(n)$ (nel caso in cui, a ogni passo, una nuova cella di ciascun nastro venga scandita), abbiamo che il numero di passi eseguiti da T' per simulare un passo di T può essere proporzionale a $t_T(n)$. Pertanto, possiamo concludere che $t_{T'}(n) \in O(t_T^2(n))$, ovvero che la simulazione può richiedere un numero di passi quadratico rispetto al numero di passi eseguiti dalla macchina multi-nastro.

1.4 Configurazioni di una macchina di Turing

DURANTE L'ESECUZIONE del suo programma, una macchina di Turing T può cambiare lo stato della CPU, il contenuto del nastro e la posizione della testina. La combinazione di queste tre componenti è detta **configurazione** di T e viene generalmente rappresentata nel seguente modo: date due stringhe x e y, con $x, y \in \Sigma^*$ dove Σ è l'alfabeto di lavoro di T, e dato uno stato q di T, $x^q y$ rappresenta la configurazione in cui lo stato della CPU è q, il nastro contiene la stringa xy e la testina è posizionata sul primo simbolo di y.

Notiamo che sebbene il nastro di una macchina di Turing sia infinito, la rappresentazione di una configurazione, così come è stata appena definita, fa riferimento a sequenze finite di simboli. Ciò non costituisce una contraddizione, in quanto possiamo sempre assumere che i simboli alla sinistra e alla destra di una configurazione siano tutti uguali al simbolo $\square$.

In particolare, in queste dispense faremo sempre riferimento a rappresentazioni minimali che sono definite nel modo seguente: una configurazione $x^q y$ è **minimale** se (1) tutti i simboli che si trovano alla sinistra di x nel nastro sono uguali a $\square$, (2) tutti i simboli che si trovano alla destra di y nel nastro sono uguali a $\square$, (3) il primo simbolo di x è diverso da $\square$, nel caso in cui $|x| \geqslant 1$ e (4) l'ultimo simbolo di y è diverso da $\square$, nel caso in cui $|y| \geqslant 1$.

Data una stringa $x \in \Sigma^* - \{\square\}$, la **configurazione iniziale** della macchina di Turing T con input la stringa x è dunque la configurazione ^{q_0}x, dove q_0 è lo stato iniziale di T. Inoltre, una configurazione $x^q y$ è detta **finale** o **di accettazione** se q è uno stato finale: l'**output** di tale configurazione è il prefisso più lungo di y che non contiene alcun simbolo $\square$. Infine, una configurazione $x^q y$ è detta **di rigetto** se non esiste alcuna transizione a partire dallo stato q la cui etichetta contenga una tripla il cui primo elemento è il primo simbolo di y se $|y| \geqslant 1$, il simbolo $\square$ altrimenti.

Esempio 1.4: configurazioni della macchina per il complemento bit a bit

Partendo dalla configurazione iniziale $^{q_0}0101$, in cui quindi l'input è la stringa 0101, la macchina di Turing rappresentata in Figura 1.2 passerà attraverso le seguenti configurazioni: $1^{q_0}101$, $10^{q_0}01$, $101^{q_0}1$, 1010^{q_0}, $101^{q_1}0$, $10^{q_1}10$, $1^{q_1}010$, $^{q_1}1010$ e $^{q_1}\square 1010$. L'ultima configurazione sarà seguita dalla configurazione $^{q_2}1010$, che è finale (in quanto lo stato q_2 è uno stato finale) e che corrisponde al termine dell'esecuzione del programma. Come era lecito aspettarsi, la stringa prodotta in output dall'esecuzione, ovvero 1010, è il complemento bit a bit della stringa di input.

Connettivi e quantificatori

Faremo spesso uso di notazioni della logica elementare: in particolare, 'e' sarà abbreviato con $\wedge$, 'o' con $\vee$, 'solo se' con $\rightarrow$ e 'non' con $\neg$. Tutti questi simboli sono chiamati **connettivi**. I simboli $\forall$ e $\exists$ sono invece i **quantificatori esistenziale** e **universale**, rispettivamente: pertanto, $\forall x$ si legge 'per ogni x' mentre $\exists x$ si legge 'esiste x tale che'. I simboli logici appena introdotti consentiranno di rappresentare in modo sintetico espressioni del linguaggio matematico ordinario: ad esempio, $A \subseteq B$ può essere espresso come $\forall x[x \in A \rightarrow x \in B]$ oppure $A \neq B$ può essere espresso come $\exists x[\neg(x \in A \wedge x \in B)]$.

1.4.1 Produzioni tra configurazioni

In precedenza, abbiamo ripetutamente parlato di passi eseguiti da una macchina di Turing con input una determinata stringa. In questo paragrafo, specifichiamo formalmente che cosa intendiamo per singolo passo di una macchina di Turing: a tale scopo, faremo uso di alcune funzioni ausiliarie. In particolare, dato un qualunque alfabeto Σ contenente il simbolo $\square$, definiamo le seguenti funzioni con dominio l'insieme Σ^*.

$$\texttt{left}(x) = \begin{cases} x' & \text{se } x = x'\alpha \text{ con } \alpha \in \Sigma, \\ \lambda & \text{altrimenti} \end{cases}$$

$$\texttt{right}(x) = \begin{cases} x' & \text{se } x = \alpha x' \text{ con } \alpha \in \Sigma, \\ \lambda & \text{altrimenti} \end{cases}$$

$$\texttt{first}(x) = \begin{cases} \alpha & \text{se } x = \alpha x' \text{ con } \alpha \in \Sigma, \\ \square & \text{altrimenti} \end{cases}$$

$$\texttt{last}(x) = \begin{cases} \alpha & \text{se } x = x'\alpha \text{ con } \alpha \in \Sigma, \\ \square & \text{altrimenti} \end{cases}$$

$$\texttt{reduceLR}(x) = \begin{cases} \texttt{reduceLR}(x') & \text{se } x = \square x', \\ x & \text{altrimenti} \end{cases}$$

$$\texttt{reduceRL}(x) = \begin{cases} \texttt{reduceRL}(x') & \text{se } x = x'\square, \\ x & \text{altrimenti} \end{cases}$$

Data una macchina di Turing T con alfabeto di lavoro Σ, dati due stati q e p di T di cui q non è finale e date quattro stringhe x, y, u e v in Σ^*, diremo che la configurazione $C_1 = x^q y$ produce la configurazione $C_2 = u^p v$ se T può passare da C_1 a C_2 eseguendo un'istruzione del suo programma. Formalmente, diremo che C_1

produce C_2 se esiste un arco del grafo delle transizioni di T da q a p la cui etichetta contiene una tripla (σ, τ, m) tale che

$$(y = \sigma y') \vee (y = \lambda \wedge \sigma = \square)$$

e una delle seguenti tre condizioni sia soddisfatta.

Mossa a sinistra $m = \mathtt{L} \wedge u = \mathtt{left}(x) \wedge v = \mathtt{reduceRL}(\mathtt{last}(x)\tau\mathtt{right}(y))$.

Nessuna mossa $m = \mathtt{S} \wedge u = x \wedge v = \mathtt{reduceRL}(\tau\mathtt{right}(y))$.

Mossa a destra $m = \mathtt{R} \wedge u = \mathtt{reduceLR}(x\tau) \wedge v = \mathtt{right}(y)$.

Esempio 1.5: produzioni della macchina per il complemento bit a bit

In base a quanto visto nell'Esempio 1.4, possiamo dire che, facendo riferimento alla macchina di Turing rappresentata in Figura 1.2, la configurazione $\mathtt{101}^{\mathtt{q0}}\mathtt{1}$ produce la configurazione $\mathtt{1010}^{\mathtt{q0}}$. In questo caso, infatti, $x = \mathtt{101}$, $y = \mathtt{1}$ e l'etichetta della transizione da q0 a q0 contiene la tripla $(\mathtt{1},\mathtt{0},\mathtt{R})$: quindi, $y = \mathtt{1} = \sigma\lambda$, $m = \mathtt{R}$, $u = \mathtt{1010} = \mathtt{reduceLR}(x\tau)$ e $v = \lambda = \mathtt{right}(y)$. Al contrario, non è vero che la configurazione $\mathtt{1}^{\mathtt{q0}}\mathtt{101}$ produce la configurazione $\mathtt{101}^{\mathtt{q0}}\mathtt{1}$. In questo caso, infatti, $x = \mathtt{1}$, $y = \mathtt{101}$ e l'etichetta della transizione da q0 a q0 contiene le due triple $(\mathtt{0},\mathtt{1},\mathtt{R})$ e $(\mathtt{1},\mathtt{0},\mathtt{R})$. Nel caso della prima tripla, non è soddisfatta la condizione $(y = \sigma y') \vee (y = \lambda \wedge \sigma = \square)$, mentre nel caso della seconda tripla non sono soddisfatte le due condizioni $u = \mathtt{reduceLR}(x\tau)$ (in quanto $u = \mathtt{101} \neq \mathtt{10} = \mathtt{reduceLR}(x\tau)$) e $v = \mathtt{right}(y)$ (in quanto $v = \mathtt{1} \neq \mathtt{01} = \mathtt{right}(y)$).

1.5 Sotto-macchine

LA STRUTTURAZIONE di un programma, per la risoluzione di un dato problema, in sotto-programmi, ciascuno di essi dedicato alla risoluzione di un problema più semplice di quello originale, è una pratica molto comune nel campo dello sviluppo di software: ci si riferisce spesso a tale pratica con il termine di **programmazione procedurale**. Tale strategia di programmazione può essere applicata anche nel campo dello sviluppo di macchine di Turing, immaginando che una macchina di Turing possa al suo interno "invocare" delle sotto-macchine, ciascuna di esse dedicata al calcolo di una specifica funzione o, più in generale, alla realizzazione di una specifica operazione.[1]

[1] Abbiamo già fatto uso di una sotto-macchina per lo slittamento del contenuto del nastro di una posizione a destra all'interno della macchina con un solo nastro che simulava una macchina multi-nastro.

Una **sotto-macchina** di una macchina di Turing T è un nodo s del grafo delle transizioni di T a cui è associata una macchina di Turing T_s e il cui comportamento in fase di esecuzione si differenzia da quello di un normale nodo (ovvero, di un nodo corrispondente a uno stato di T) nel modo seguente. Nel momento in cui la CPU di T "entra" in s, il controllo passa alla CPU di T_s, la quale inizia l'esecuzione del programma di T_s a partire dal suo stato iniziale e con il contenuto del nastro e la posizione della testina così come lo erano in T. Nel momento in cui la CPU di T_s raggiunge uno stato finale, il controllo ritorna alla CPU di T la quale eseguirà, se esiste, la transizione da s allo stato successivo in base al simbolo lasciato sotto la testina da T_s.

In altre parole, se T si trova nella configurazione $u^q v$ e il suo grafo delle transizioni include un arco da q a s la cui etichetta contiene la tripla (σ, τ, m) e $(v = \sigma v') \vee (v = \lambda \wedge \sigma = \square)$, allora la macchina T, invece di passare nella configurazione $w^s x$ prodotta da $u^q v$, passa nella configurazione $w^{q^s_0} x$ di T_s, dove q^s_0 è lo stato iniziale di T_s. A questo punto, l'esecuzione procede con il programma di T_s: nel momento in cui tale macchina raggiunge una configurazione finale $y^{q^s_f} z$, allora il controllo passa nuovamente a T che si troverà nella configurazione $y^s z$ e che proseguirà nella configurazione da essa prodotta (se è definita).

1.5.1 Una macchina multi-nastro per la moltiplicazione

Abbiamo visto in precedenza come sia possibile calcolare con una macchina di Turing la somma di due numeri interi non negativi rappresentati in forma binaria. Vediamo ora come calcolare l'operazione di prodotto: a tale scopo, definiamo una macchina di Turing che, dati in input due numeri interi binari x e y separati dal simbolo $\star$, fornisca in output la rappresentazione binaria di $x \times y$. Tale macchina potrebbe ripetutamente sommare x al risultato corrente (partendo da 0), tante volte quanto vale y: tale strategia, tuttavia, porterebbe a un numero di passi proporzionale a y e, quindi, esponenziale rispetto alla lunghezza della sua rappresentazione binaria. Un modo più efficiente di eseguire la moltiplicazione si basa sull'ovvia considerazione che $x \times y = 2 \times x \times \frac{y}{2}$ e consiste nel raddoppiare ripetutamente il risultato corrente (partendo da x), tante volte quante sia necessario dividere per 2 il valore di y prima di farlo diventare pari a 1 (ovvero, $\lfloor \log y \rfloor$ volte). Se $y' = y - 2^{\lfloor \log y \rfloor} > 0$ (ovvero, y non è una potenza di 2), al risultato così ottenuto va aggiunto il prodotto di x per y', applicando lo stesso metodo. Poiché tale procedimento deve al più essere ripetuto un numero di volte pari al numero di bit della rappresentazione binaria di y e poiché ogni iterazione richiede un numero di divisioni e moltiplicazioni per 2 (facilmente realizzabili nel caso di numeri rappresentati in forma binaria) non superiore a $\lfloor \log y \rfloor$, abbiamo che complessivamente questo modo di procedere richiede un tempo quadratico rispetto alla lunghezza dell'input.

Per implementare tale algoritmo faremo uso di diverse macchine di Turing multi-nastro, a cui faremo poi riferimento come sotto-macchine all'interno della macchina di Turing multi-nastro principale. In particolare, quest'ultima usa cinque nastri e opera nel modo seguente.

1. Copia y sul secondo nastro, cancellandolo dal primo nastro e cancellando anche il simbolo $\star$. Inizializza il risultato della moltiplicazione scrivendo 0 sul terzo nastro.

2. Cancella i simboli 0 in testa alla stringa binaria contenuta nel secondo nastro, che rappresenta l'attuale valore di y.

3. Se il contenuto del secondo nastro è la stringa vuota, cancella il contenuto del primo nastro (ovvero x), copia in esso il contenuto del terzo nastro (ovvero il risultato della moltiplicazione) e termina.

4. Copia il contenuto del primo nastro (ovvero x) nel quarto nastro e quello del secondo nastro (ovvero l'attuale valore di y) nel quinto nastro.

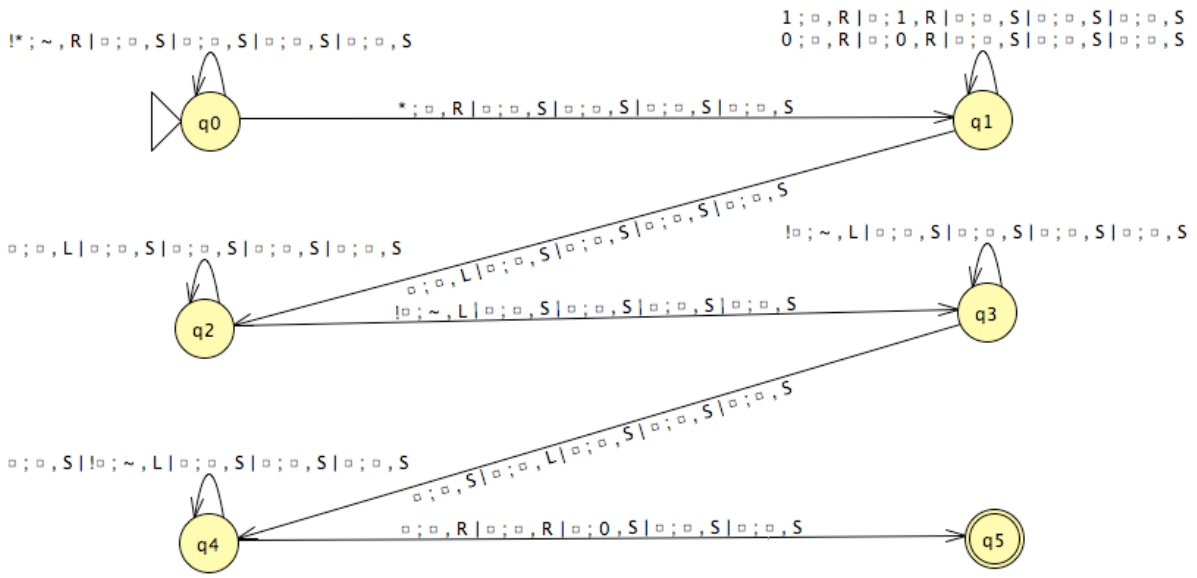
5. Raddoppia il numero contenuto nel quarto nastro (ovvero aggiunge un simbolo 0 a destra) tante volte quanti sono i simboli non vuoti nel quinto nastro meno 1 (in altre parole, calcola $x \times 2^{\lfloor \log y \rfloor}$).

6. Somma il contenuto del quarto nastro a quello del terzo nastro, memorizzando tale somma (ovvero il risultato attuale della moltiplicazione) in quest'ultimo nastro.

7. Cancella il primo simbolo 1 a sinistra contenuto nel secondo nastro (ovvero, calcola il resto della divisione dell'attuale valore di y per 2) e torna al secondo passo.

Per realizzare la macchina appena descritta, possiamo definire una sotto-macchina T_i corrispondente all'i-esimo passo, per ogni i con $1 \leqslant i \leqslant 7$. Per fare ciò, è necessario però specificare in modo preciso quale siano le *pre-condizioni* e le *post-condizioni* associate a tali sotto-macchine: in altre parole, deve essere chiaro cosa ciascuna sotto-macchina si aspetta sia presente sui cinque nastri al momento del suo avvio e cosa la sotto-macchina stessa lascia presente sui cinque nastri al momento della sua terminazione. Ad esempio, la sotto-macchina che realizza il primo passo assume che il primo nastro contenga i due numeri interi binari separati dal simbolo $\star$ e che i rimanenti quattro nastri siano vuoti. Sulla base di queste assunzioni, tale macchina può operare nel modo seguente (si veda la Figura 1.3).

Figura 1.3: il primo passo della macchina di multi-nastro per la moltiplicazione.



1. Scorre il primo nastro verso destra fino a incontrare il simbolo $\star$: le testine degli altri quattro nastri rimangono ferme.

2. Cancella il simbolo $\star$ sul primo nastro e si sposta a destra: le testine degli altri quattro nastri rimangono ferme.

3. Scorre il primo nastro verso destra fino a incontrare il simbolo $\square$: per ogni simbolo diverso da $\square$ che incontra, lo cancella e lo scrive sul secondo nastro, spostando la testina di quest'ultimo a destra. Le testine degli altri tre nastri rimangono ferme.

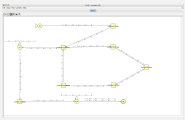
4. Scorre il primo nastro verso sinistra fino a posizionare la testina sul primo simbolo della rappresentazione binaria del primo numero: le testine degli altri quattro nastri rimangono ferme.

5. Scorre il secondo nastro verso sinistra fino a posizionare la testina sul primo simbolo della rappresentazione binaria del secondo numero: le testine degli altri quattro nastri rimangono ferme.

Da tale descrizione segue che, al termine della sua esecuzione, la sotto-macchina lascia sul primo nastro il solo primo numero da moltiplicare con la testina posizionata sul suo primo simbolo e sul secondo nastro il solo secondo numero da moltiplicare con la testina posizionata sul suo primo simbolo: i rimanenti tre nastri non vengono assolutamente modificati dall'esecuzione della sotto-macchina. In modo analogo possiamo definire le sotto-macchine che realizzano i rimanenti sei passi della macchina di Turing principale, specificando per ciascuna di esse le pre-condizioni e le post-condizioni.

JFLAP: macchina multi-nastro per la moltiplicazione di numeri interi binari

Osserva, sperimenta e verifica la macchina
`multiplier.jff`



1.5.2 Sotto-macchine e riusabilità del codice

Oltre che per semplificare la progettazione e la realizzazione di una macchina di Turing, come visto nell'esempio precedente, le sotto-macchine sono uno strumento essenziale per la formulazione di "codice" riutilizzabile. Partendo sempre dall'assunto che siano ben chiare le pre-condizioni e le post-condizioni associate a una specifica macchina di Turing, una tale macchina potrà essere utilizzata in un qualunque contesto essa possa risultare utile (purché ovviamente le pre-condizioni e le post-condizioni siano soddisfatte al momento della sua invocazione e della sua terminazione, rispettivamente).

Consideriamo, ad esempio, il problema di calcolare la somma modulo 9 delle cifre di un numero intero positivo (se tale somma è pari a 0, allora il risultato viene posto per convenzione pari a 9): tale problema è di per sè interessante, in quanto la somma delle cifre della data di nascita di una persona costituisce quello che in numerologia è chiamato *numero del destino* e che determina, appunto, il destino della persona stessa. Ad esempio, l'autore di queste dispense è nato il 21 Giugno 1961, per cui il suo numero del destino è pari a $2+1+6+1+9+6+1 = 26 \equiv 8 \pmod 9$.

È facile definire una macchina di Turing $T_{\texttt{digitadder}}$ con un solo nastro che, partendo con la testina posizionata sulla prima cifra di un numero intero positivo decimale seguito da un simbolo $\square$ (pre-condizione), termini lasciando sul nastro il numero seguito da un simbolo $\square$ e dalla somma modulo 9 delle sue cifre, con la testina posizionata sul primo e unico simbolo di tale somma (post-condizione). Una tale macchina, infatti, può fare uso di nove stati $q_0, \ldots, q_8$ (oltre a ulteriori dieci stati ausiliari) in modo che, per ogni i con $0 \leqslant i \leqslant 8$ e per ogni j con $0 \leqslant j \leqslant 9$, trovandosi

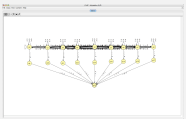
Aritmetica modulare

L'aritmetica modulare rappresenta un importante ramo della matematica, che ha applicazioni nella crittografia e nella teoria dei numeri e che è alla base di molte delle più comuni operazioni aritmetiche e algebriche. Essa si basa sul concetto di **congruenza modulo** n: dati tre numeri interi a, b e n, con $n \neq 0$, diciamo che a e b sono congruenti modulo n se la loro differenza $(a-b)$ è un multiplo di n (in tal caso, scriviamo $a \equiv b \pmod{n}$ e diciamo che a è congruo a b modulo n). Per esempio, possiamo scrivere che $20 \equiv 2 \pmod{9}$, in quanto $20-2=18$, che è un multiplo di 9.

nello stato q_i e leggendo la cifra j vada nello stato $q_{(i+j) \bmod 9}$ e sposti la testina a destra di una posizione. Nel momento in cui, trovandosi nello stato q_i, la macchina legge un simbolo $\square$, essa si sposta a destra di una posizione, scrive la cifra i se $i > 0$ oppure la cifra 9 altrimenti, e termina. Per ogni numero intero decimale di n cifre, $T_{\texttt{digitadder}}$ esegue chiaramente un numero di passi lineare in n, ovvero $t_{T_{\texttt{digitadder}}}(n) \in O(n)$.

JFLAP: macchina per la somma delle cifre di un numero intero

Osserva, sperimenta e verifica la macchina `digitadder.jff`



La macchina appena descritta può essere utile per realizzare una macchina di Turing che esegua la **prova del nove**, che è un metodo per verificare la correttezza di una moltiplicazione, basato sulle proprietà dell'aritmetica modulare, la cui forma tradizionale è la seguente. Innanzitutto si traccia una croce, che delimita quattro zone che chiameremo A, B, C e D, come mostrato nella figura seguente.

A	B
C	D

Le quattro zone vengono riempite applicando le seguenti quattro regole (come esempio, verifichiamo che $1902 \times 1964 = 3735528$ supera la prova del nove).

1. Si sommano ripetutamente le cifre del primo fattore, finché non resta un numero a una sola cifra che va scritto in A (nel nostro esempio, $1+9+0+2=12$ e $1+2=3$, per cui in A va scritto 3).

2. Si applica lo stesso procedimento con il secondo fattore, e il risultato va scritto in B (nel nostro esempio, $1+9+6+4=20$ e $2+0=2$, per cui in B va scritto 2).

3. Si moltiplicano i due numeri in A e in B e, se il risultato della moltiplicazione ha più cifre, esse sono ripetutamente sommate come in precedenza: il risultato finale va scritto in C (nel nostro esempio, $3 \times 2 = 6$, per cui in C va scritto 6).

4. Si sommano ripetutamente le cifre del risultato presunto della moltiplicazione, finché non resta un numero a una sola cifra che va scritto in D (nel nostro esempio, $3+7+3+5+5+2+8 = 33$ e $3+3 = 6$, per cui in D va scritto 6).

Nel caso del nostro esempio avremo, quindi, che la croce viene riempita come mostrato nella figura seguente.

3	2
6	6

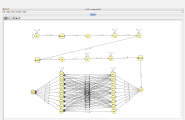
Quando, come in questo caso, i due numeri in basso sono uguali allora la prova ha esito positivo, altrimenti ha esito negativo. Si noti però che se la prova ha esito negativo allora la moltiplicazione è sicuramente errata, mentre se la prova ha esito positivo, il risultato trovato potrebbe essere errato, e differire dal risultato reale per un multiplo di 9. Infatti, se al risultato si sommasse o sottraesse 9 o un suo multiplo, il test sarebbe ancora superato (in quanto $9 \equiv 0 \pmod 9$).

Come ultimo esempio di questo capitolo, vogliamo ora definire una macchina di Turing T che, dato un nastro iniziale contenente una moltiplicazione con un risultato presunto nella forma axb=c, dove a, b e c sono tre numeri interi positivi, termini in uno stato finale se e solo se la prova del nove ha esito positivo. Ad esempio, se l'input è la stringa `1902x1964=3735528`, allora la macchina deve terminare in uno stato finale, così come nel caso in cui l'input sia la stringa `1902x1964=3735519`, mentre se l'input è la stringa `1902x1964=3735529`, allora la macchina deve terminare in una configurazione di rigetto.

Per definire T, possiamo fare uso di $T_{\texttt{digitadder}}$ nel modo seguente. Per prima cosa, T calcola la somma s_3 delle cifre del risultato presunto, quindi la somma s_2 delle cifre del secondo fattore e, infine, la somma s_1 delle cifre del primo fattore: notiamo che, così facendo, possiamo eseguire le tre somme direttamente sulla stringa in ingresso, senza dover copiare i vari numeri in un'altra parte del nastro. Successivamente, T calcola il prodotto modulo 9 di s_1 per s_2 e verifica se il risultato è uguale a s_3. Sapendo che $t_{T_{\texttt{digitadder}}}(n) \in O(n)$, abbiamo che anche il numero di passi eseguiti da T è lineare nella lunghezza della stringa in ingresso, ovvero $t_T(n) \in O(n)$.

JFLAP: macchina per la prova del nove

Osserva, sperimenta e verifica la macchina
`nineproof.jff`



Esercizi

Esercizio 1.1. Si definisca una macchina di Turing T con un solo nastro che, dato in input un numero intero non negativo rappresentato in forma binaria, produca in output la stringa `0`. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.2. Si definisca una macchina di Turing T con un solo nastro che, data in input una sequenza binaria x, produca in output due copie di x separate da un simbolo `x`. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.3. Si definisca una macchina di Turing T con un solo nastro che, data in input una sequenza binaria, termini in uno stato finale se e solo se tale sequenza contiene un ugual numero di `0` e di `1`. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.4. Si definisca una macchina di Turing T con un solo nastro che, data in input una sequenza binaria x, termini in uno stato finale se e solo se tale sequenza è palindroma, ovvero se e solo se $x_1 \cdots x_n = x_n \cdots x_1$. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.5. Si definisca una macchina di Turing T con un solo nastro che, date in input due sequenze binarie x e y separate dal simbolo $\square$, produca in output la sequenza binaria corrispondente alla disgiunzione bit a bit di x e y. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.6. Si definisca una macchina di Turing T con un solo nastro che, data in input la rappresentazione binaria di un numero intero non negativo x, produca in output la sequenza di $x+1$ simboli `1`. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.7. Si definisca una macchina di Turing T con un solo nastro che, data in input la rappresentazione binaria di due numeri interi non negativi x e y, produca in output la rappresentazione binaria del minimo tra x e y. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.8. Data una stringa x sull'alfabeto $\Sigma = \{0, 1, 2\}$, la stringa ordinata corrispondente a x è la stringa su Σ ottenuta a partire da x mettendo tutti i simboli `0` prima dei simboli `1` e tutti questi ultimi prima dei simboli `2`: ad esempio, la stringa ordinata corrispondente a `012021102120201210` è la stringa `000000111111222222`. Si definisca una macchina di Turing T con un solo nastro che, data in input una stringa x sull'alfabeto Σ, termini producendo in output la sequenza ordinata corrispondente a x. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.9. Si definisca una macchina di Turing T con un solo nastro che, dati in input $(n+1)$ numeri interi non negativi $i, x_1, \ldots, x_n$ con $1 \leqslant i \leqslant n$ rappresentati in forma binaria e separati dal simbolo $\square$, produca in output la rappresentazione binaria di x_i. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.10. Si definisca una macchina di Turing T con un solo nastro che, data in input la rappresentazione binaria di un numero intero non negativo x, termini nello stato finale se e solo se x è un multiplo di 3 oppure è un multiplo di 7. Si determini, inoltre, la funzione $t_T(n)$.

Esercizio 1.11. A partire dal sito `http://www.di.unipi.it/settcult/` è possibile accedere all'elenco degli esercizi delle passate edizioni della gara con le macchine di Turing organizzata dal Dipartimento di Informatica dell'Università di Pisa (da cui è stato tratto il problema della prova del nove). Si svolgano quanti più esercizi possibile di quelli inclusi in tale elenco.

Esercizio 1.12. Definire una macchina di Turing T con 2 nastri che, data in input una stringa binaria x, termini producendo in output la sequenza ordinata corrispondente a x e tale che $t_T(n) \in O(n)$.

Esercizio 1.13. Definire una macchina di Turing T con 2 nastri che calcoli la rappresentazione binaria di un numero intero positivo a partire dalla sua rappresentazione unaria e tale che $t_T(n) \in O(n \log n)$.

Esercizio 1.14. Definire una macchina di Turing T con 3 nastri che calcoli la somma di due numeri interi non negativi codificati in binario e separati dal simbolo + e tale che $t_T(n) \in O(n)$.

Esercizio 1.15. Per ciascuna macchina di Turing T definita in uno degli Esercizi 1.1-1.10, si definisca un'equivalente macchina di Turing T' multi-nastro e si confronti $t_T(n)$ con $t_{T'}(n)$.

Esercizio 1.16. Si definisca una macchina di Turing T con due nastri che, data in input una sequenza di simboli dell'alfabeto $\{(,),[,],\{,\}\}$, termini nello stato finale se e solo se le parentesi sono bilanciate e tale che $t_T(n) \in O(n)$.

Capitolo 2

La macchina di Turing universale

SOMMARIO

In questo capitolo mostriamo come il modello delle macchine di Turing, pur nella sua semplicità, consenta di definire una macchina di Turing universale, ovvero quanto di più simile vi possa essere agli odierni calcolatori: tale macchina, infatti, è in grado di simulare il comportamento di una qualunque altra macchina con input una qualunque stringa. Allo scopo di definire la macchina di Turing universale, introdurremo inoltre il concetto di codifica di una macchina di Turing (dopo aver mostrato come non sia restrittivo considerare solo macchine di Turing con un alfabeto di lavoro costituito da tre simboli). Concluderemo, quindi, il capitolo dimostrando che diverse ulteriori varianti del modello di base delle macchine di Turing sono a esso computazionalmente equivalenti.

2.1 Macchine di Turing con alfabeto limitato

NELLA MAGGIOR parte degli esempi di macchine di Turing visti fino a ora, l'alfabeto di lavoro era costituito da un numero molto limitato di simboli (sempre al di sotto della decina). La domanda che possiamo porci è se limitare ulteriormente tale alfabeto riduca le potenzialità di calcolo del modello. In particolare, consideriamo il caso in cui l'alfabeto di lavoro sia costituito, oltre che dal simbolo $\square$, dai soli simboli `0` e `1`. Chi ha già familiarità con l'informatica, sa bene che questo è l'alfabeto utilizzato dagli odierni calcolatori: ogni carattere di un diverso alfabeto viene, infatti, associato a un codice binario, che può essere costituito da 8 bit (nel caso del codice ASCII) oppure da 16 bit (nel caso del codice Unicode).

Pertanto, è ragionevole supporre che la restrizione a un alfabeto binario non sia assolutamente limitativa rispetto al potere computazionale di una macchina di Turing. L'unica attenzione che dobbiamo prestare, risiede nel fatto di garantire che la

codifica di una sequenza di simboli non binari sia fatta in modo da consentirne in modo univoco la successiva decodifica.

Se, ad esempio, decidessimo di codificare i tre simboli `A`, `B` e `C` associando a essi le sequenze binarie `0`, `1` e `00`, rispettivamente, allora di fronte alla sequenza `000` non saremmo in grado di dire se tale sequenza sia la codifica di `AAA`, `AC` oppure `CA`. Il problema sta nel fatto che la codifica di `A` è un **prefisso** della codifica di `C`, per cui una volta letta tale codifica non sappiamo dire se ciò che abbiamo letto rappresenta la codifica di `A` oppure l'inizio della codifica di `C`.

Esistono molti modi per definire codici che evitano questo problema: in questo contesto, ci accontentiamo di definirne uno molto semplice e del tutto analogo a quello utilizzato dagli odierni calcolatori. Dato un alfabeto $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ con $n \geqslant 2$, sia $k = \lceil \log_2 n \rceil$. Per ogni i con $1 \leqslant i \leqslant n$, associamo al simbolo σ_i la stringa binaria $c_\Sigma(\sigma_i)$ di k simboli `0` e `1`, ottenuta calcolando la rappresentazione binaria di $i-1$ che fa uso di k bit. Data una stringa $x = x_1 \cdots x_m$ su Σ, indichiamo con $c_\Sigma(x)$ la stringa $c_\Sigma(x_1) \cdots c_\Sigma(x_m)$ di lunghezza km.

Esempio 2.1: codifica binaria di un alfabeto

Consideriamo l'alfabeto

$$\Sigma = \{\sigma_1 = a, \sigma_2 = b, \sigma_3 = c, \sigma_4 = d, \sigma_5 = e\}$$

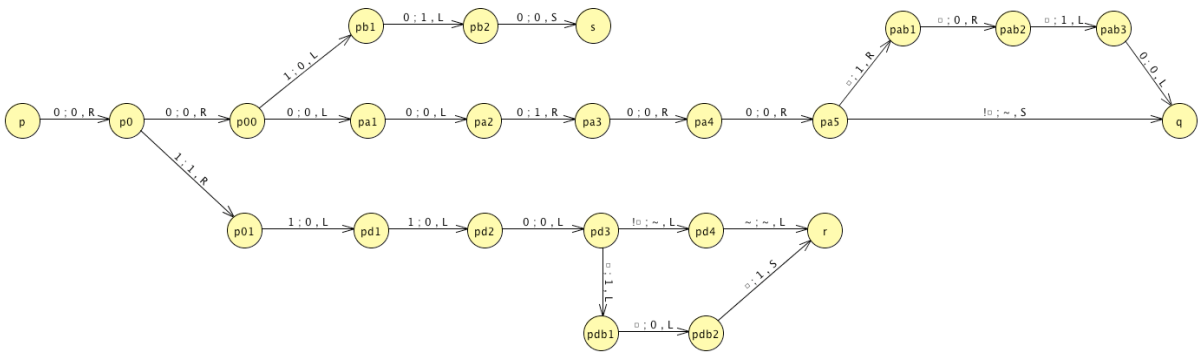
In questo caso, $k = \lceil \log_2 5 \rceil = 3$. Pertanto, $c(a) = c(\sigma_1) = 000$, $c(b) = c(\sigma_2) = 001$, $c(c) = c(\sigma_3) = 010$, $c(d) = c(\sigma_4) = 011$ e $c(e) = c(\sigma_5) = 100$. La codifica binaria della stringa `cade` è uguale a `010000011100`, mentre quella della stringa `debba` è uguale a `011100001001000`.

Data una macchina di Turing T con alfabeto di lavoro $\Gamma = \{\square\} \cup \Sigma$, dove $|\Sigma| \geqslant 2$, possiamo definire una macchina di Turing T' con alfabeto di lavoro $\{\square, 0, 1\}$ tale che valgono le seguenti affermazioni.

- Per ogni $x \in \Sigma^*$, se T con input x non termina, allora T' con input $c_\Sigma(x) \in \{0,1\}^*$ non termina.
- Per ogni $x \in \Sigma^*$, se T con input x termina in una configurazione di rigetto, allora T' con input $c_\Sigma(x) \in \{0,1\}^*$ termina in una configurazione di rigetto.
- Per ogni $x \in \Sigma^*$, se T con input x termina in uno stato finale e produce in output la stringa $y \in \Sigma^*$, allora T' con input $c_\Sigma(x) \in \{0,1\}^*$ termina in uno stato finale e produce in output la stringa $c_\Sigma(y) \in \{0,1\}^*$.

Il comportamento di T' è identico a quello di T con l'unica differenza che ogni singola transizione di T viene simulata mediante $O(k)$ transizioni di T' (dove $k = \lceil \log_2 |\Gamma| \rceil$),

Figura 2.1: le transizioni di una macchina con alfabeto binario.



che consentano di leggere la codifica di un simbolo di Γ, sostituire tale codifica con quella di un altro simbolo di Γ e posizionare in modo corretto la testina. Pertanto, $t_{T'}(n) \in O(t_T(n))$.

Esempio 2.2: macchine di Turing con alfabeto limitato

Consideriamo una macchina di Turing T il cui alfabeto di lavoro sia $\Gamma = \{\square\} \cup \Sigma$ dove Σ è l'alfabeto definito nell'Esempio 2.1, ovvero $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$: supponiamo, inoltre, che la codifica del simbolo $\square$ sia `101`. Se T include (1) una transizione dallo stato `p` allo stato `s` che, leggendo il simbolo `b`, lo sostituisce con il simbolo `c` e non muove la testina, (2) una transizione dallo stato `p` allo stato `q` che, leggendo il simbolo `a`, lo sostituisce con il simbolo `e` e sposta la testina a destra e (3) una transizione dallo stato `p` allo stato `r` che, leggendo il simbolo `d`, lo sostituisce con il simbolo `a` e sposta la testina a sinistra, la macchina T' simula ciascuna di tali transizioni mediante una sequenza di al più 11 transizioni come mostrato nella Figura 2.1 (notiamo come sia necessario, nel caso di uno spostamento a destra o a sinistra, gestire anche la trasformazione di un eventuale simbolo $\square$ nella sua corrispondente codifica binaria).

Possiamo, dunque, concludere che restringendo l'alfabeto di lavoro di una macchina di Turing a tre soli simboli, incluso il simbolo $\square$, non si riduce il potere computazionale delle macchine di Turing. Ovviamente, l'utilizzo di siffatte macchine di Turing richiede che i valori di input e quelli di output siano stringhe binarie. Una macchina di Turing il cui alfabeto di lavoro sia $\{\square, \mathtt{0}, \mathtt{1}\}$ non potrà mai ricevere come stringa di input una sequenza che contenga simboli diversi da `0` e da `1` né potrà mai produrre un output che non sia una sequenza binaria: in altre parole, per poter usare

una tale macchina sarà necessario eseguire una fase di *pre-processing*, in cui l'input originale venga eventualmente codificato in una sequenza binaria, e una fase di *post-processing*, in cui l'output prodotto dalla macchina venga eventualmente decodificato in modo da ottenere l'output desiderato.

2.2 Codifica delle macchine di Turing

FINO A ora abbiamo parlato di macchine di Turing intendendo formalizzare con tale modello di calcolo la definizione del concetto di algoritmo. In questo capitolo e in quello successivo, il nostro scopo sarà quello di caratterizzare il potere computazionale delle macchine di Turing mostrandone sia le capacità che i limiti. In particolare, in questo capitolo mostreremo l'esistenza di una macchina di Turing universale in grado di simulare una qualunque altra macchina di Turing a partire da una descrizione di quest'ultima.

Per poter definire una macchina di Turing universale è dunque necessario stabilire anzitutto in che modo una macchina di Turing da simulare possa essere codificata mediante una stringa. Tale codifica può essere fatta in tanti modi diversi e, in questo paragrafo, ne introdurremo uno che assume di dover operare solo su macchine di Turing con alfabeto di lavoro costituito dai simboli $\square$, `0` e `1` (abbiamo visto, nel paragrafo precedente, come tale restrizione non costituisca alcuna perdita di generalità).

Per codificare una macchina di Turing, dobbiamo decidere come codificare i simboli del suo alfabeto di lavoro, i possibili stati della sua CPU e i possibili movimenti della sua testina. Nel fare ciò, possiamo fare uso di un qualunque insieme di simboli, in quanto la trasformazione di una macchina di Turing universale che abbia un alfabeto di lavoro con più di tre simboli in una macchina con alfabeto di lavoro uguale a $\{\square, \texttt{0}, \texttt{1}\}$ può essere successivamente realizzata in modo simile a quanto visto nel paragrafo precedente.

Codifichiamo, pertanto, i simboli dell'alfabeto di lavoro di una macchina di Turing T nel modo seguente: la codifica di `0` è `Z`, quella di `1` è `U` e quella di $\square$ è `B`. I movimenti della testina di T sono codificati mediante i simboli `L` (movimento a sinistra), `S` (nessun movimento) e `R` (movimento a destra). Infine, se l'insieme degli stati di T (inclusi quello iniziale e quelli finali) ha cardinalità $\mathtt{k}$, allora, per ogni $\mathtt{i}$ con $0 \leqslant \mathtt{i} \leqslant \mathtt{k}-1$, l'$(\mathtt{i}+1)$-esimo stato sarà codificato dalla rappresentazione binaria di $\mathtt{i}$: nel seguito, senza perdita di generalità, assumeremo sempre che il primo stato (ovvero quello con codifica `0`) sia lo stato iniziale e che il secondo stato (ovvero quello con codifica `1`) sia l'*unico* stato finale.[1]

[1] Se la macchina di Turing non ha stati finali, possiamo sempre aggiungerne uno che non sia raggiungibile da nessun altro stato, mentre nel caso in cui la macchina di Turing abbia più di uno stato finale possiamo sempre farli "convergere" in un unico stato finale attraverso delle transizioni fittizie.

Ciascuna transizione di T viene codificata giustapponendo le codifiche dei vari elementi che la compongono, ovvero dello stato di partenza, del simbolo letto, dello stato di arrivo, del simbolo scritto e del movimento della testina. L'intero programma della macchina T viene quindi codificato elencando tutte le codifiche delle possibili transizioni una di seguito all'altra. Osserviamo che tale codifica non è ambigua, in quanto utilizza alfabeti diversi per la codifica degli stati (che fa uso dei simboli `0` e `1`), per quella dei movimenti (che fa uso dei simboli `L`, `R` e `S`) e per quella dei simboli (che fa uso dei simboli `B`, `U` e `Z`). Osserviamo inoltre che alla stessa macchina possono essere associate codifiche diverse, in quanto gli stati possono essere numerati in modo diverso (a parte quello iniziale e quello finale) e le transizioni possono essere elencate in un diverso ordine.

Esempio 2.3: codifica di una macchina di Turing

Consideriamo la macchina di Turing mostrata nella Figura 1.2, che calcola il complemento bit a bit di una sequenza binaria e la cui rappresentazione tabellare è la seguente (una volta rinominati gli stati in modo da soddisfare le assunzioni fatte in precedenza relativamente allo stato iniziale e a quello finale).

stato	simbolo	stato	simbolo	movimento	codifica
q0	0	q0	1	R	0Z0UR
q0	1	q0	0	R	0U0ZR
q0	□	q2	□	R	0B10BR
q2	0	q2	0	L	10Z10ZL
q2	1	q2	1	L	10U10UL
q2	□	q1	□	R	10B1BR

La codifica di una transizione è indicata nell'ultima colonna della tabella, per cui una codifica dell'intera macchina di Turing è `0Z0UR0U0ZR0B10BL10Z10ZL10U10UL10B1BR`. La stessa macchina di Turing potrebbe essere rappresentata dalla seguente tabella.

stato	simbolo	stato	simbolo	movimento	codifica
q0	□	q2	□	R	0B10BR
q0	1	q0	0	R	0U0ZR
q0	0	q0	1	R	0Z0UR
q2	□	q1	□	R	10B1BR
q2	1	q2	1	L	10U10UL
q2	0	q2	0	L	10Z10ZL

In tal caso, la codifica sarebbe `0B10BL0U0ZR0Z0UR10B1BR10U10UL10Z10ZL` che è diversa da quella precedente ma che, comunque, rappresenta la stessa macchina.

2.3 La macchina di Turing universale

LA MACCHINA di Turing universale fu proposta la prima volta da Turing stesso e ha svolto un ruolo importante nello stimolare lo sviluppo di calcolatori basati sulla cosiddetta architettura di von Neumann, ovvero calcolatori la cui memoria conserva sia i dati che i programmi da eseguire sui dati stessi. Esistono diverse definizioni di tale macchina, che differiscono per il numero di nastri, per il numero di simboli e per il numero di stati utilizzati: in questo paragrafo, ne definiamo una basata sulla codifica delle macchine di Turing introdotta nel paragrafo precedente. In particolare, la nostra **macchina di Turing universale** U farà uso di tre nastri: il primo nastro inizialmente conterrà la codifica c_{T} di una macchina di Turing T seguita da un simbolo `;` e dalla stringa binaria x in input, il secondo nastro a regime memorizzerà la codifica dello stato corrente di T, mentre il terzo e ultimo nastro servirà a simulare il nastro di lavoro di T. La macchina U opera nel modo seguente.

1. Copia sul terzo nastro l'input x codificato mediante i simboli `U` e `Z`.
2. Inizializza il contenuto del secondo nastro con la codifica dello stato iniziale di T.
3. In base allo stato contenuto nel secondo nastro e il simbolo letto sul terzo nastro, cerca sul primo nastro una transizione che possa essere applicata. Se tale transizione non viene trovata, termina in una configurazione di rigetto.
4. Altrimenti, applica la transizione modificando il contenuto del terzo nastro e aggiornando il secondo nastro in base al nuovo stato di T.
5. Se il nuovo stato è uno stato finale di T, termina nell'unico stato finale di U. Altrimenti, torna al Passo 3.

La descrizione formale di U sarà data ricorrendo alla programmazione procedurale, ovvero all'uso di sotto-macchine: in particolare, definiremo una sotto-macchina per ciascuno dei cinque passi sopra descritti.

2.3.1 Il primo passo della macchina universale

La sotto-macchina che realizza il primo passo (ovvero la copia sul terzo nastro dell'input x codificato mediante i simboli `U` e `Z`) è definita dal grafo delle transizioni mostrato nella Figura 2.2 e opera nel modo seguente.

1. Posiziona la testina sul primo simbolo alla destra del simbolo `;`, il quale viene cancellato.

Figura 2.2: il primo passo della macchina di Turing universale.

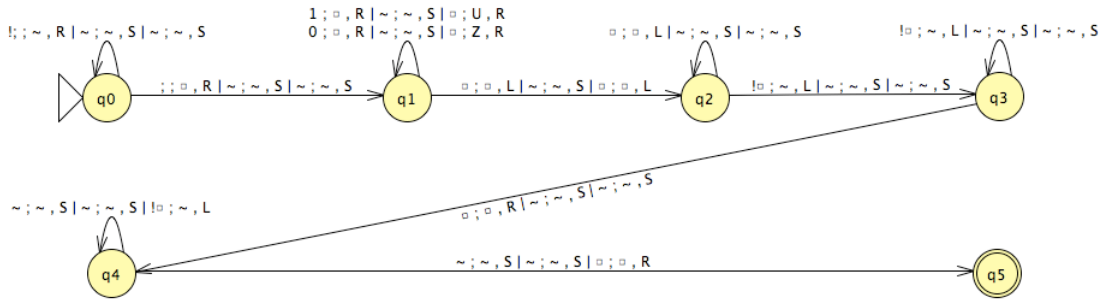
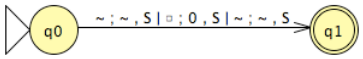


Figura 2.3: il secondo passo della macchina di Turing universale.



2. Scorre il primo nastro verso destra e, per ogni simbolo diverso da □, lo copia sul terzo nastro (spostando la testina di questo nastro a destra) e lo cancella.

3. Posiziona la testina del primo nastro e quella del terzo nastro sul simbolo diverso da □ più a sinistra.

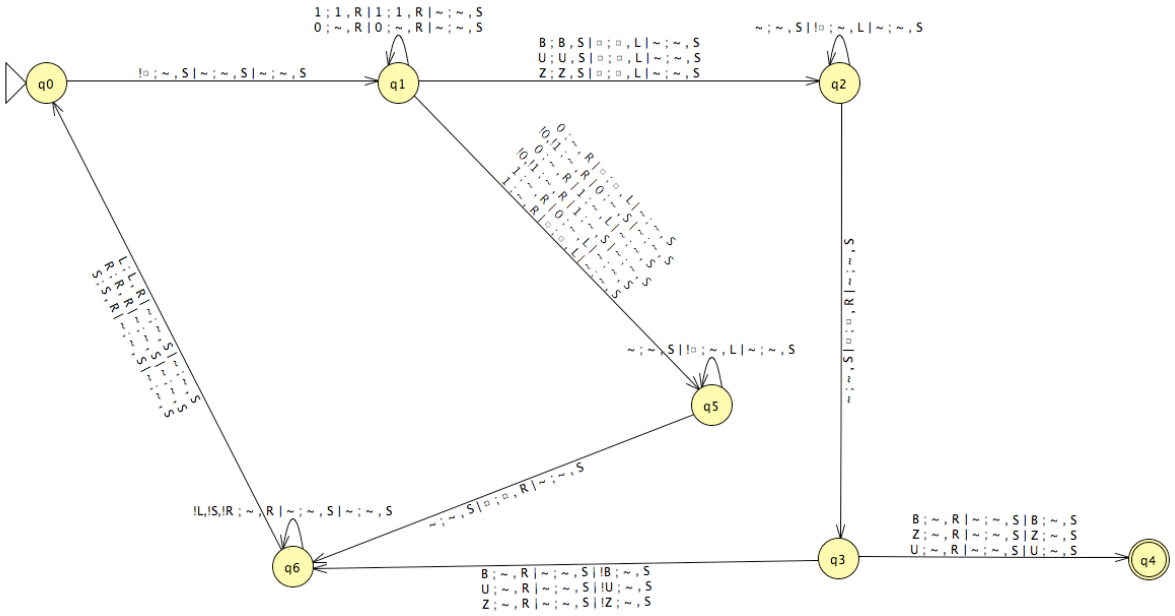
2.3.2 Il secondo passo della macchina universale

La sotto-macchina che esegue il secondo passo della macchina di Turing universale, ovvero che inizializza il contenuto del secondo nastro con la codifica dello stato iniziale, deve semplicemente scrivere su tale nastro il simbolo `0` e posizionare la testina su di esso: ricordiamo infatti che, in base alle assunzioni fatte nel paragrafo precedente, lo stato iniziale di una qualunque macchina di Turing è quello di indice 0. Il grafo delle transizioni di tale sotto-macchina è mostrato nella Figura 2.3.

2.3.3 Il terzo passo della macchina universale

La ricerca della transizione che deve essere eventualmente applicata è probabilmente il passo più complicato da realizzare della macchina di Turing universale e può esse-

Figura 2.4: il terzo passo della macchina di Turing universale.



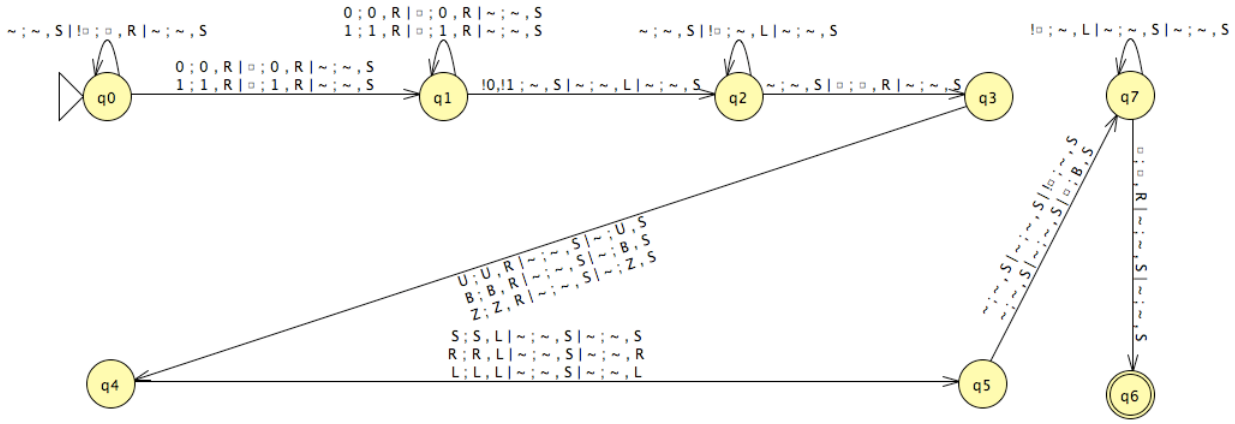
re effettuata risolvendo, anzitutto, un problema di *pattern matching* tra il contenuto del primo nastro e quello del secondo nastro, ovvero cercando sul primo nastro la prima occorrenza del contenuto del secondo nastro. Una volta trovata tale occorrenza la ricerca prosegue verificando se il simbolo immediatamente successivo è uguale al simbolo attualmente scandito sul terzo nastro: in tal caso, la sotto-macchina può terminare posizionando la testina sulla seconda parte della transizione appena identificata. Altrimenti, deve proseguire cercando sul primo nastro un'altra occorrenza del contenuto del secondo nastro.

In particolare, la sotto-macchina che realizza il terzo passo della macchina di Turing universale è definita dal grafo delle transizioni mostrato nella Figura 2.4 e opera nel modo seguente.

1. Scorre il primo e il secondo nastro verso destra fintanto che trova simboli uguali (osserviamo che il secondo nastro include solo simboli `0` e `1`).

2. Se non incontra un simbolo diverso da `0` e `1` su entrambi i nastri, allora posiziona nuovamente la testina del secondo nastro sul primo simbolo a sinistra

Figura 2.5: il quarto passo della macchina di Turing universale.



diverso da □, posiziona la testina del primo nastro sul primo simbolo della transizione successiva a quella attualmente esaminata (ovvero, sul primo simbolo successivo a un simbolo incluso in {L, R, S}) e torna al passo precedente.

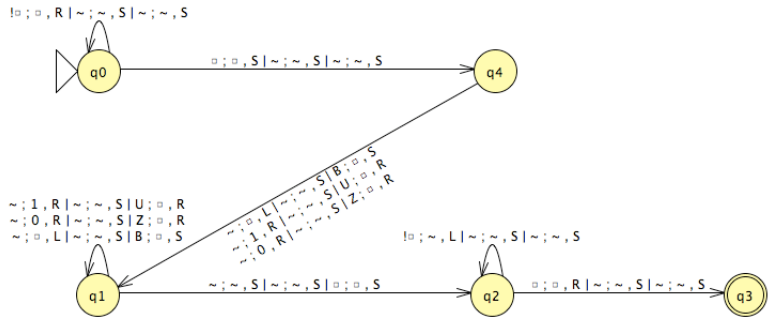
3. Se incontra un simbolo diverso da 0 e 1 su entrambi i nastri, allora posiziona nuovamente la testina del secondo nastro sul primo simbolo a sinistra diverso da □ (preparandosi, così, a un'eventuale successiva ricerca) e verifica se il simbolo letto sul primo nastro è uguale a quello letto sul terzo nastro. In tal caso, sposta la testina del primo nastro di una posizione a destra e termina. Altrimenti, posiziona la testina del primo nastro sul primo simbolo della transizione successiva a quella attualmente esaminata e torna al primo passo.

2.3.4 Il quarto passo della macchina universale

Trovata la transizione da applicare, l'effettiva esecuzione della transizione stessa viene realizzata mediante una semplice sotto-macchina, il cui grafo delle transizioni è mostrato nella Figura 2.5 e che opera nel modo seguente.

1. Cancella l'intero contenuto del secondo nastro e copia su di esso la sequenza di simboli 0 e 1 contenuta sul primo nastro e il cui primo simbolo è attualmente scandito (al termine della copia la testina del primo nastro si troverà sul simbolo immediatamente a destra della sequenza binaria). Quindi, posiziona la testina del secondo nastro sul primo simbolo a sinistra diverso da □.

Figura 2.6: il quinto passo della macchina di Turing universale.



2. Sostituisce il simbolo attualmente scandito sul terzo nastro con quello attualmente scandito sul primo nastro e sposta la testina di quest'ultimo nastro di una posizione a destra.

3. In base al simbolo letto sul primo nastro, sposta la testina del terzo nastro: se il nuovo simbolo letto su tale nastro è □, allora lo sostituisce con il simbolo `B`.

4. Posiziona la testina del primo nastro sul primo simbolo a sinistra diverso da □ (preparandosi, così, a un'eventuale nuova ricerca di una transizione) e termina.

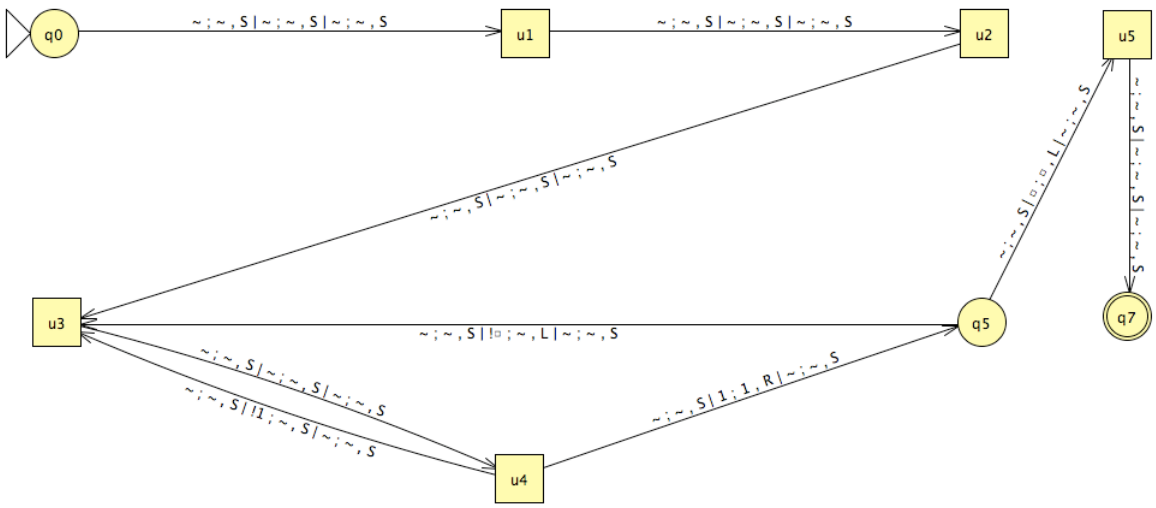
2.3.5 Il quinto passo della macchina universale

Terminata l'esecuzione della transizione, la macchina di Turing universale deve verificare se lo stato attuale della macchina di Turing simulata sia quello finale: in base alle assunzioni fatte nel paragrafo precedente, a tale scopo è sufficiente verificare se il contenuto attuale del secondo nastro è uguale alla stringa `1`. Se così non è allora la macchina di Turing universale torna a eseguire il terzo passo, altrimenti cancella il contenuto del primo nastro (anche se ciò non sarebbe strettamente necessario) e vi copia l'output della macchina simulata, decodificato facendo uso di simboli `0` e `1`. In particolare, quest'ultima operazione viene eseguita dalla sotto-macchina il cui grafo delle transizioni è mostrato nella Figura 2.6.

2.3.6 La macchina universale

In conclusione, facendo riferimento a tutte le sotto-macchine sopra definite, possiamo ora definire formalmente la macchina di Turing universale `U` mediante il grafo delle

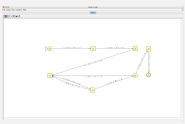
Figura 2.7: la macchina di Turing universale.



transizioni mostrato nella Figura 2.7. Osserviamo che, sebbene tale macchina faccia uso di tre nastri e di un alfabeto di lavoro contenente dieci simboli (ovvero □, `0`, `1`, `B`, `U`, `Z`, `L`, `R`, `S` e `;`), in base a quanto abbiamo mostrato nel capitolo precedente e nel primo paragrafo di questo capitolo, essa può essere trasformata in una macchina di Turing con un solo nastro e con un alfabeto di lavoro contenente i tre soli simboli □, `0` e `1` (ovviamente tale trasformazione richiede che la codifica delle macchina di Turing mostrata nel paragrafo precedente sia pre-elaborata in modo da usare solo simboli binari). Ciò implica che la macchina di Turing universale U può simulare se stessa con input la codifica di un'altra macchina di Turing T e una sequenza di simboli binari x: il risultato di tale simulazione sarà uguale all'esecuzione di T con input x.

JFLAP: macchina di Turing universale

Osserva, sperimenta e verifica la macchina
`u.jff`



Osserviamo, infine, che ogni passo della macchina di Turing T simulata da quella universale richiede da parte di quest'ultima un numero di passi che dipende esclusi-

vamente dalla lunghezza della codifica di T: in altre parole, se fissiamo la macchina di Turing T da simulare, il numero di passi eseguiti da U è proporzionale a quello dei passi eseguiti da T. In generale, però, per ogni stringa x di lunghezza n, il primo, il quarto e il quinto passo di U richiedono un numero di passi lineare in n, mentre il secondo passo ne richiede un numero costante. Il terzo passo, invece, richiede un numero di passi quadratico in n, in quanto, dopo ogni fallita ricerca della transizione da eseguire, la macchina deve "riavvolgere" il secondo nastro. Pertanto, possiamo concludere che $t_U(n) \in O(n^2)$.

2.4 Varianti delle macchine di Turing

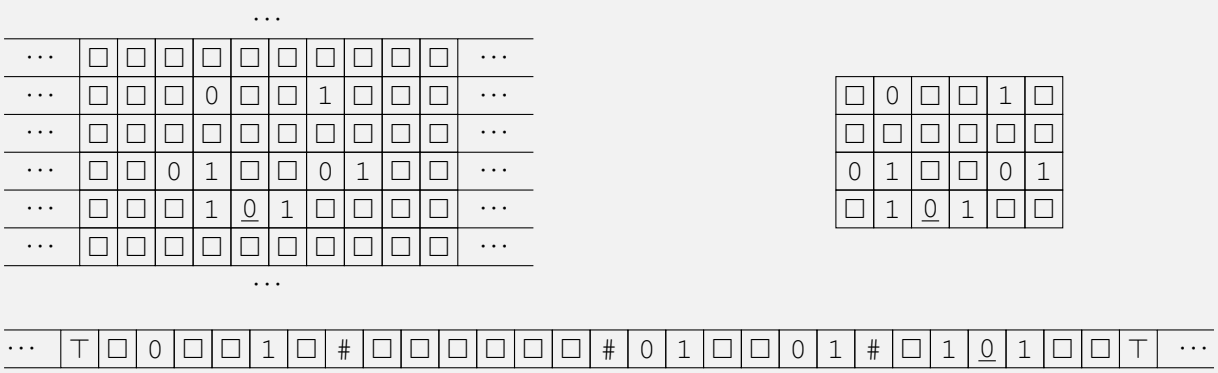
IN QUESTO paragrafo mostreremo come il modello di calcolo delle macchine di Turing sia sufficientemente *robusto* da non modificare il suo potere computazionale, cambiando alcuni aspetti della sua definizione. Abbiamo già visto, in realtà, come ciò sia vero nel caso in cui si modifichi il numero di nastri oppure il numero di simboli dell'alfabeto di lavoro: in questo paragrafo, analizziamo due ulteriori *estensioni* della macchina di Turing, ovvero quella con nastri bidimensionali e quella multi-traccia, e due ulteriori *restrizioni*, ovvero quella in cui non è consentito alla testina di stare ferma e quella con un nastro semi-infinito. In tutti questi casi, dimostreremo che il nuovo modello di calcolo è equivalente a quello introdotto nel precedente capitolo: in generale, questi risultati consentono di utilizzare, in qualunque momento, la variante di macchina di Turing che riteniamo più opportuna, eventualmente semplificando in tal modo la dimostrazione di altri risultati (ciò è esattamente quello che abbiamo fatto nel paragrafo precedente definendo la macchina di Turing universale come una macchina con tre nastri).

2.4.1 Macchine di Turing con nastri bidimensionali

Una macchina di Turing con **nastro bidimensionale** opera come una macchina di Turing ordinaria, con l'unica differenza che il nastro, sempre suddiviso in celle, si sviluppa sia in orizzontale che in verticale e che la testina può quindi eseguire cinque tipi di movimento: oltre ai tre movimenti disponibili in una macchina di Turing ordinaria (ovvero L, S, R), la testina può infatti muoversi anche verso l'alto oppure verso il basso.

Per dimostrare che una macchina di Turing T con un nastro bidimensionale può essere simulata da una macchina di Turing ordinaria T′, osserviamo anzitutto che, in ogni istante della sua esecuzione, solo una parte finita del nastro bidimensionale di T contiene simboli diversi da □ (si veda la parte in alto a sinistra della Figura 2.8 dove abbiamo indicato la posizione della testina sottolineando il simbolo da essa

Figura 2.8: nastro bidimensionale e sua rappresentazione unidimensionale.



scandito).[2] In particolare, chiameremo *rettangolo di interesse* il più piccolo rettangolo che contenga tale porzione finita del nastro bidimensionale e che includa la cella attualmente scandita dalla testina (si veda la parte in alto a destra della figura).

Tale rettangolo può essere rappresentato sul nastro unidimensionale di T' in modo simile a quanto abbiamo fatto nel corso della simulazione di una macchina di Turing multi-nastro mediante una con un nastro singolo (si veda la parte in basso della figura): in particolare, possiamo rappresentare ciascuna riga del rettangolo di interesse una dopo l'altra, separando due righe consecutive con un simbolo speciale (ad esempio, il simbolo #) e racchiudendo l'intera rappresentazione del rettangolo con un altro simbolo speciale (ad esempio, il simbolo $\top$).

La macchina T' usa tre nastri e il suo alfabeto di lavoro contiene, oltre ai simboli dell'alfabeto di lavoro di T e ai simboli # e $\top$, un insieme di simboli che consentano di determinare la cella attualmente scandita: in particolare, per ogni simbolo σ dell'alfabeto di lavoro di T, l'alfabeto di lavoro di T' include il simbolo $\underline{\sigma}$.

Con input la stringa $x = x_1 \cdots x_n$, la macchina T' per prima cosa crea sul primo nastro la rappresentazione del rettangolo corrispondente alla configurazione iniziale di T con input x, ovvero la stringa $\top \underline{x_1} x_2 \cdots x_n \top$. Quindi, scorre il primo nastro nastro alla ricerca del simbolo attualmente scandito e determina ed esegue la transizione di T corrispondente a tale simbolo e al suo stato attuale (di cui T' terrà memoria all'interno del proprio stato). Nell'eseguire la transizione, la macchina T' deve prestare

[2] Abbiamo fatto un'analoga osservazione nel momento in cui abbiamo introdotto, nel primo capitolo, il concetto di configurazione di una macchina di Turing.

attenzione alla possibilità che il movimento della testina corrisponda a un cambio di riga del rettangolo di interesse oppure a un allargamento dello stesso.

Nel primo caso, facendo uso del secondo nastro, T' memorizza la distanza della cella attualmente scandita da T dal bordo sinistro del rettangolo e utilizza tale distanza per determinare la posizione, all'interno della riga sopra o sotto quella attuale, della cella su cui la testina di T si deve spostare.

Ad esempio, facendo riferimento alla configurazione mostrata nella parte bassa della Figura 2.8 e supponendo che T leggendo il simbolo 0, nello stato attuale, muova la testina verso l'alto, T' determina che la distanza della cella attualmente scandita dalla testina di T dal primo simbolo $\#$ alla sua sinistra è pari a 3 e, quindi, che la nuova cella scandita dalla testina di T è quella che si trova a distanza 3 dal successivo simbolo $\#$ a sinistra: tale simbolo è $\square$, il quale viene sostituito dal simbolo $\underline{\square}$.

Nel caso in cui il movimento della testina corrisponda a un allargamento del rettangolo di interesse, la macchina T', oltre a dover memorizzare la distanza della cella attualmente scandita da T dal bordo sinistro del rettangolo, dovrà anche creare la nuova riga oppure la nuova colonna del rettangolo e, successivamente, utilizzare la distanza precedentemente calcolata per determinare la posizione della cella su cui la testina di T si deve spostare.

Ad esempio, facendo riferimento alla configurazione mostrata nella parte bassa della Figura 2.8 e supponendo che T leggendo il simbolo 0, nello stato attuale, muova la testina verso il basso, T' determina che la distanza della cella attualmente scandita dalla testina di T dal primo simbolo $\#$ alla sua sinistra è pari a 3. Spostandosi verso destra alla ricerca del simbolo $\#$, T' si accorge che il rettangolo termina nel momento in cui incontra il simbolo $\top$: in tal caso, facendo uso del terzo nastro, T' determina la lunghezza l attuale delle righe e crea una nuova riga alla destra del simbolo $\top$ (che sostituisce con il simbolo $\#$) contenente l simboli $\square$ seguiti dal simbolo $\top$ (di questi l simboli $\square$, il terzo viene sostituito dal simbolo $\underline{\square}$).

Nel caso in cui l'allargamento del rettangolo di interesse implichi la creazione di una nuova colonna (il che può accadere solo se la testina di T si sposta a sinistra oppure a destra), T' dovrà aggiungere un simbolo $\square$ all'estremità sinistra oppure destra di ciascuna riga del rettangolo: il simbolo aggiunto alla riga attuale dovrà poi essere sostituito dal simbolo $\underline{\square}$.

Per ogni stringa x di lunghezza n, la simulazione da parte della macchina T' con due nastri di un passo della macchina T con nastro bidimensionale con input x richiede un numero costante di scansioni del rettangolo di interesse: poiché tale rettangolo può contenere al più $t_T^2(n)$ celle (nel caso in cui una nuova cella venga scandita a ogni passo), abbiamo che $t_{T'}(n) \in O(t_T^3(n))$, ovvero che la simulazione richiede un numero di passi cubico rispetto al numero di passi eseguiti dalla macchina di Turing con nastro bidimensionale.

2.4.2 Macchine di Turing multi-traccia

Una macchina di Turing **multi-traccia** opera come una macchina di Turing ordinaria, con l'unica differenza che le celle del nastro sono suddivise in un numero costante di **tracce** e che, scandendo una cella del nastro, la testina legge contemporaneamente i simboli di tutte le tracce. A una transizione di una macchina di Turing con k tracce, quindi, è associata un'etichetta formata da una lista di triple

$$((\sigma_1,\ldots,\sigma_k),(\tau_1,\ldots,\tau_k),m)$$

I simboli $\sigma_1,\ldots,\sigma_k$ e $\tau_1,\ldots,\tau_k$ che appaiono all'interno di una tripla dell'etichetta di una transizione indicano, rispettivamente, i k simboli attualmente letti sulle k tracce e i k simboli da scrivere, mentre il valore m specifica il movimento che deve essere effettuato dall'unica testina.

Notiamo come le macchine di Turing multi-traccia differiscono da quelle multi-nastro per il fatto che le prime hanno un unico nastro e, quindi, un'unica testina, mentre le seconde hanno più nastri e, quindi, più testine tra di loro indipendenti. Questa differenza rende la simulazione di una macchina di Turing multi-traccia T da parte di una macchina di Turing ordinaria T' molto più facile di quella descritta nel caso delle macchine di Turing multi-nastro: in effetti, la simulazione può essere realizzata semplicemente definendo in modo opportuno l'alfabeto di lavoro di T'. In particolare, se Σ_T è l'alfabeto di lavoro di T, allora l'alfabeto di lavoro di T' sarà il seguente.

$$\Sigma_{T'} = \{\sigma_{\sigma_1\cdots\sigma_k} : \sigma_i \in \Sigma_T \text{ per ogni } i \text{ con } 1 \leqslant i \leqslant k\}$$

A ogni transizione di T dallo stato q allo stato p a cui sia associata un'etichetta formata da una lista di triple

$$((\sigma_1,\ldots,\sigma_k),(\tau_1,\ldots,\tau_k),m)$$

corrisponde una transizione di T' dallo stato q allo stato p a cui è associata l'etichetta formata dalla lista di triple

$$(\sigma_{\sigma_1\cdots\sigma_k},\sigma_{\tau_1\cdots\tau_k},m)$$

È immediato verificare che $t_{T'}(n) \in O(t_T(n))$: in effetti, ogni passo di T viene simulato da T' con un solo passo.

2.4.3 Macchine di Turing con testina limitata

Nei due paragrafi precedenti abbiamo considerato due possibili estensioni del modello della macchina di Turing definito nel primo capitolo, mostrando come tali estensioni non influenzino in alcun modo il potere computazionale del modello di calcolo

(se non si tiene conto del numero di passi eseguiti). In questo e nel prossimo paragrafo, considereremo invece due restrizioni del modello della macchina di Turing ottenute ponendo alcuni vincoli sui movimenti della testina e sulla tipologia di nastro utilizzata.

Una domanda naturale che è lecito porsi consiste nell'imporre che la testina di una macchina di Turing T possa eseguire solo un sotto-insieme delle possibili mosse del modello originario, e nel verificare se tale restrizione modifichi il potere computazionale del modello di calcolo. Supponiamo, ad esempio, che la testina non possa muoversi, ma solo rimanere sulla posizione in cui si trova (ovvero, l'insieme dei movimenti sia ridotto al solo $\mathtt{S}$). È ovvio che in tal caso, non potendo neanche leggere completamente la stringa in input, tale restrizione produce un modello di calcolo più debole di quello originario.

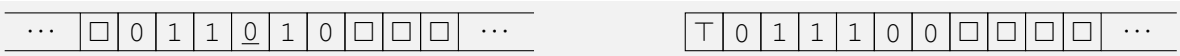
Se supponiamo che la testina possa solo muoversi a destra (ovvero, l'insieme dei movimenti sia ridotto al solo $\mathtt{R}$), la macchina di Turing così definita risulta essere un automa a stati finiti. Come vedremo nella seconda parte di queste dispense, tali automi sono in grado di decidere solo linguaggi generati da grammatiche regolari: ad esempio, non sono in grado di decidere se una stringa appartiene al linguaggio $L = \{0^n 1^n : n \geqslant 0\}$ (si veda il secondo paragrafo del capitolo precedente). Pertanto, anche questa restrizione produce un modello di calcolo più debole di quello originario.

Se restringiamo l'insieme dei movimenti della testina a $\{\mathtt{S},\mathtt{R}\}$, allora possiamo facilmente convincerci del fatto che tale modello di calcolo è di poco più potente di quello in cui la testina può solo muoversi a destra. Supponiamo, infatti, che trovandosi nello stato q e leggendo un simbolo σ la macchina di Turing esegue un numero (limitato) di passi in cui la testina non viene spostata a cui fa seguito un passo in cui la testina scrive il simbolo τ e si sposta a destra e la macchina entra nello stato p. Allora tale sequenza di passi può essere direttamente simulata da un solo passo in cui la testina scrive il simbolo τ e si sposta a destra e la macchina entra nello stato p. L'unica differenza tra una macchina in cui la testina può anche stare ferma e una in cui può solo spostarsi a destra consiste nella possibilità di produrre diversi output.

Ad esempio, con input la stringa x, una macchina la cui testina può anche stare ferma è in grado di produrre in output la stringa x stessa, mentre una macchina la cui testina può solo muoversi a destra non è in grado di produrre tale output. Tale differenza, tuttavia, non è ovviamente sufficiente a costruire, ad esempio, una macchina in grado decidere se una stringa appartiene al linguaggio $L = \{0^n 1^n : n \geqslant 0\}$: pertanto, anche questa restrizione produce un modello di calcolo più debole di quello originario.

In modo analogo, possiamo concludere che, restringendo l'insieme dei movimenti della testina a $\{\mathtt{L}\}$ oppure a $\{\mathtt{S},\mathtt{L}\}$ si ottiene un modello di calcolo più debole delle macchine di Turing ordinarie. Rimane da considerare il caso in cui la testina non

Figura 2.9: rappresentazione di un nastro infinito mediante uno semi-infinito.



possa rimanere ferma (ovvero, l'insieme dei movimenti sia $\{\mathtt{L}, \mathtt{R}\}$): in tal caso, il corrispondente modello di calcolo è equivalente a quello originario. In effetti, il "non movimento" può banalmente essere simulato da uno spostamento a destra seguito da uno a sinistra (o viceversa).

2.4.4 Macchine di Turing con nastro semi-infinito

L'ultima restrizione che consideriamo riguarda nuovamente la struttura del nastro di una macchina di Turing. In particolare, consideriamo il caso in cui tale nastro non si prolunghi indefinitamente da entrambi i lati, ma solo da un lato (ad esempio, quello destro). Per convenzione, assumiamo che, se la testina cerca di oltrepassare il limite sinistro del nastro, allora la macchina termina in modo anomalo (al pari di una transizione non definita).

Una macchina di Turing normale può chiaramente simularne una con nastro semi-infinito. L'unico problema che deve essere considerato è la gestione dell'errore nel caso di uno spostamento a sinistra del limite sinistro del nastro. Ciò può essere fatto inserendo un simbolo speciale subito prima della stringa in input, e non definendo alcuna transizione da un qualunque stato leggendo tale simbolo speciale.

Mostriamo ora come sia vero anche il viceversa, ossia che una macchina di Turing con nastro semi-infinito può simularne una con nastro infinito da entrambi i lati. L'idea della dimostrazione consiste nel mantenere il contenuto alla destra di una prefissata cella 0 del nastro infinito (inclusa tale cella) nelle celle "dispari" del nastro semi-infinito e quello alla sinistra nelle celle "pari" (assumendo che la prima cella del nastro semi-infinito sia pari): la prima cella del nastro semi-infinito conterrà un simbolo speciale (ad esempio, il simbolo $\top$) che consenta di non oltrepassare mai il limite sinistro del nastro (si veda la Figura 2.9 in cui la cella 0 del nastro infinito è quella contenente il simbolo sottolineato).

L'unico problema che la simulazione della macchina $\mathtt{T}$ con nastro infinito da parte della macchina $\mathtt{T}'$ con nastro semi-infinito dovrà gestire, riguarda il movimento della testina: in particolare, ciò viene fatto nel modo seguente.

- Trovandosi in celle dispari (rispettivamente, pari), ogni spostamento a destra (rispettivamente, a sinistra) di $\mathtt{T}$ viene simulato da $\mathtt{T}'$ con due spostamenti a

destra.

- Trovandosi in celle dispari, ogni spostamento a sinistra di T viene simulato da T' con un primo spostamento a sinistra: se il simbolo letto non è quello speciale, viene effettuato il secondo spostamento a sinistra, altrimenti vengono effettuati due spostamenti a destra.
- Trovandosi in celle pari, ogni spostamento a destra di T viene simulato da T' con due spostamenti a sinistra: se il simbolo letto è quello speciale, viene effettuato un ulteriore spostamento a destra.

Osserviamo che, per realizzare tale simulazione, dobbiamo essere in grado di distinguere il caso in cui la testina si trovi su una cella dispari da quello in cui si trovi su una cella pari. Ciò può essere fatto raddoppiando gli stati, ovvero creando per ogni stato q uno stato $q_{\texttt{odd}}$ e uno stato $q_{\texttt{even}}$, e gestendo opportunamente le transizioni tra gli stati nel momento in cui si passa da un lato all'altro del nastro di T. Poiché la simulazione di un passo di T richiede un numero costante di passi eseguiti da T', abbiamo che $t_{T'}(n) = O(t_T(n))$.

Esercizi

Esercizio 2.1. Dato l'alfabeto $\Sigma = \{0, 1, B, U, Z, L, R, S\}$, si definisca il codice binario c_Σ a esso corrispondente.
Esercizio 2.2. Dato $k \geqslant 3$ e dato un alfabeto Σ di n simboli, in modo simile a quanto fatto nel primo paragrafo di questo capitolo si definisca il codice c_Σ^k che utilizza k simboli e, per ogni stringa x su Σ di lunghezza m, si confronti la lunghezza di $c_\Sigma(x)$ con quella di $c_\Sigma^k(x)$.
Esercizio 2.3. Si definisca la macchina di Turing con alfabeto di lavoro $\Sigma = \{\square, 0, 1\}$ corrispondente alla macchina di Turing per la conversione da unario a binario descritta nel secondo paragrafo del primo capitolo.
Esercizio 2.4. Data una macchina di Turing T definita mediante il suo grafo delle transizioni, si calcoli il numero di diverse possibili codifiche di T.
Esercizio 2.5. Si scriva la codifica delle macchine di Turing viste nel secondo paragrafo del primo capitolo.
Esercizio 2.6. Facendo uso dell'applicativo JFLAP e facendo riferimento all'esercizio precedente, si verifichi il comportamento della macchina di Turing universale con input la codifica delle macchine di Turing viste nel secondo paragrafo del primo capitolo.
Esercizio 2.7. Una macchina di Turing con nastro tridimensionale è simile a una macchina di Turing ordinaria a eccezione del fatto che il nastro è tridimensionale e che ogni cella del nastro è un cubo. I possibili movimenti della testina, a ogni singolo passo, sono verso sinistra, verso destra, verso l'alto, verso il basso, in avanti oppure indietro. Dimostrare che il modello

delle macchine di Turing tridimensionali è equivalente a quello delle macchine di Turing ordinarie.

Esercizio 2.8. Una macchina di Turing multi-testina è simile a una macchina di Turing ordinaria a eccezione del fatto che il nastro è scandito da un numero limitato di testine, che si muovono indipendentemente l'una dall'altra. A ogni transizione è associata un'etichetta formata da una lista di triple

$$((\sigma_1,\ldots,\sigma_k),(\tau_1,\ldots,\tau_k),(m_1,\ldots,m_k))$$

I simboli $\sigma_1,\ldots,\sigma_k$ e $\tau_1,\ldots,\tau_k$ che appaiono all'interno di una tripla dell'etichetta di una transizione indicano, rispettivamente, i k simboli attualmente letti dalle k testine e i k simboli da scrivere (assumendo che se due testine devono scrivere due simboli diversi nella stessa cella, la priorità viene assegnata alla testina con indice maggiore), mentre i valori $m_1,\ldots,m_k$ specificano i movimenti che devono essere effettuati dalle k testine. Dimostrare che il modello delle macchine di Turing con k testine, dove k è un qualunque numero intero con $k \geqslant 2$, è equivalente a quello delle macchine di Turing ordinarie.

Esercizio 2.9. Dimostrare che il modello delle macchine di Turing la cui testina può operare solo movimenti nell'insieme $\{\mathtt{L}\}$ oppure nell'insieme $\{\mathtt{S},\mathtt{L}\}$ è meno potente del modello ordinario.

Esercizio 2.10. Una macchina di Turing con due stack è una macchina di Turing dotata di tre nastri semi-infiniti, di cui il primo è di sola lettura e gli altri due, detti *stack*, sono tali che ogni qualvolta la testina si sposta a sinistra, il simbolo scritto è necessariamente il simbolo $\square$. Dimostrare che il modello delle macchine di Turing con due stack è equivalente a quello delle macchine di Turing ordinarie.

Capitolo 3

Limiti delle macchine di Turing

SOMMARIO

In questo capitolo mostriamo l'esistenza di funzioni che non possono essere calcolate da alcuna macchina di Turing. In particolare, dopo aver introdotto il concetto di funzione calcolabile e di linguaggio decidibile, mostriamo anzitutto (in modo non costruttivo), mediante la tecnica di diagonalizzazione, che devono esistere linguaggi non decidibili. Quindi, diamo alcuni esempi naturali di tali linguaggi (come, ad esempio, il linguaggio della terminazione), facendo anche uso della tecnica di riduzione. Successivamente, dimostriamo il teorema della ricorsione che, informalmente, afferma che esistono macchine di Turing in grado di produrre la loro stessa descrizione. Infine, concludiamo il capitolo applicando il teorema della ricorsione per fornire una dimostrazione alternativa della non decibilità del linguaggio della terminazione e per dimostrare il teorema del punto fisso e quello di Rice, il quale afferma fondamentalmente che nessuna proprietà non banale delle macchine di Turing può essere decisa in modo automatico.

3.1 Funzioni calcolabili e linguaggi decidibili

ABBIAMO VISTO nel primo capitolo come le configurazioni siano uno strumento utile per rappresentare in modo sintetico l'esecuzione di una macchina di Turing con input una stringa specificata e per poter definire il concetto di sotto-macchina. Il motivo principale per cui le abbiamo introdotte, tuttavia, è quello di poter definire formalmente il concetto di funzione calcolabile da una macchina di Turing e quello di linguaggio decidibile. A tale scopo, ricordiamo anzitutto che la configurazione iniziale di una macchina di Turing con input una stringa x sull'alfabeto Σ non contenente $\square$, è definita come ^{q_0}x dove q_0 denota lo stato iniziale della macchina. Tale definizione corrisponde a quanto detto nel primo capitolo, in cui abbiamo assunto che, all'inizio della sua computazione, una macchina di Turing abbia la testina

Relazioni e funzioni

Dato un insieme X, una **relazione** r su X è un sottoinsieme di $X \times X$. Se, per ogni $x \in X$, $(x,x) \in r$, allora la relazione è detta essere **riflessiva**. Se, per ogni x e y in X, $(x,y) \in r$ se e solo se $(y,x) \in r$, allora la relazione è detta essere **simmetrica**. Infine, se, per ogni x, y e z in X, $(x,y) \in r$ e $(y,z) \in r$ implicano che $(x,z) \in r$, allora la relazione è detta essere **transitiva**. Dati due insiemi X e Y, una **funzione** $f : X \to Y$ è un sottoinsieme di $X \times Y$ tale che, per ogni $x \in X$, esiste al più un $y \in Y$ per cui $(x,y) \in f$. Se $(x,y) \in f$, allora scriveremo $f(x) = y$ e diremo che f è *definita* per x e che y è l'**immagine** di x. Se una funzione f è definita per ogni $x \in X$, allora f è detta essere **totale**. Se, per ogni $y \in Y$, esiste un $x \in X$ per cui $f(x) = y$, allora la funzione è detta essere **suriettiva**. Se, per ogni x_1 e x_2 in X per cui f è definita, si ha che $x_1 \neq x_2$ implica $f(x_1) \neq f(x_2)$, allora la funzione è detta essere **iniettiva**. Una funzione totale, iniettiva e suriettiva è detta essere **biettiva**.

posizionata sul primo simbolo della stringa di input e si trovi nel suo (unico) stato iniziale.

Definizione 3.1: funzione calcolabile

Dati due alfabeti Σ e Γ tali che $\square \notin \Sigma \cup \Gamma$ e data una funzione $f : \Sigma^* \to \Gamma^*$, f è **calcolabile** se esiste una macchina di Turing T tale che, per ogni $x \in \Sigma^*$, esiste una sequenza $c_1, \ldots, c_n$ di configurazioni di T per cui valgono le seguenti affermazioni.

1. c_1 è la configurazione iniziale di T con input x.
2. Per ogni i con $1 \leqslant i < n$, c_i produce c_{i+1}.
3. c_n è una configurazione finale $y\,q\,f(x)\square z$, dove q è uno stato finale di T e y e z sono due stringhe (eventualmente vuote) sull'alfabeto di lavoro di T.

Abbiamo già visto nel primo capitolo diversi esempi di funzioni calcolabili, ovvero la funzione che complementa ogni bit di una stringa binaria o quella che ne calcola il complemento a due, la funzione di somma di due numeri interi rappresentati in binario o quella che ordina una stringa binaria.

Osserviamo che l'esecuzione di una macchina di Turing con input una stringa x può portare a tre possibili risultati. Il primo caso si ha quando la macchina termina in una configurazione finale $y\,q\,w\square z$, con $w \in \Gamma^*$: in tal caso, la macchina ha calcolato il valore w a partire dal valore x. Il secondo caso si ha quando la macchina non termina, ovvero la CPU non entra mai in uno stato finale. Il terzo e ultimo caso si ha quando la macchina termina in una configurazione non finale $y\,q\,w\square z$ in quanto non esistono transizioni da applicare: in questo caso, pur terminando l'esecuzione, la macchina non ha calcolato alcun valore. Osserviamo che tra questi due ultimi casi, il secondo è preferibile al primo, in quanto, come vedremo più avanti, decidere se

una macchina terminerà la sua esecuzione risulta essere un compito particolarmente difficile: per questo motivo, preferiremo sempre avere a che fare con macchine di Turing che terminano la propria esecuzione per qualunque stringa di input (anche senza calcolare alcun valore).

3.1.1 Linguaggi decidibili

Avendo definito il concetto di funzione calcolabile, siamo ora in grado di introdurre quello di linguaggio decidibile. A tale scopo, associamo a ogni linguaggio una funzione, la cui calcolabilità determina la decidibilità del linguaggio corrispondente. Dato un linguaggio L su un alfabeto Σ, la **funzione caratteristica** $\chi_L : \Sigma^* \to \{0,1\}$ di L è definita nel modo seguente. Per ogni $x \in \Sigma^*$,

$$\chi_L(x) = \begin{cases} 1 & \text{se } x \in L, \\ 0 & \text{altrimenti} \end{cases}$$

Definizione 3.2: linguaggio decidibile

Un linguaggio L è **decidibile** se la sua funzione caratteristica è calcolabile.

Un linguaggio L su un alfabeto Σ è, quindi, decidibile se esiste una macchina di Turing T che *decide* L, ovvero per cui valgono le seguenti due affermazioni.

- Per ogni $x \in L$, T con input x termina nella configurazione finale $y\,q\,1\,\square w$, dove q è uno stato finale di T e y e w sono due stringhe (eventualmente vuote) sull'alfabeto di lavoro di T.
- Per ogni $x \notin L$, T con input x termina nella configurazione finale $y\,q\,0\,\square w$, dove q è uno stato finale di T e y e w sono due stringhe (eventualmente vuote) sull'alfabeto di lavoro di T.

Consideriamo, ad esempio, il linguaggio $L_{\text{parity}} = \{1^k : k \text{ è pari}\}$ e dimostriamo che tale linguaggio è decidibile, definendo una macchina di Turing che ne calcoli la funzione caratteristica. Tale macchina scorre la stringa in input verso destra, sostituendo tutti i simboli `1` con il simbolo $\square$ e alternandosi tra due stati fino a raggiungere il primo $\square$: il primo stato memorizza il fatto che il numero di simboli `1` letti sinora è pari, mentre il secondo stato memorizza il fatto che tale numero è dispari. Se la macchina si trova nel primo stato al termine della lettura dell'input, allora scrive un simbolo `1` e si ferma, altrimenti scrive un simbolo `0` e si ferma.

JFLAP: macchina di Turing per il linguaggio L_{parity}

Osserva, sperimenta e verifica la macchina
`parity.jff`

Teorema 3.1

Se L è un linguaggio decidibile, allora anche L^c è decidibile.

Dimostrazione. Una macchina di Turing T^c che decide L^c può essere facilmente ottenuta a partire da una macchina di Turing T che decide L, sostituendo il simbolo `0` con il simbolo `1` e viceversa al momento in cui T entra in uno stato finale. ◇

Teorema 3.2

Un linguaggio L su un alfabeto Σ è decidibile se e solo se esiste una macchina di Turing T con un solo stato finale tale che, per ogni $x \in \Sigma^*$, T con input x termina e termina in una configurazione finale se e solo se $x \in L$.

Dimostrazione. Supponiamo che L sia decidibile e, anzitutto, osserviamo che la sua funzione caratteristica è una funzione totale $\chi_L : \Sigma^* \to \{0,1\}$. Sia S la macchina di Turing che calcola χ_L: modifichiamo S per ottenere T nel modo seguente. Aggiungiamo all'insieme degli stati finali uno stato $q_{\texttt{end}}$ e facciamo diventare non finali tutti gli stati finali di S. Per ogni stato finale q di S introduciamo una transizione a partire da questo stato che collega q a $q_{\texttt{end}}$ e la cui etichetta contiene la tripla $(\texttt{1}, \square, \texttt{S})$. Poiché χ_L è totale, per ogni $x \in \Sigma^*$, esiste uno stato finale q di S per cui S con input x termina nella configurazione $y\,q\,\texttt{1}\square w$ (se $x \in L$), oppure nella configurazione $y\,q\,\texttt{0}\square w$ (se $x \notin L$), dove y e w sono due stringhe (eventualmente vuote) sull'alfabeto di lavoro di S. Nel primo caso, T termina nello stato $q_{\texttt{end}}$, mentre nel secondo caso termina nello stato q, che non è finale per T, in quanto non sono definite transizioni a partire da q che leggano il simbolo `0`. Pertanto, T con input x termina sicuramente e termina in una configurazione finale se e solo se $x \in L$. Viceversa, supponiamo che esista una macchina di Turing T che soddisfa le due condizioni del teorema: modifichiamo T, in modo da ottenere una macchina S che decide L, nel modo seguente. Aggiungiamo agli stati finali di T uno stato $q_{\texttt{end}}$, facciamo diventare non finale l'unico stato finale di T e aggiungiamo un nuovo stato (non finale) $q_{\texttt{0/1}}$. Introduciamo, quindi, una transizione dallo stato finale di T a $q_{\texttt{0/1}}$ che, qualunque sia il simbolo letto, scrive `1` e si sposta a destra. Introduciamo, inoltre, una transizione da $q_{\texttt{0/1}}$ a $q_{\texttt{end}}$ che, qualunque sia il simbolo letto, scriva $\square$ e si sposti a sinistra. Infine, per ogni stato non finale q di T e per ogni simbolo σ per cui non siano definite transizioni a partire da q che leggano il simbolo σ, definiamo una transizione da q a $q_{\texttt{0/1}}$ che, qualunque sia il simbolo

Figura 3.1: due diverse macchine di Turing per il linguaggio L_{parity}.

letto, scrive 0 e si sposta a destra. Evidentemente, se T termina in una configurazione finale, allora S termina nella configurazione finale $y\,q_{end}\,1\square w$, mentre se T termina in uno stato non finale q, allora S termina nella configurazione finale $y\,q_{end}\,0\square w$ (dove y e w sono due stringhe, eventualmente vuote, sull'alfabeto di lavoro di T): poiché T termina in una configurazione finale se e solo se $x \in L$, allora S calcola la funzione caratteristica di L. ◇

Applicando la dimostrazione del teorema precedente alla macchina di Turing per il linguaggio parità precedentemente descritta, otteniamo la macchina mostrata nella parte sinistra della Figura 3.1. Ovviamente, potremmo definire in modo più diretto una macchina di Turing equivalente, facendo passare lo stato q0 nello stato finale nel momento in cui il simbolo letto sia □ e non definendo alcuna transizione a partire da q1 che legga il simbolo □: tale macchina è mostrata nella parte destra della figura.

3.2 Il metodo della diagonalizzazione

PER MOSTRARE l'esistenza di un linguaggio non decidibile faremo uso di una tecnica introdotta da Cantor alla fine del XIX secolo. Quest'ultimo era interessato a capire come fosse possibile confrontare la cardinalità di due insiemi infiniti. In effetti, mentre è immediato decidere se un insieme finito è più o meno grande di un altro insieme finito, non è altrettanto evidente decidere quale tra due insiemi infiniti risulti essere il più grande. Ad esempio, se consideriamo l'insieme N dei numeri interi e quello Σ^* di tutte le stringhe su un alfabeto Σ, entrambi questi insiemi sono infiniti (e quindi più grandi di un qualunque insieme finito): tuttavia, come possiamo dire se N è più grande di Σ^* o viceversa? La soluzione proposta da Cantor è basata sulla possibilità di mettere in corrispondenza biunivoca gli elementi di un insieme con quelli di un altro insieme.

Definizione 3.3: insiemi equi-cardinali

Due insiemi A e B sono **equi-cardinali** o, equivalentemente, hanno la *stessa cardinalità* se esiste una funzione totale e biettiva $f : A \to B$.

Esempio 3.1: numeri naturali e numeri naturali positivi

Consideriamo l'insieme $\mathbb{N}$ dei numeri naturali e l'insieme $\mathbb{N}^+$ dei numeri naturali positivi. Sia $\mathbb{N}$ che $\mathbb{N}^+$ sono insiemi infiniti. Inoltre, è ovvio che $\mathbb{N}^+$ è un sotto-insieme di $\mathbb{N}$: questi due insiemi hanno comunque la stessa cardinalità. Possiamo, infatti, definire la funzione totale e biettiva $f : \mathbb{N}^+ \to \mathbb{N}$ nel modo seguente: per ogni numero naturale $n > 0$, $f(n) = n - 1$.

Esempio 3.2: numeri naturali e numeri pari

Consideriamo l'insieme $\mathbb{N}$ dei numeri naturali e l'insieme $\mathbb{P}$ dei numeri naturali pari. Sia $\mathbb{N}$ che $\mathbb{P}$ sono insiemi infiniti. Inoltre, è ovvio che $\mathbb{P}$ è un sotto-insieme di $\mathbb{N}$: questi due insiemi hanno comunque la stessa cardinalità. Possiamo, infatti, definire la funzione totale e biettiva $f : \mathbb{P} \to \mathbb{N}$ nel modo seguente: per ogni numero naturale pari n, $f(n) = \frac{n}{2}$.

Definizione 3.4: insiemi numerabili

Un insieme A è detto essere **numerabile** se ha la stessa cardinalità dell'insieme dei numeri naturali $\mathbb{N}$.

Il termine "numerabile" deriva dal fatto che, se un insieme A è numerabile, allora è possibile elencare uno dopo l'altro senza ripetizioni gli elementi di A, ovvero associare a ciascuno degli elementi di A una posizione univoca all'interno di tale elenco. In base a quanto esposto nei due esempi precedenti, possiamo concludere che sia l'insieme dei numeri naturali positivi che quello dei numeri pari sono insiemi numerabili: in entrambi i casi la numerazione dei loro elementi è quella più naturale. I prossimi due esempi mostrano, invece, altri due insiemi numerabili di numeri, la cui numerazione non è altrettanto naturale.

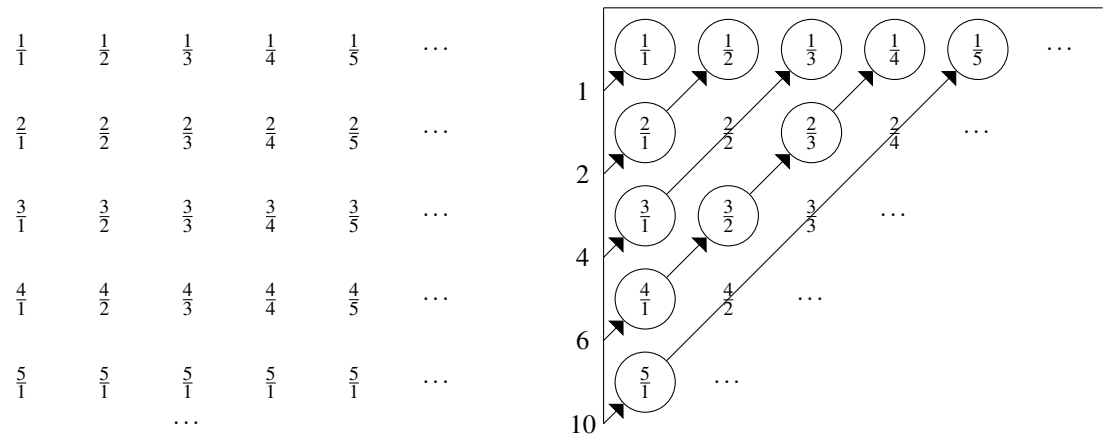
Esempio 3.3: numeri naturali e numeri interi

Consideriamo l'insieme $\mathbb{Z}$ dei numeri interi. È ovvio che $\mathbb{N}$ è un sotto-insieme di $\mathbb{Z}$. Possiamo ora mostrare che $\mathbb{Z}$ è numerabile, ovvero che ha la stessa cardinalità di $\mathbb{N}$. La funzione totale e biettiva $f : \mathbb{Z} \to \mathbb{N}$ è definita nel modo seguente: per ogni numero intero n,

$$f(n) = \begin{cases} 0 & \text{se } n = 0, \\ 2n - 1 & \text{se } n > 0, \\ -2n & \text{altrimenti} \end{cases}$$

In altre parole, associamo il numero 0 (inteso come numero naturale) allo 0 (inteso come numero intero), i numeri naturali dispari ai numeri interi positivi e i numeri naturali pari ai numeri interi negativi, definendo una numerazione dei numeri interi che inizia nel modo seguente: $0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5$.

Figura 3.2: numerabilità dei numeri razionali positivi



Esempio 3.4: numeri naturali e numeri razionali

Consideriamo l'insieme N^+ dei numeri naturali positivi e l'insieme Q^+ dei numeri razionali positivi. Sia N^+ che Q^+ sono insiemi infiniti. Inoltre, è ovvio che N^+ è un sotto-insieme di Q^+: questi due insiemi hanno comunque la stessa cardinalità (quindi, Q^+ è un insieme numerabile). Per definire la funzione biettiva $f : Q^+ \to N^+$, costruiamo una matrice infinita contenente tutti i numeri razionali positivi con eventuali ripetizioni, come mostrato nella parte sinistra della Figura 3.2: l'elemento in i-esima riga e j-esima colonna è il numero razionale $\frac{i}{j}$. Per ottenere una numerazione degli elementi della matrice, non possiamo ovviamente pensare di numerare prima tutti gli elementi della prima riga, poi quelli della seconda riga e così via, poiché in tal modo non termineremmo mai di numerare gli elementi della prima riga che è infinita. Per ovviare a tale inconveniente, possiamo adottare una strategia simile alla visita in ampiezza di un grafo, procedendo per diagonali di dimensione crescente (si veda la parte destra della figura). La prima di tali diagonali è formata dal solo elemento $\frac{1}{1}$, a cui viene quindi associato il numero naturale positivo 1. La seconda diagonale contiene gli elementi $\frac{2}{1}$ e $\frac{1}{2}$, ai quali vengono quindi associati i numeri naturali positivi 2 e 3. Nel caso della terza diagonale, sorge il problema dovuto al fatto che tale diagonale contiene l'elemento $\frac{2}{2} = \frac{1}{1}$: se associassimo un numero naturale positivo a ciascun elemento della diagonale, avremmo che allo stesso numero razionale positivo sarebbero associati più numeri naturali positivi diversi. Quindi, l'elemento $\frac{2}{2}$ viene saltato nella numerazione e ai due restanti elementi della terza diagonale (ovvero, $\frac{3}{1}$ e $\frac{1}{3}$) vengono associati i numeri naturali positivi 4 e 5. Continuando in questo modo, otteniamo una numerazione di tutti gli elementi di Q^+ e, quindi, la funzione totale e biettiva f.

Dimostrazioni per assurdo

Un modo di dimostrare un enunciato del tipo "`se A è vero, allora B è vero`", consiste nel ragionare per assurdo, assumendo che `B` sia falso e mostrando come la congiunzione di `A` e della negazione di `B` porti a una conseguenza logica notoriamente falsa. Una ben nota dimostrazione per assurdo è quella del seguente enunciato: "se il lato di una quadrato ha lunghezza unitaria, allora la lunghezza d della sua diagonale non è un numero razionale" (equivalentemente, tale enunciato afferma che il numero $\sqrt{2}$ non è un numero razionale). Assumendo, infatti, che d sia un numero razionale (ovvero, *assumendo che `B` sia falso*), abbiamo che $d = \frac{n}{m}$ con n e m relativamente primi (ovvero, il massimo comun divisore di n e m è uguale a 1). Inoltre, avendo il lato del quadrato lunghezza unitaria (*essendo `A` vero*), dal teorema di Pitagora segue che $d^2 = 2$: quindi, $n^2 = 2m^2$, ovvero n^2 è un numero pari. Poichè solo il quadrato di un numero pari può essere pari, abbiamo che n è pari e, quindi, che n^2 è un multiplo di 4. Inoltre, poiché n e m sono relativamente primi, abbiamo che m e, quindi, m^2 sono dispari: ciò implica l'esistenza di un numero dispari il cui doppio è un multiplo di 4 e quest'affermazione è notoriamente falsa.

Una domanda naturale che possiamo porci, a questo punto, è se esistano insiemi infiniti di numeri che non siano numerabili, ovvero che non possano essere elencati in modo univoco senza ripetizioni. Il primo candidato a godere di tale proprietà che può venire in mente, è l'insieme R dei numeri reali: intuitivamente, tale insieme sembra essere molto più grande dell'insieme dei numeri interi e di quello dei numeri razionali, in quanto contiene numeri che non possono essere rappresentati con un numero finito di cifre. Il prossimo teorema conferma che tale intuizione è in effetti giusta: la dimostrazione del teorema ci consentirà, tra l'altro, di introdurre il **metodo della diagonalizzazione**.

Teorema 3.3

L'insieme R dei numeri reali non è numerabile.

Dimostrazione. La dimostrazione procede per assurdo, assumendo che R sia numerabile, ovvero che esista una funzione totale e biettiva $f : R \to N^+$. Possiamo quindi numerare tutti i numeri reali (senza ripetizioni) e costruire una matrice M di cifre decimali tale che la cifra alla riga i e alla colonna j sia la j-esima cifra della parte decimale dell'i-esimo numero reale (ovvero del numero reale x tale $f(x) = i$). Ad esempio, supponiamo che la tabella mostrata nella parte sinistra della Figura 3.3 sia l'inizio di un'ipotetica numerazione dei numeri reali: allora, la parte iniziale della matrice M sarebbe quella mostrata nella parte destra della figura. Definiamo ora un numero reale $x_{\text{diag}} < 1$ nel modo seguente: per ogni numero naturale positivo i, l'i-esima cifra della parte decimale di x_{diag} è uguale a $[(M[i,i]+1) \bmod 10]$. In altre parole, l'i-esima cifra della parte decimale di x_{diag} è posta uguale all'i-esima cifra del numero reale corrispondente a i incrementata di 1 (modulo 10). Ad esempio, il numero x_{diag} corrispondente all'ipotetica numerazione mostrata in precedenza inizierà

Figura 3.3: metodo della diagonalizzazione

i	x
1	7.4875690...
2	33.3333333 ...
3	0.8362719 ...
4	65.4971652 ...
5	0.3384615 ...
6	0.3442623 ...
⋮	⋮

4	8	7	5	6	9	0	...
3	**3**	3	3	3	3	3	...
8	3	**6**	2	7	1	9	...
4	9	7	**1**	6	5	2	...
3	3	8	4	**6**	1	5	...
3	4	4	2	6	**2**	3	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

nel modo seguente: $0.547273\ldots$. Chiaramente, per ogni numero naturale positivo $\mathtt{i}$, $\mathtt{i}$ non è l'immagine di x_{diag} in quanto x_{diag} differisce dal numero reale la cui immagine è $\mathtt{i}$ per almeno una cifra decimale (ovvero, l'$\mathtt{i}$-esima):[1] quindi, a x_{diag} non corrisponde alcun numero naturale positivo, contraddicendo l'ipotesi (fatta per assurdo) che $\mathtt{f}$ sia una funzione totale e biettiva da $\mathtt{R}$ in $\mathtt{N}^+$. Pertanto, abbiamo dimostrato che non può esistere una tale funzione e che $\mathtt{R}$ non è numerabile. ◇

Dalla dimostrazione del teorema precedente, dovrebbe risultare chiaro il motivo per cui la tecnica utilizzata sia comunemente chiamata "diagonalizzazione". In generale, se A e B sono due insiemi numerabili e se $\mathtt{f}_{A,B} : \mathtt{N} \times \mathtt{N} \to X$, con $|X| \geqslant 2$, è una funzione totale che, per ogni coppia $\mathtt{i}$ e $\mathtt{j}$ di numeri naturali, specifica una determinata relazione tra l'$(\mathtt{i}+1)$-esimo elemento di A e il $(\mathtt{j}+1)$-esimo elemento di B, tale tecnica consiste nel definire un *oggetto diagonale* $\mathtt{d}$ il quale si "rapporta" con almeno un elemento di B in modo diverso da ciascun altro elemento di A: in particolare, per ogni numero naturale $\mathtt{i}$, $\mathtt{d}$ è quell'oggetto la cui relazione con l'$(\mathtt{i}+1)$-esimo elemento di B è nell'insieme $X - \{\mathtt{f}_{A,B}(\mathtt{i},\mathtt{i})\}$. Osserviamo che se siamo in grado di dimostrare che l'oggetto diagonale appartiene ad A, allora abbiamo ottenuto una contraddizione. Infatti, in quanto membro di A, $\mathtt{d}$ corrisponde in modo univoco a un numero naturale $\mathtt{i}_\mathtt{d}$ (poiché A è numerabile): tuttavia, la relazione tra $\mathtt{d}$ e l'$(\mathtt{i}_\mathtt{d}+1)$-esimo elemento di B è per definizione diverso da $\mathtt{f}_{A,B}(\mathtt{i}_\mathtt{d},\mathtt{i}_\mathtt{d})$ (generalmente, tale contraddizione è dovuta all'assunzione che l'insieme A sia numerabile). Nel caso della dimostrazione del teorema precedente, A è l'insieme dei numeri reali, B è l'insieme dei numeri naturali, $X = \{0,1,\ldots,9\}$ e, assumendo che A sia numerabile, $\mathtt{f}_{A,B}(\mathtt{i},\mathtt{j})$ determina la $(\mathtt{j}+1)$-esima cifra decimale dell'$(\mathtt{i}+1)$-esimo numero reale, per ogni coppia di

[1] In realtà, è possibile che due numeri reali siano uguali pur avendo una diversa rappresentazione decimale (si pensi, ad esempio, a $0.49999999\ldots$ e a $0.5000000\ldots$): possiamo comunque risolvere questo problema evitando di scegliere le cifre 0 e 9 quando costruiamo x_{diag}.

numeri naturali $\mathtt{i}$ e $\mathtt{j}$: l'oggetto diagonale x_{diag} è un numero reale che si distingue per almeno una cifra decimale da ogni elemento di A (da cui la contraddizione e l'impossibilità di numerare l'insieme dei numeri reali).

3.2.1 Linguaggi non decidibili

Utilizzando il metodo della diagonalizzazione possiamo mostrare l'esistenza di linguaggi non decidibili. A tale scopo, mostreremo anzitutto che l'insieme di tutte le macchine di Turing è numerabile: poiché una macchina di Turing può decidere al più un linguaggio, ciò implica che l'insieme di tutti i linguaggi decidibili non è più che numerabile. D'altra parte, mostreremo come esista un insieme numerabile di linguaggi decidibili: quindi, l'insieme di tutti i linguaggi decidibili è numerabile. Infine, dimostreremo che l'insieme di tutti i linguaggi sull'alfabeto binario non è numerabile: ciò implica che deve esistere un linguaggio che non è decidibile. Nel seguito di questo capitolo, ci limiteremo a considerare l'insieme $\mathcal{L}$ di tutti i linguaggi sull'alfabeto binario e l'insieme $\mathcal{T}$ di tutte le macchine di Turing il cui alfabeto di lavoro è $\{\square, \mathtt{0}, \mathtt{1}\}$ (in base a quanto detto nel capitolo precedente, questa restrizione non causa alcuna perdita di generalità).

Lemma 3.1

L'insieme di tutti i linguaggi decidibili appartenenti a $\mathcal{L}$ è numerabile.

Dimostrazione. Sia $\mathcal{D}$ l'insieme di tutti i linguaggi decidibili appartenenti a $\mathcal{L}$. La dimostrazione consiste nel far vedere che $\mathcal{T}$ è numerabile (il che implica che $\mathcal{D}$ non è più che numerabile) e che esiste un insieme numerabile di linguaggi decidibili contenuto in $\mathcal{L}$ (il che implica che $\mathcal{D}$ è almeno numerabile). Per quanto riguarda la prima affermazione, abbiamo già visto nel precedente capitolo come sia possibile associare in modo univoco, a ogni macchina di Turing in $\mathcal{T}$, una stringa sull'alfabeto $\{\mathtt{0},\mathtt{1},\mathtt{B},\mathtt{U},\mathtt{Z},\mathtt{L},\mathtt{R},\mathtt{S}\}$. D'altra parte, per ogni alfabeto, l'insieme di tutte le stringhe su di esso è numerabile: è sufficiente, infatti, elencare le stringhe in base all'ordinamento lessicografico e associare il numero $\mathtt{i}$ all'$(\mathtt{i}+1)$-esima stringa in tale elenco. Poiché un sottoinsieme di un insieme numerabile è anch'esso numerabile, abbiamo che l'insieme di tutte le codifiche di una macchina di Turing in $\mathcal{T}$ è numerabile: pertanto, $\mathcal{T}$ è numerabile e, quindi, $\mathcal{D}$ non è più che numerabile. Rimane ora da dimostrare che tale insieme è almeno numerabile, mostrando l'esistenza di un suo sottoinsieme numerabile. Tale sottoinsieme $\mathcal{B}$ è costituito dai seguenti linguaggi contenuti in $\mathcal{L}$: per ogni $\mathtt{i} \geqslant 0$, $L_{\mathtt{i}}$ contiene solo la rappresentazione binaria di $\mathtt{i}$. È facile definire, per ogni $\mathtt{i} \geqslant 0$, una macchina di Turing in $\mathcal{T}$ che decide $L_{\mathtt{i}}$: inoltre, $\mathcal{B}$ è chiaramente numerabile. Quindi, l'insieme $\mathcal{D}$ include un sottoinsieme numerabile e, pertanto, è anch'esso almeno numerabile: il lemma risulta dunque essere dimostrato. ◇

Lemma 3.2

$\mathcal{L}$ non è numerabile.

Dimostrazione. Supponiamo, per assurdo, che $\mathcal{L}$ sia numerabile, ovvero che esista un funzione biettiva $f : \mathcal{L} \to \mathbb{N}$: per ogni numero naturale i, indichiamo con L_i il linguaggio la cui immagine è i. Consideriamo poi la numerazione $\sigma_0, \sigma_1, \sigma_2, \ldots$ di tutte le stringhe su Σ ottenuta in base all'ordinamento lessicografico. Ragionando per diagonalizzazione, definiamo un linguaggio L_{diag} nel modo seguente: per ogni numero naturale i, $\sigma_i \in L_{diag}$ se e solo se $\sigma_i \notin L_i$. Chiaramente, per ogni numero naturale i, $L_{diag} \neq L_i$ in quanto L_{diag} differisce da L_i per almeno una stringa (ovvero, σ_i): quindi, a L_{diag} non corrisponde alcun numero naturale, contraddicendo l'ipotesi (fatta per assurdo) che f sia una funzione biettiva da $\mathcal{L}$ in $\mathbb{N}$. Pertanto, abbiamo dimostrato che non può esistere una tale funzione e che $\mathcal{L}$ non è numerabile. ◇

Teorema 3.4

Esiste un linguaggio in $\mathcal{L}$ che non è decidibile.

Dimostrazione. La dimostrazione segue immediatamente dai due lemmi precedenti. ◇

3.3 Il problema della terminazione

LA DIMOSTRAZIONE del Teorema 3.4, pur testimoniando il fatto che esistano linguaggi non decidibili, risulta essere alquanto artificiale e, in qualche modo, di poco interesse pratico. In questo paragrafo mostreremo, sempre mediante il metodo della diagonalizzazione, che esistono linguaggi "interessanti" che non sono decidibili. In particolare, considereremo il problema di decidere se un dato programma con input un dato valore termina la sua esecuzione. Non è necessario, crediamo, evidenziare l'importanza di un tale problema: se fossimo in grado di risolverlo in modo automatico, potremmo, ad esempio, incorporare tale soluzione all'interno dei compilatori e impedire, quindi, la distribuzione di programmi che non abbiano termine. Sfortunatamente, il risultato principale di questo paragrafo afferma che nessun programma è in grado di determinare la terminazione di tutti gli altri programmi.

Nel seguito di questo capitolo, indicheremo con $\mathcal{C}$ l'insieme delle codifiche binarie c_T (secondo quanto visto nel primo paragrafo del capitolo precedente) delle codifiche di una macchina di Turing $T \in \mathcal{T}$ (secondo quanto visto nel secondo paragrafo del capitolo precedente). Inoltre, per ogni stringa binaria x e per ogni $T \in \mathcal{T}$, indicheremo con $T(x)$ la computazione della macchina di Turing T con input x: tale computazione può terminare in una configurazione finale, può terminare in una configurazione non finale oppure può non terminare. Infine, date k stringhe binarie $x_1, \ldots, x_k$, indicheremo con $\langle x_1, \ldots, x_k \rangle$ la stringa binaria $d(x_1)01d(x_2)01 \cdots 01d(x_k)$ dove, per

ogni i con $1 \leqslant i \leqslant k$, $d(x_i)$ denota la stringa binaria ottenuta a partire da x_i raddoppiando ciascun suo simbolo: ad esempio, $\langle 0110, 1100 \rangle$ denota la stringa binaria 00111100001111100000. Osserviamo che, grazie alla presenza del separatore 01, questo modo di specificare sequenze di stringhe binarie non è ambiguo, nel senso che, data la stringa binaria $\langle x_1, \ldots, x_k \rangle$, è possibile determinare in modo univoco le k stringhe binarie $x_1, \ldots, x_k$.

Il **problema della terminazione** consiste nel decidere il seguente linguaggio.

$$L_{stop} = \{\langle c_T, x \rangle : c_T \in \mathcal{C} \wedge x \in \{0,1\}^* \wedge T(x) \text{ termina}\}$$

Osserviamo che il linguaggio L_{stop} include due tipi di stringhe: le stringhe $\langle c_T, x \rangle$ per cui T con input x termina in una configurazione finale e quelle per cui T con input x termina in una configurazione non finale.

Teorema 3.5

L_{stop} non è decidibile.

Dimostrazione. Supponiamo, per assurdo, che L_{stop} sia decidibile: in base al Teorema 3.2, deve esistere una macchina di Turing $T_{stop} \in \mathcal{T}$ tale che, per ogni stringa binaria y,

$$T_{stop}(y) \text{ termina in una } \begin{cases} \text{configurazione finale} & \text{se } y = \langle c_T, x \rangle, c_T \in \mathcal{C} \text{ e } T(x) \text{ termina,} \\ \text{configurazione non finale} & \text{altrimenti} \end{cases}$$

Definiamo, ora, una nuova macchina di Turing $T_{diag} \in \mathcal{T}$ che usa T_{stop} come sottomacchina nel modo seguente: per ogni stringa binaria z,

$$T_{diag}(z) \begin{cases} \text{termina} & \text{se } z = c_T \in \mathcal{C} \text{ e } T_{stop}(\langle c_T, c_T \rangle) \\ & \text{termina in una configurazione non finale} \\ \text{non termina} & \text{altrimenti} \end{cases}$$

(è ovviamente facile forzare una macchina a non terminare). Consideriamo il comportamento di T_{diag} con input la sua codifica, ovvero $c_{T_{diag}}$. Dalla definizione di T_{stop} e di T_{diag}, abbiamo che $T_{diag}(c_{T_{diag}})$ termina se e solo se $T_{stop}(\langle c_{diag}, c_{diag} \rangle)$ termina in una configurazione non finale se e solo se $T_{diag}(c_{T_{diag}})$ non termina. Abbiamo, pertanto, generato una contraddizione, dimostrando così che non può esistere la macchina di Turing T_{stop} e che, quindi, L_{stop} non è decidibile. ◇

Anche se può non sembrare del tutto evidente, la dimostrazione del teorema precedente fa uso della tecnica di diagonalizzazione così come l'abbiamo descritta nel paragrafo precedente. In questo caso, l'insieme A e l'insieme B coincidono entrambi con $\mathcal{C}$, l'insieme X contiene i due valori `true` e `false` e la funzione $f_{A,B}$ è definita

nel modo seguente: per ogni coppia di numeri naturali i e j, $f_{A,B}(i,j) = \texttt{true}$ se e solo se la macchina di Turing corrispondente all'$(i+1)$-esima codifica in $\mathcal{C}$ con input la $(j+1)$-esima codifica in $\mathcal{C}$ termina. La codifica binaria della macchina di Turing T_{diag} della dimostrazione del teorema precedente corrisponde all'oggetto diagonale, il quale appartenendo a $\mathcal{C}$ genera una contraddizione.

3.3.1 Linguaggi semi-decidibili

Il Teorema 3.5 mostra che esistono linguaggi interessanti da un punto di vista applicativo, che non possono essere decisi da alcuna macchina di Turing. La dimostrazione di tale risultato usa esplicitamente il fatto che una macchina di Turing che decide un linguaggio debba terminare per ogni input. Ci possiamo quindi porre, in modo naturale, la seguente domanda: se rilassiamo il vincolo della terminazione per ogni input, è ancora vero che esistono problemi interessanti che non possono essere risolti da una macchina di Turing? Per rispondere a tale domanda, introduciamo anzitutto il concetto di semi-decidibilità.

Definizione 3.5: linguaggi semi-decidibili

Un linguaggio $L \in \mathcal{L}$ è detto essere **semi-decidibile** se esiste una macchina di Turing $T \in \mathcal{T}$ tale che, per ogni stringa binaria x, se $x \in L$, allora $T(x)$ termina in una configurazione finale, altrimenti $T(x)$ non termina.

Esempio 3.5: problema della terminazione

La semi-decidibilità di L_{stop} deriva facilmente dall'esistenza della macchina di Turing universale U. Infatti, possiamo definire una macchina T_{stop} che, per ogni stringa binaria y, verifica anzitutto se $y = \langle c_T, x \rangle$ con $c_T \in \mathcal{C}$: in caso ciò non sia vero, T_{stop} non termina. Altrimenti, T_{stop} esegue U con input c_T e x: se tale computazione termina, allora T_{stop} termina in una configurazione finale. Pertanto, se $y \in L_{stop}$, allora $T_{stop}(y)$ termina in una configurazione finale, altrimenti non termina: ciò dimostra che L_{stop} è semi-decidibile.

Se un linguaggio $L \in \mathcal{L}$ è decidibile, allora L è semi-decidibile: in effetti, è sufficiente modificare la macchina T che decide L in modo che, ogni qualvolta T termina con la testina posizionata sul simbolo 0, la macchina modificata entri in uno stato non finale in cui rimanga poi indefinitamente. Inoltre, essendo anche L^c decidibile (si veda il Teorema 3.1), abbiamo che L^c è semi-decidibile. Il prossimo risultato afferma che la semi-decidibilità di L e di L^c è in realtà una condizione non solo necessaria ma anche sufficiente per la decidibilità di L.

Lemma 3.3

Un linguaggio $L \in \mathcal{L}$ è decidibile se e solo se L e L^c sono entrambi semi-decidibili.

Dimostrazione. Abbiamo già visto che se L è decidibile, allora L e L^c sono entrambi semi-decidibili. Per dimostrare il viceversa, supponiamo che esistano due macchine di Turing $T, T^c \in \mathcal{T}$ tali che, per ogni stringa binaria x, valga una delle seguenti due affermazioni.

- $x \in L$ e $T(x)$ termina in una configurazione finale.
- $x \notin L$ e $T^c(x)$ termina in una configurazione finale.

In altre parole, per ogni input x, $T(x)$ termina oppure $T^c(x)$ termina. Possiamo allora definire una macchina di Turing T' che esegua entrambe le computazioni in parallelo: in particolare, per ogni $i \geqslant 0$, T' esegue i passi di $T(x)$ e i passi di $T^c(x)$ e, se nessuna delle due computazioni termina, passa al valore successivo di i. La macchina T' si ferma nel momento in cui una delle due computazioni termina (cosa che deve necessariamente accadere). Se la prima computazione a fermarsi è $T(x)$, allora T termina in una configurazione finale, altrimenti (ovvero, se la prima computazione a fermarsi è $T^c(x)$) T' termina in una configurazione non finale. In base al Teorema 3.2, T' è una macchina di Turing che testimonia la decidibilità di L e il teorema risulta essere così dimostrato. ◇

Dal fatto che L_{stop} è semi-decidibile, dal teorema precedente e dal Teorema 3.5, segue immediatamente il prossimo risultato, il quale mostra che esistono linguaggi "interessanti" che non sono semi-decidibili.

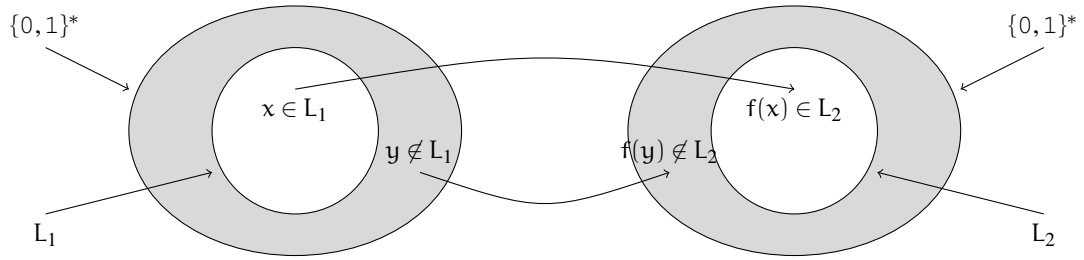
Corollario 3.1

L^c_{stop} non è semi-decidibile.

3.4 Riducibilità tra linguaggi

UNA VOLTA dimostrata l'esistenza di un linguaggio non decidibile, possiamo mostrare che molti altri problemi non sono decidibili facendo uso della tecnica di riduzione. Intuitivamente, tale tecnica consiste nel dimostrare che, dati due linguaggi L_1 e L_2, L_1 non è più difficile di L_2 o, più precisamente, che se esiste una macchina di Turing che decide L_2, allora esiste anche una macchina di Turing che decide L_1. Un lettore abituato allo sviluppo di algoritmi per la risoluzione di problemi conoscerà già la tecnica di riduzione, essendo probabilmente questa una tra le più utilizzate in tale

Figura 3.4: riducibilità tra linguaggi



contesto: in questo contesto, tuttavia, faremo principalmente uso della riducibilità per dimostrare risultati negativi, ovvero per dimostrare che determinati linguaggi non sono decidibili o che non lo sono in modo efficiente (ovvero, come vedremo nella terza parte di queste dispense, che non lo sono in tempo polinomiale). A tale scopo, diamo ora una definizione formale del concetto intuitivo, dato in precedenza, di riducibilità.

Definizione 3.6: linguaggi riducibili

Un linguaggio $L_1 \in \mathcal{L}$ è detto essere **riducibile** a un linguaggio $L_2 \in \mathcal{L}$ se esiste una funzione totale calcolabile $f : \{0,1\}^* \to \{0,1\}^*$, detta **riduzione**, tale che, per ogni stringa binaria x, $x \in L_1$ se e solo se $f(x) \in L_2$.

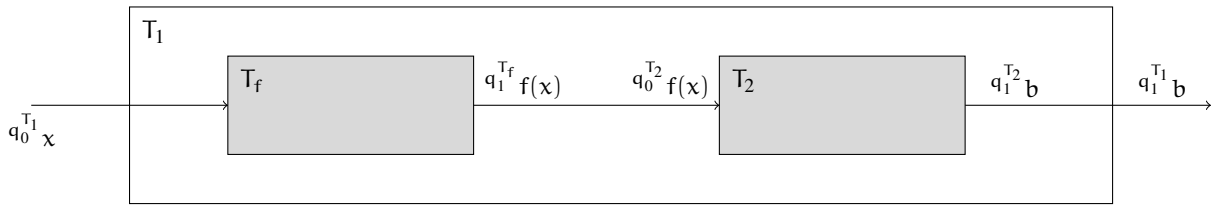
In base alla definizione precedente, una riduzione tra un linguaggio L_1 e un linguaggio L_2 deve essere definita per ogni stringa binaria e deve trasformare stringhe appartenenti a L_1 in stringhe appartenenti a L_2 e stringhe non appartenenti a L_1 in stringhe non appartenenti a L_2 (si veda la Figura 3.4). Osserviamo che la riduzione non deve necessariamente essere suriettiva, ovvero che possono esistere stringhe binarie che non sono immagine di alcuna stringa binaria mediante f. Il risultato seguente formalizza quanto detto in precedenza, ovvero che se un linguaggio L_1 è riducibile a un linguaggio L_2, allora L_1 non è più difficile di L_2 (si veda anche la Figura 3.5).

Lemma 3.4

Siano L_1 e L_2 due linguaggi in $\mathcal{L}$ tali che L_1 è riducibile a L_2. Se L_2 è decidibile, allora L_1 è decidibile.

Dimostrazione. Sia f la funzione totale calcolabile che testimonia il fatto che L_1 è riducibile a L_2 e sia T_f una macchina di Turing che calcola f (senza perdita di generalità, possiamo assumere che, per ogni stringa binaria x, $T_f(x)$ termina con la sola stringa

Figura 3.5: composizione di riduzioni e decisori



$f(x)$ sul nastro e con la testina posizionata sul primo simbolo a sinistra diverso da $\square$). Inoltre, sia T_2 una macchina di Turing che decide L_2. Definiamo allora una macchina di Turing T_1 che decide L_1 nel modo seguente (fondamentalmente, T_1 non è altro che la composizione di T_f con T_2). Per ogni stringa binaria x, T_1 avvia l'esecuzione di T_f con input x: quando T_f termina l'esecuzione lasciando la sola stringa $f(x)$ sul nastro con la testina posizionata sul suo primo simbolo, T_1 avvia l'esecuzione di T_2 con input $f(x)$ (in altre parole, effettua una transizione dallo stato finale di T_f allo stato iniziale di T_2). Per ogni input x, $T_1(x)$ termina con la testina posizionata su un simbolo `1` oppure su un simbolo `0` e termina con la testina posizionata su un simbolo `1` se e solo se $T_2(f(x))$ termina con la testina posizionata su un simbolo `1`. D'altra parte, $T_2(f(x))$ termina con la testina posizionata su un simbolo `1` se e solo se $f(x) \in L_2$ e $f(x) \in L_2$ se e solo se $x \in L_1$. Quindi, per ogni input x, $T_1(x)$ termina con la testina posizionata su un simbolo `1` se solo se $x \in L_1$: ovvero, T_1 decide L_1 e il lemma risulta essere dimostrato. $\diamond$

Come abbiamo già detto, in queste dispense faremo principalmente un uso negativo del teorema precedente allo scopo di dimostrare che determinati linguaggi non sono decidibili: in particolare, utilizzeremo la seguente immediata conseguenza del teorema stesso.

Corollario 3.2

Siano L_1 e L_2 due linguaggi in $\mathcal{L}$ tali che L_1 è riducibile a L_2. Se L_1 non è decidibile, allora L_2 non è decidibile.

La prima applicazione che vedremo del precedente corollario risponde alla domanda che naturalmente ci potremmo porre di sapere se la difficoltà di risoluzione del problema della terminazione risieda nel fatto che non abbiamo posto alcun vincolo sulle stringhe di input: il prossimo esempio mostra come ciò non sia vero (osserviamo che l'esempio può essere facilmente adattato a una qualsiasi stringa binaria diversa da `0`).

Esempio 3.6: problema della terminazione con input fissato

Consideriamo il seguente linguaggio: $L_{stop-0} = \{c_T : c_T \in \mathcal{C} \wedge T(0) \text{ termina}\}$. Dimostriamo ora che L_{stop} è riducibile a L_{stop-0}: dal Teorema 3.5 e dal Corollario 3.2, segue che L_{stop-0} non è decidibile. Data una stringa binaria y, la riduzione f per prima cosa verifica se $y = \langle c_T, x\rangle$ con $c_T \in \mathcal{C}$: in caso contrario (per cui $y \notin L_{stop}$), $f(y)$ produce la codifica di una qualunque macchina di Turing che con input 0 non termina (per cui $f(y) \notin L_{stop-0}$). Altrimenti, $f(\langle c_T, x\rangle)$ produce la codifica $c_{T'}$ di una macchina di Turing T' che, con input una stringa binaria z, per prima cosa cancella z e lo sostituisce con x e, quindi, esegue T con input x. Chiaramente, $\langle c_T, x\rangle \in L_{stop}$ se e solo se $c_{T'} = f(\langle c_T, x\rangle) \in L_{stop-0}$.

Nell'esempio precedente, abbiamo definito il comportamento della riduzione nel caso in cui la stringa y da trasformare non appartenesse a L_{stop} per motivi puramente sintattici (ovvero, $y \neq \langle c_T, x\rangle$ con $c_T \in \mathcal{C}$). In generale, questo aspetto della riduzione non è difficile da gestire nel momento in cui si conosca almeno una stringa binaria w che non appartenga al linguaggio di destinazione: per questo motivo, negli esempi che seguiranno eviteremo di definire esplicitamente il comportamento della riduzione nel caso in cui la stringa binaria da trasformare non sia sintatticamente corretta.

Esempio 3.7: problema dell'accettazione

Consideriamo il seguente linguaggio: $L_{acc} = \{\langle c_T, x\rangle : c_T \in \mathcal{C} \wedge T(x) \text{ termina in una configurazione finale}\}$. Dimostriamo ora che L_{stop} è riducibile a L_{acc}: dal Teorema 3.5 e dal Corollario 3.2, segue che L_{acc} non è decidibile. Date due stringhe binarie $c_T \in \mathcal{C}$ e x, $f(\langle c_T, x\rangle)$ produce la coppia $\langle c_{T'}, x\rangle$ dove $c_{T'}$ è la codifica di una macchina di Turing T' che, con input una stringa binaria z, esegue T con input z: se $T(z)$ termina, allora T' termina in una configurazione finale. Chiaramente, $\langle c_T, x\rangle \in L_{stop}$ se e solo se $T'(x)$ termina in una configurazione finale se e solo se $\langle c_{T'}, x\rangle = f(\langle c_T, x\rangle) \in L_{acc}$.

Esempio 3.8: problema del linguaggio vuoto

Consideriamo il seguente linguaggio: $L_{empty} = \{c_T : c_T \in \mathcal{C} \wedge \forall x \in \{0,1\}^* [T(x) \text{ non termina in una configurazione finale}]\}$. Dimostriamo ora che L_{acc} è riducibile a L^c_{empty}: dall'esempio precedente, dal Corollario 3.2 e dal Teorema 3.1, segue che L^c_{empty} e L_{empty} non sono decidibili. Date due stringhe binarie $c_T \in \mathcal{C}$ e x, $f(\langle c_T, x\rangle)$ produce la codifica $c_{T'}$ di una macchina di Turing T' che, con input una stringa binaria y, per prima cosa cancella y e lo sostituisce con x e, quindi, esegue T con input x. Abbiamo che $\langle c_T, x\rangle \in L_{acc}$ se e solo se $T(x)$ termina in una configurazione finale se e solo se, per ogni stringa binaria y, $T'(y)$ termina in una configurazione finale se e solo se $c_{T'} \in L^c_{empty}$ (in quanto T' si comporta allo stesso modo indipendentemente dalla stringa di input y).

Esempio 3.9: problema dell'equivalenza tra linguaggi

Consideriamo il seguente linguaggio.

$$L_{eq} = \{\langle c_{T_1}, c_{T_2}\rangle \quad : \quad c_{T_1} \in \mathcal{C} \wedge c_{T_2} \in \mathcal{C} \wedge \forall x \in \{0,1\}^* [T_1(x) \text{ termina in una configurazione finale se e solo se } T_2(x) \text{ termina in una configurazione finale}]\}$$

(in altre parole, L_{eq} include tutte le coppie di codifiche di macchine di Turing che decidono lo stesso linguaggio). Riduciamo ora L_{empty} a L_{eq}: dall'esempio precedente e dal Corollario 3.2, segue che L_{eq} non è decidibile. Data una stringa binaria $c_T \in \mathcal{C}$, $f(c_T)$ produce la coppia $\langle c_T, c_R\rangle$ dove c_R è la codifica di una macchina di Turing R che non accetta alcuna stringa. Chiaramente, $c_T \in L_{empty}$ se e solo $\langle c_T, c_R\rangle \in L_{eq}$.

3.4.1 Il problema della corrispondenza di Post

Concludiamo questo paragrafo fornendo un'ulteriore dimostrazione di non decidibilità basata sulla tecnica della riducibilità. Questo esempio ci permetterà tra l'altro di mostrare una metodologia comunemente utilizzata per dimostrare che un linguaggio L non è decidibile, che consiste nel ridurre a esso ogni possibile computazione di una generica macchina di Turing. Tale metodologia, come vedremo, risulterà essere alla base di uno dei più importanti teoremi della teoria della complessità, ovvero il teorema di Cook.

Dato una sequenza $\mathcal{P}$ di n stringhe binarie $p_1 = \langle x_1, y_1\rangle, \ldots, p_n = \langle x_n, y_n\rangle$, dette *pezzi*, dove x_i e y_i sono stringhe binarie, una **corrispondenza di Post** su $\mathcal{P}$ consiste in una sequenza $p_{i_1}, \ldots, p_{i_k}$ di occorrenze di pezzi di $\mathcal{P}$ (eventualmente con ripetizioni) tale che $x_{i_1} \cdots x_{i_k} = y_{i_1} \cdots y_{i_k}$. Nel seguito, per chiarezza di esposizione, rappresenteremo un pezzo $p = \langle x, y\rangle$ come $\begin{bmatrix} x \\ y \end{bmatrix}$. Inoltre, sempre per chiarezza di esposizione, ammetteremo che le stringhe di un pezzo siano costruite su un alfabeto diverso da quello binario: facendo riferimento al codice binario introdotto nel primo paragrafo del precedente capitolo, ciò non causerà alcuna perdita di generalità.

Consideriamo, ad esempio, il seguente insieme di pezzi.

$$\mathcal{P} = \left\{ \begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix} \right\}$$

Un esempio di corrispondenza di Post su $\mathcal{P}$ è la sequenza

$$\begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} b \\ ca \end{bmatrix} \begin{bmatrix} ca \\ a \end{bmatrix} \begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} abc \\ c \end{bmatrix}$$

in quanto si ha che

$$abcaaabc = abcaaabc$$

Al contrario, se

$$\mathcal{P} = \left\{ \begin{bmatrix} abc \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} acc \\ ba \end{bmatrix} \right\}$$

allora non esiste alcuna corrispondenza di Post su $\mathcal{P}$, in quanto la stringa in alto di ogni pezzo in $\mathcal{P}$ è più lunga della corrispondente stringa in basso.

Definiamo il linguaggio L_{Post} come l'insieme di tutte le sequenze finite di pezzi, contenenti un pezzo speciale p^*, che ammettono almeno una corrispondenza di Post che inizi con p^*: mostreremo ora che L_{Post} non è decidibile. L'idea della dimostrazione consiste nel costruire, data una macchina di Turing T e un input x, una sequenza di pezzi $\mathcal{P}$ tale che $T(x)$ termina in una configurazione finale se e solo se $\mathcal{P} \in L_{Post}$. In altre parole, dimostreremo che L_{acc} è riducibile a L_{Post}: dall'Esempio 3.7 e dal Corollario 3.2, segue che L_{Post} non è decidibile.

La sequenza $\mathcal{P}$ includerà dei pezzi che ci consentiranno di costruire una corrispondenza di Post che contenga (sopra e sotto) la sequenza delle configurazioni della computazione $T(x)$ (separate dal simbolo speciale $\#$ che non fa parte dell'alfabeto di lavoro di T): la sequenza di sotto sarà però sempre un passo avanti rispetto a quella di sopra. Inoltre, $\mathcal{P}$ includerà dei pezzi che, se $T(x)$ termina in una configurazione finale, ci consentiranno di pareggiare la sequenza di sopra con quella di sotto. Nel seguito assumeremo, senza perdita di generalità, che T sia una macchina di Turing con nastro semi-infinito e con un solo stato finale q_1 il cui alfabeto di lavoro sia $\Sigma = \{\square, 0, 1\}$ e la cui testina possa solo eseguire movimenti a destra e a sinistra.

Data una tale macchina di Turing T e una stringa binaria $x = x_1 \cdots x_n$, l'insieme $\mathcal{P}$ contiene, come pezzo speciale, il pezzo

$$\begin{bmatrix} \# \\ \# q_0 x_1 \cdots x_n \# \end{bmatrix}$$

dove q_0 denota lo stato iniziale di T. Per ogni transizione di T da uno stato q a uno stato p in cui il simbolo letto è σ, quello scritto è τ e il movimento della testina è verso destra, aggiungiamo a $\mathcal{P}$ il pezzo

$$\begin{bmatrix} q\sigma \\ \tau p \end{bmatrix}$$

Per ogni transizione di T da uno stato q a uno stato p in cui il simbolo letto è σ, quello scritto è τ e il movimento della testina è verso sinistra, aggiungiamo a $\mathcal{P}$, per ogni simbolo $\gamma \in \Sigma$, i pezzi

$$\begin{bmatrix} \gamma q \sigma \\ p \gamma \tau \end{bmatrix}$$

Per ogni simbolo $\sigma \in \Sigma$, aggiungiamo a $\mathcal{P}$ i pezzi

$$\left[\begin{array}{c}\sigma\\ \sigma\end{array}\right], \left[\begin{array}{c}\sigma q_1\\ q_1\end{array}\right] \text{ e } \left[\begin{array}{c}q_1\sigma\\ q_1\end{array}\right]$$

Infine, aggiungiamo a $\mathcal{P}$ i pezzi

$$\left[\begin{array}{c}\#\\ \#\end{array}\right], \left[\begin{array}{c}\#\\ \square\#\end{array}\right] \text{ e } \left[\begin{array}{c}q_1\#\#\\ \#\end{array}\right]$$

Piuttosto che dimostrare formalmente che tale costruzione di $\mathcal{P}$ costituisce una riduzione da L_{acc} a L_{Post}, vediamo con un esempio come sia possibile simulare la computazione di T con input x mediante una corrispondenza di Post su $\mathcal{P}$. Supponiamo che x sia la stringa `01` e che T, leggendo `0` nello stato iniziale, scriva `1`, si sposti a destra ed entri nello stato q_2. Poiché $\mathcal{P}$ contiene i pezzi $\left[\begin{array}{c}\#\\ \#q_0 01\#\end{array}\right]$, $\left[\begin{array}{c}q_0 0\\ 1q_2\end{array}\right]$, $\left[\begin{array}{c}1\\ 1\end{array}\right]$ e $\left[\begin{array}{c}\#\\ \#\end{array}\right]$, possiamo costruire la seguente porzione iniziale di corrispondenza.

$$\left[\begin{array}{c}\#\\ \#q_0 01\#\end{array}\right]\left[\begin{array}{c}q_0 0\\ 1q_2\end{array}\right]\left[\begin{array}{c}1\\ 1\end{array}\right]\left[\begin{array}{c}\#\\ \#\end{array}\right]$$

Osserviamo come la concatenazione delle stringhe in alto produca la stringa $\#q_0 01\#$ mentre quella delle stringhe in basso produca la stringa $\#q_0 01\#1q_2 1\#$: in altre parole, abbiamo nella parte sopra duplicato la configurazione iniziale di T e prodotto nella parte sotto la configurazione successiva a quella iniziale. Continuando con l'esempio, supponiamo che T, leggendo `1` nello stato q_2, scriva `0`, si sposti a destra ed entri nello stato q_3. Poiché $\mathcal{P}$ contiene i pezzi $\left[\begin{array}{c}q_2 1\\ 0q_3\end{array}\right]$, $\left[\begin{array}{c}1\\ 1\end{array}\right]$ e $\left[\begin{array}{c}\#\\ \square\#\end{array}\right]$, possiamo estendere la corrispondenza nel modo seguente.

$$\left[\begin{array}{c}1\\ 1\end{array}\right]\left[\begin{array}{c}q_2 1\\ 0q_3\end{array}\right]\left[\begin{array}{c}\#\\ \square\#\end{array}\right]$$

Ancora una volta abbiamo nella parte sopra duplicato la configurazione attuale di T e prodotto nella parte sotto la configurazione immediatamente successiva. Supponiamo ora che T, leggendo $\square$ nello stato q_3, scriva $\square$, si sposti a sinistra ed entri nello stato q_1. Poiché $\mathcal{P}$ contiene i pezzi $\left[\begin{array}{c}0q_3\square\\ q_1 0\square\end{array}\right]$, $\left[\begin{array}{c}1\\ 1\end{array}\right]$ e $\left[\begin{array}{c}\#\\ \#\end{array}\right]$, possiamo estendere la corrispondenza nel modo seguente.

$$\left[\begin{array}{c}1\\ 1\end{array}\right]\left[\begin{array}{c}0q_3\square\\ q_1 0\square\end{array}\right]\left[\begin{array}{c}\#\\ \#\end{array}\right]$$

Anche questa volta abbiamo nella parte sopra duplicato la configurazione attuale di T e prodotto nella parte sotto la configurazione immediatamente successiva. Poiché q_1 è lo stato finale di T, $\mathcal{P}$ include i pezzi $\begin{bmatrix} q_1 0 \\ q_1 \end{bmatrix}$, $\begin{bmatrix} q_1 \square \\ q_1 \end{bmatrix}$, $\begin{bmatrix} 1 q_1 \\ q_1 \end{bmatrix}$ e $\begin{bmatrix} q_1 \# \# \\ \# \end{bmatrix}$, oltre ai pezzi $\begin{bmatrix} \square \\ \square \end{bmatrix}$, $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ e $\begin{bmatrix} \# \\ \# \end{bmatrix}$: questi pezzi ci permettono a questo punto di estendere e terminare la corrispondenza nel modo seguente.

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} q_1 0 \\ q_1 \end{bmatrix} \begin{bmatrix} \square \\ \square \end{bmatrix} \begin{bmatrix} \# \\ \# \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} q_1 \square \\ q_1 \end{bmatrix} \begin{bmatrix} \# \\ \# \end{bmatrix} \begin{bmatrix} 1 q_1 \\ q_1 \end{bmatrix} \begin{bmatrix} \# \\ \# \end{bmatrix} \begin{bmatrix} q_1 \# \# \\ \# \end{bmatrix}$$

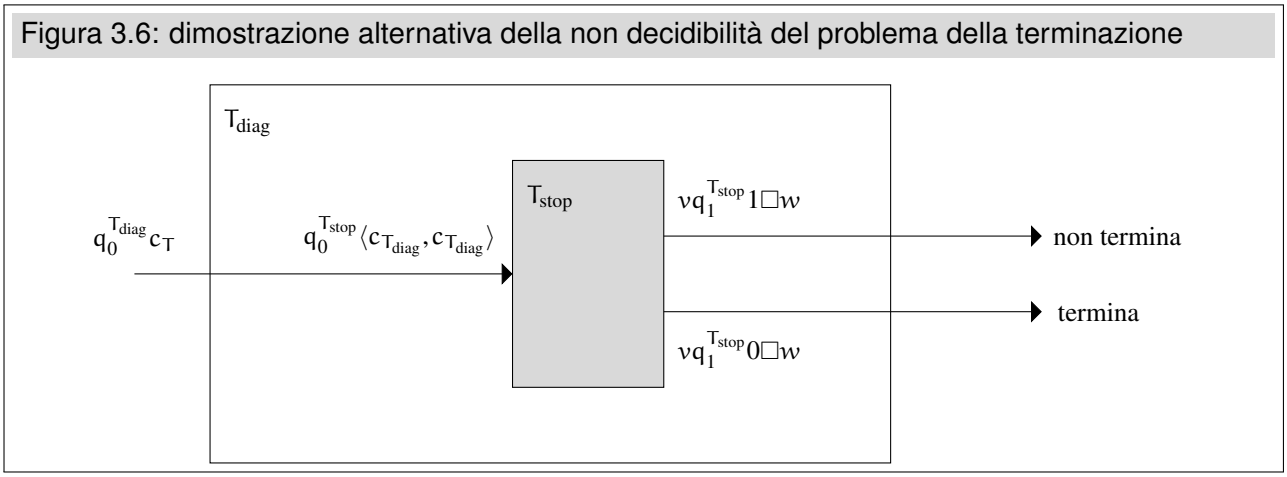
Osserviamo che, se non avessimo imposto il vincolo che il primo pezzo da utilizzare fosse $\begin{bmatrix} \# \\ \# q_0 x_1 \cdots x_n \# \end{bmatrix}$, allora esisterebbe sempre una corrispondenza di Post, indipendentemente dal fatto che T con input x termini o meno in una configurazione finale: infatti, ciò è dovuto al fatto che $\mathcal{P}$ contiene diversi pezzi composti da stringhe uguali e che ciascuno di essi costituisce chiaramente una corrispondenza di Post. È possibile, comunque, ridurre il linguaggio di Post con il vincolo del pezzo iniziale al linguaggio senza tale vincolo, mostrando così come anche quest'ultimo linguaggio non sia decidibile.

3.5 Teorema della ricorsione

IL TEOREMA della ricorsione è un risultato matematico che svolge un ruolo molto importante nello sviluppo della teoria della calcolabilità e che ha connessioni con la logica matematica, con la teoria dei sistemi ad auto-riproduzione e anche con i virus informatici. Intuitivamente, tale teorema afferma che è possibile costruire macchine di Turing che riproducono se stesse. In questo paragrafo daremo una dimostrazione del teorema della ricorsione e ne mostreremo alcune applicazioni.

Per giustificare il nostro interesse nei confronti del teorema della ricorsione, consideriamo la seguente dimostrazione alternativa del fatto che il problema della terminazione non sia decidibile (si veda anche la Figura 3.6). Supponiamo, per assurdo, che L_{stop} sia decidibile e che, quindi, esista una macchina di Turing T_{stop} che decide L_{stop}. Definiamo, allora, una macchina di Turing T_{diag} che, con input una stringa binaria $c_T \in C$, esegue T_{stop} con input la coppia $\langle c_{T_{diag}}, c_{T_{diag}} \rangle$: se T_{stop} termina con la testina posizionata su un simbolo 1, allora T_{diag} non termina, altrimenti termina. Abbiamo, quindi, che $T_{diag}(c_{T_{diag}})$ non termina se e solo se $T_{stop}(\langle c_{T_{diag}}, c_{T_{diag}} \rangle)$ termina con la testina posizionata su un simbolo 1 se e solo $T_{diag}(c_{T_{diag}})$ termina: tale contraddizione dimostra che la macchina di Turing T_{stop} non può esistere e che, quindi, L_{stop} non è decidibile.

Figura 3.6: dimostrazione alternativa della non decidibilità del problema della terminazione



Nella dimostrazione sopra esposta, abbiamo definito una macchina di Turing T_{diag} che usa la sua stessa codifica $c_{T_{diag}}$. La domanda che allora ci poniamo è se ciò sia possibile: ovvero, può una macchina di Turing avere accesso alla sua propria codifica, oppure per scrivere la codifica di una macchina di Turing T dobbiamo definire una macchina di Turing "più potente" di T? Il teorema della ricorsione consente di rispondere a tale domanda, mostrando che esistono macchine di Turing che possono stampare la propria codifica e, successivamente, utilizzarla per eseguire ulteriori calcoli.

L'idea della dimostrazione del teorema della ricorsione si basa sulle seguenti considerazioni. Supponiamo di volere scrivere una frase in italiano che induca chi la legge a scrivere la stessa frase. Una prima soluzione a tale problema potrebbe essere la frase seguente.

```
Scrivi questa frase.
```

Da un punto di vista semantico, questa frase otterrebbe lo scopo previsto. Tuttavia, essa contiene una componente di auto-referenzialità (il termine "questa") che la rende irrealizzabile in un qualunque linguaggio di programmazione. Tale auto-referenzialità può però essere eliminata se consideriamo la seguente frase alternativa.

```
Scrivi due copie della frase racchiusa tra virgolette che
segue i due punti, racchiudendo la seconda copia tra
virgolette:``Scrivi due copie della frase racchiusa tra
virgolette che segue i due punti, racchiudendo la seconda
copia tra virgolette:''
```

Da un punto di vista più informatico, quindi, l'idea è quella di scrivere un programma P che, applicato a una stringa Q (che in realtà è il testo di P), riproduca tale stringa seguita da una versione di Q racchiusa tra virgolette. Non è difficile realizzare tale programma in un qualunque linguaggio di programmazione: ad esempio, facendo riferimento a Java, un programma che produca se stesso può essere definito procedendo nel modo seguente. Anzitutto, definiamo una prima versione della classe corrispondente al programma P (nel seguito, per motivi di leggibilità, includeremo nel codice gli accapo, ma questi ultimi devono in realtà essere eliminati).

```
class P {
  public static void main(String[] a) {
    char quote = 34;
    System.out.print(Q+quote+Q+quote);
  }
  static String Q = "frase";
}
```

Eseguendo il metodo principale di tale classe, il risultato ottenuto sarà la stampa sullo schermo della seguente stringa.

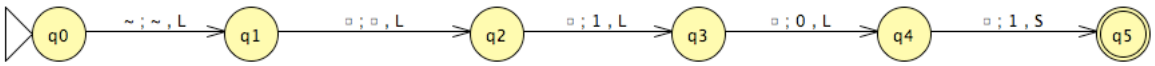
```
frase"frase"
```

A questo punto, è sufficiente assegnare alla variabile `Q` il testo del programma stesso, fermandosi al punto in cui apparirebbe la frase fra virgolette e ottenendo così la definizione della seguente classe.

```
class P {
  public static void main(String[] a) {
    char quote = 34;
    System.out.print(Q+quote+Q+quote);
  }
  static String Q = "class P { public ... String Q = ";
}
```

Eseguendo nuovamente il metodo principale, ci accorgiamo che il risultato ottenuto sarà la stampa del codice stesso a eccezione dell'ultimo punto e virgola e dell'ultima parentesi graffa. Per ottenere esattamente lo stesso codice è quindi sufficiente modificare la definizione della classe `P` nel modo seguente (chiaramente, anche la stringa assegnata alla variabile `Q` deve essere opportunamente modificata).

Figura 3.7: la macchina di Turing P_{101}.




```
class P {
  public static void main(String[] a) {
    char quote = 34;
    System.out.print(Q+quote+Q+quote+';'+ '+'}');
  }
  static String Q = "class P { public ... String Q = ";
}
```

Proviamo, ad esempio, a eseguire il metodo principale di tale classe in ambiente Linux e a vedere il risultato ottenuto, digitando i seguenti comandi.

```
javac P.java
java P > output
diff P.java output
```

Al termine dell'esecuzione del comando `diff`, nessun messaggio sarà generato, dimostrando così che la stringa prodotta dalla classe `P` è esattamente uguale alla definizione della classe `P` stessa.

La dimostrazione del teorema della ricorsione, il quale afferma che le macchine di Turing hanno la capacità di produrre la loro stessa codifica, per poi effettuare delle computazioni su di essa, segue essenzialmente il procedimento appena descritto (anche se, per forza di cose, tale dimostrazione risulta essere leggermente più involuta).

Teorema 3.6

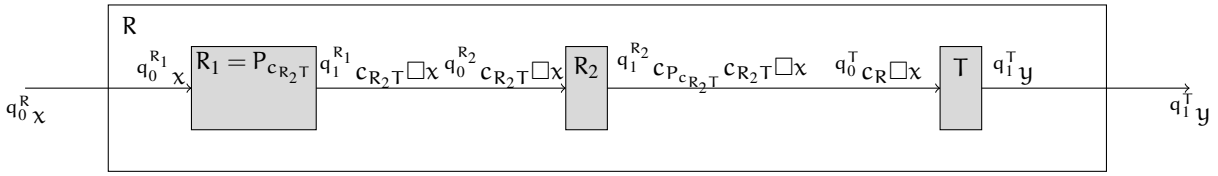
Per ogni macchina di Turing $T \in \mathcal{T}$, esiste una macchina di Turing $R \in \mathcal{T}$ tale che, per ogni stringa binaria x, R con input x termina se e solo se T con input $c_R \square x$ termina e R con input x termina nella stessa configurazione in cui termina T con input $c_R \square x$.

Dimostrazione. La macchina di Turing R è definita come la composizione[2] di R_1, R_2 e T, dove, intuitivamente, R_2 corrisponde al programma P a cui abbiamo fatto riferimento nella discussione precedente e R_1 corrisponde al testo dello stesso programma, racchiuso tra virgolette, e fornisce, quindi, una copia della macchina R_2 che la macchina R_2 stessa può elaborare. In particolare, R_1 avrà il compito di produrre la codifica c_{R_2T} della composizione di R_2 con T e R_2 avrà il compito di produrre, in fronte a c_{R_2T}, la codifica c_{R_1} di R_1. Pertanto, la composizione di R_1 con R_2 produrrà la codifica $c_{R_1R_2T}$ della composizione di R_1 con R_2 e, quindi, con T, ovvero la codifica di R. Successivamente, il controllo passerà a T, che sarà eseguita con input $c_R\square x$: il risultato finale sarà, dunque, lo stesso dell'esecuzione di T con input $c_R\square x$. Data una stringa binaria y, definiamo la macchina di Turing P_y nel modo seguente (si veda la Figura 3.7): per ogni stringa binaria z, P_y con input z, termina lasciando sul nastro la sola stringa $y\square z$ con la testina posizionata sul primo simbolo di y. La macchina di Turing R_1 è definita come $P_{c_{R_2T}}$. Tale definizione dipende chiaramente dal fatto di avere a disposizione la definizione della macchina R_2: quindi, per poterla completare, dobbiamo definire la costruzione di quest'ultima macchina. A tale scopo, osserviamo che, per ogni stringa binaria y, la codifica di P_y dipende da y e può essere facilmente calcolata a partire da y stessa: facendo riferimento alla Figura 3.7, ad esempio, la codifica di P_y cambia al variare di y solo per quello che riguarda la sequenza di transizioni dallo stato q_2 a quello finale. La macchina di Turing R_2 è allora una macchina che, per ogni coppia di stringhe binarie z_1 e z_2, con input la stringa $z_1\square z_2$ termina lasciando sul nastro la sola stringa $c_{P_{z_1}}z_1\square z_2$ con la testina posizionata sul primo simbolo di $c_{P_{z_1}}$. Per ogni stringa binaria x, l'esecuzione di R_1 con input x lascia sul nastro la sola stringa binaria $c_{R_2T}\square x$: la successiva esecuzione di R_2, con input la stringa binaria $c_{R_2T}\square x$, lascia sul nastro la sola stringa binaria $c_{P_{c_{R_2T}}}c_{R_2T}\square x = c_{R_1}c_{R_2T}\square x = c_{R_1R_2T}\square x = c_R\square x$. Poiché successivamente a R_2 viene eseguita T, il risultato finale sarà lo stesso dell'esecuzione di T con input $c_R\square x$ (si veda la Figura 3.8): il teorema risulta così essere dimostrato. ◇

Il teorema della ricorsione consente, quindi, di definire macchine di Turing in grado di produrre la loro codifica per poi utilizzarla in succesive elaborazioni, che possono dipendere dall'input iniziale: questo sarà il modo principale con cui applicheremo, nel seguito, tale teorema allo scopo di dimostrare ulteriori risultati di non decidibilità.

[2]Nel seguito assumeremo, senza perdita di generalità, che le macchine di Turing siano definite in modo tale che, date due macchine di Turing T_1 e T_2, la codifica $c_{T_1T_2}$ della composizione di T_1 con T_2 (ovvero della macchina di Turing ottenuta definendo una transizione dallo stato finale di T_1 a quello iniziale di T_2) sia uguale alla concatenazione $c_{T_1}c_{T_2}$ della codifica di T_1 con la codifica di T_2.

Figura 3.8: la dimostrazione del teorema della ricorsione



3.5.1 Applicazioni del teorema della ricorsione

OLTRE A essere un risultato di per sé affascinante, il teorema della ricorsione è uno strumento molto potente per ottenere altri interessanti risultati della teoria della calcolabilità. Un primo tipo di applicazione di tale teorema consiste nel dimostrare proprietà di indecidibilità di un determinato linguaggio: per mostrare un esempio di tale applicazione, introduciamo prima il concetto di macchina di Turing "generatrice" di un linguaggio.

Definizione 3.7: generatori

Una macchina di Turing $T \in \mathcal{T}$ è detta essere un **generatore** di un linguaggio L, se, con input la stringa vuota λ, T stampa sul nastro una dopo l'altra (separate da un $\square$), senza mai terminare, tutte le (potenzialmente infinite) stringhe contenute in L, in un qualunque ordine e eventualmente con ripetizione.

Il prossimo risultato mostra come l'esistenza di un generatore non sia altro che un diverso modo di caratterizzare la semi-decidibilità di un linguaggio.

Teorema 3.7

Un linguaggio $L \in \mathcal{L}$ è semi-decidibile se e solo se L ammette un generatore.

Dimostrazione. Sia L un linguaggio che ammette un generatore T_{gen}: dimostriamo che L è semi-decidibile. A tale scopo, definiamo una macchina di Turing T nel modo seguente. Per ogni stringa binaria x, T con input x simula T_{gen} e, ogni qualvolta T_{gen} scrive una nuova stringa y, verifica se x è uguale a y: in tal caso termina in uno stato finale, altrimenti prosegue con la simulazione di T_{gen}. Chiaramente, se $x \in L$, allora x verrà prima o poi generata da T_{gen} e T terminerà in una configurazione finale, altrimenti T_{gen} non terminerà (e, quindi, neanche T). Dimostriamo ora che, se un linguaggio L è semi-decidibile, allora L ammette un generatore. A tale scopo, assumiamo di numerare l'insieme di tutte le possibili stringhe binarie x_i: possiamo definire un generatore T_{gen} per L nel modo seguente. Per ogni $i \geqslant 0$, T_{gen} esegue

i primi i passi di $T(x_0)$, i primi i passi di $T(x_1)$, ..., i primi i passi di $T(x_i)$: se una di queste esecuzioni termina in uno stato finale, allora T_{gen} produce la stringa di input corrispondente. In questo modo, se una stringa x appartiene a L, prima o poi T_{gen} eseguirà $T(x)$ per un numero sufficiente di passi da raggiungere lo stato finale e, quindi, produrrà la stringa x. Al contrario, se $x \notin L$, nessuna esecuzione di $T(x)$ da parte di T_{gen} potrà terminare nello stato finale e, quindi, la stringa x non sarà mai prodotta da T_{gen}. ◇

Abbiamo già visto, nel terzo paragrafo di questo capitolo, un esempio di linguaggio che non sia semi-decidibile: mostriamo ora un ulteriore esempio di un tale linguaggio, facendo uso del teorema precedente e del teorema della ricorsione.

Definizione 3.8: macchine di Turing equivalenti

Due macchine di Turing $T, T' \in \mathcal{T}$ sono dette essere **equivalenti** (in simboli, $T \equiv T'$) se, per ogni stringa binaria x, T con input x termina se e solo se T' con input x termina e T con input x termina in una configurazione finale se e solo se T' con input x termina in una configurazione finale.

In altre parole, due macchine di Turing equivalenti decidono oppure semi-decidono lo stesso linguaggio. Data una macchina di Turing $T \in \mathcal{T}$, diremo che T è **minimale** se non esiste un'altra macchina di Turing $T' \in \mathcal{T}$ tale che $|c_{T'}| < |c_T|$ e $T \equiv T'$. Consideriamo il seguente linguaggio.

$$L_{min} = \{c_T : c_T \in \mathcal{C} \text{ e } T \text{ è minimale}\}$$

Teorema 3.8

L_{min} non è semi-decidibile.

Dimostrazione. Supponiamo per assurdo che L_{min} sia semi-decidibile: dal Teorema 3.7 segue che esiste un generatore T_{gen} per L. Facendo uso del teorema della ricorsione, definiamo allora la seguente macchina di Turing T: per ogni stringa x, T con input x calcola la sua propria codifica c_T, simula T_{gen} fino a quando viene prodotta una codifica $c_{T_>}$ più lunga di c_T e, infine, simula $T_>$ con input x. Poiché L_{min} è chiaramente un linguaggio infinito, prima o poi T_{gen} produrrà una codifica $c_{T_>}$ più lunga di c_T: quindi, T è equivalente a $T_>$. Tuttavia, la codifica di T è più corta di quella di $T_>$ contraddicendo il fatto che, essendo stata prodotta da un generatore di L_{min}, $c_{T_>}$ sia la codifica di una macchina minimale. Pertanto, non può esistere il generatore T_{gen} e, quindi, L_{min} non è un linguaggio semi-decidibile. ◇

La seconda applicazione del teorema della ricorsione è un risultato di esistenza del cosiddetto **punto fisso** di una funzione, ovvero di un valore che non viene

modificato dall'applicazione a esso della funzione stessa. In particolare, il prossimo teorema mostra che, per ogni funzione t che trasformi codifiche di macchine di Turing in codifiche di macchine di Turing, esiste una macchina di Turing T la cui codifica viene trasformata dalla funzione t nella codifica di una macchina di Turing equivalente a T.

Teorema 3.9

Sia $t : \{0,1\}^* \to \{0,1\}^*$ una funzione totale calcolabile, che trasforma codifiche di macchine di Turing in codifiche di macchine di Turing. Esiste una macchina di Turing $T_{fix} \in \mathcal{T}$ per cui $t(c_{T_{fix}})$ è la codifica di una macchina di Turing equivalente a T_{fix}.

Dimostrazione. Sia T_t la macchina di Turing che calcola t: facendo uso del teorema della ricorsione, definiamo T_{fix} nel modo seguente. Per ogni stringa x, T_{fix} con input x calcola la sua propria codifica $c_{T_{fix}}$, simula T_t con input $c_{T_{fix}}$ ottenendo una codifica $c_{T'}$ di una macchina di Turing, e, infine, simula T' con input x. Poiché nell'ultimo passo della computazione appena descritta T_{fix} simula T', si ha che $c_{T_{fix}}$ e $t(c_{T_{fix}}) = c_{T'}$ codificano due macchine equivalenti: il teorema risulta dunque essere dimostrato. $\diamond$

L'ultima applicazione del teorema della ricorsione è in realtà un'applicazione del teorema del punto fisso appena dimostrato, e costituisce uno dei più forti risultati negativi della teoria della calcolabilità. Intuitivamente, tale risultato (anche detto **teorema di Rice**) afferma che ogni proprietà non "banale" di macchine di Turing non può essere decisa da una macchina di Turing (una proprietà è non banale se non è soddisfatta da tutte le macchine di Turing e se è soddisfatta da almeno una macchina di Turing).[3]

Teorema 3.10

Sia $L \in \mathcal{L}$ un insieme di codifiche di macchine di Turing per cui esiste una macchina di Turing la cui codifica appartiene a L ed esiste una macchina di Turing la cui codifica non appartiene a L. Inoltre, per ogni coppia di macchine di Turing equivalenti T_1 e T_2, $c_{T_1} \in L$ se e solo se $c_{T_2} \in L$. Allora, L non è decidibile.

Dimostrazione. Sia $c_{T_{si}}$ la codifica di una macchina di Turing che appartiene a L e $c_{T_{no}}$ la codifica di una macchina di Turing che non appartiene a L. Definiamo allora la seguente macchina di Turing T: per ogni stringa x, se x è la codifica di una macchina di Turing che non appartiene a L, T con input x produce in output $c_{T_{si}}$, altrimenti produce in output $c_{T_{no}}$. Se, per assurdo, L fosse decidibile, allora T calcolerebbe una funzione totale $t : \Sigma^* \to \Sigma^*$, che trasforma codifiche di macchine di Turing in

[3] In realtà, la proprietà deve anche essere chiusa rispetto all'equivalenza di macchine di Turing, nel senso che se essa è soddisfatta da una macchina di Turing T, allora è anche soddisfatta da tutte le macchine di Turing equivalenti a T.

codifiche di macchine di Turing. Osserviamo che, in base alla definizione di T, per ogni codifica c di una macchina di Turing, $t(c) \in L$ se e solo se $c \notin L$. Dal Teorema 3.9 segue che esiste una macchina di Turing T_{fix} tale che $t(c_{T_{fix}})$ è la codifica di una macchina di Turing equivalente a T_{fix}. In base alle proprietà di L, si ha quindi che $c_{T_{fix}} \in L$ se e solo se $t(c_{T_{fix}}) \in L$, contraddicendo il fatto che, per ogni codifica c di una macchina di Turing, $t(c) \in L$ se e solo se $c \notin L$. Pertanto, l'insieme L non può essere decidibile e il teorema risulta essere dimostrato. ◇

Vediamo, ad esempio, come il teorema precedente possa essere utilizzato per fornire una dimostrazione alternativa dell'indecidibilità di L_{stop-0}. Sia L l'insieme delle codifiche di macchine di Turing che terminano ricevendo in input la stringa 0. Chiaramente, L soddisfa le ipotesi del teorema di Rice, in quanto è facile definire una macchina di Turing che, con input la stringa 0, termina e una che, con input la stringa 0, non termina. Inoltre, se due macchine T e T' sono equivalenti, esse avranno lo stesso comportamento con input la stringa 0 e, quindi, le loro due codifiche appartengono entrambe a L oppure non vi appartengono entrambe. Quindi, il teorema di Rice ci consente di affermare che $L = L_{stop-0}$ non è decidibile.

Esercizi

Esercizio 3.1. Dimostrare che se $L \in \mathcal{L}$ oppure $L^c \in \mathcal{L}$ è un linguaggio finito, allora L e L^c sono due linguaggi decidibili.
Esercizio 3.2. Dimostrare che se $L_1, L_2 \in \mathcal{L}$ sono due linguaggi decidibili, allora anche $L_1 \cup L_2$ e $L_1 \cap L_2$ sono decidibili.
Esercizio 3.3. Dimostrare che il seguente linguaggio è decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ e } T(\lambda) \text{ scrive prima o poi un simbolo diverso da } \square\}$$

Esercizio 3.4. Dimostrare che la relazione di equi-cardinalità è una relazione di equivalenza.
Esercizio 3.5. Sia $A = \{(i,j,k) : i,j,k \in N\}$. Dimostrare che A è numerabile.
Esercizio 3.6. Dimostrare che l'insieme di tutti i sottoinsiemi finiti di $\{0,1\}^*$ è un insieme numerabile.
Esercizio 3.7. Dimostrare che l'insieme di tutte le stringhe infinite binarie contenenti esattamente due 1 è numerabile.
Esercizio 3.8. Dimostrare che l'insieme di tutte le stringhe infinite binarie non è numerabile.
Esercizio 3.9. Dire, giustificando la risposta, se le seguenti affermazioni sono vere o false.

1. Se $L \in \mathcal{L}$ è un linguaggio decidibile e $L' \in \mathcal{L}$ è un linguaggio semi-decidibile, allora il linguaggio $L \cap L'$ è decidibile.
2. Se $L, L' \in \mathcal{L}$ sono due linguaggi semi-decidibili, allora il linguaggio $L \cap L'$ è semi-decidibile.

3. Se $L, L' \in \mathcal{L}$ sono due linguaggi semi-decidibili, allora il linguaggio $L - L'$ è semi-decidibile.
4. Se $L \in \mathcal{L}$ è un linguaggio semi-decidibile ma non decidibile, allora una qualunque macchina di Turing che semi-decide L deve non terminare per un numero infinito di stringhe binarie di input.
5. Se $L \in \mathcal{L}$ è un linguaggio semi-decidibile ma non decidibile, allora non esiste un linguaggio infinito $L' \subseteq L$ che sia decidibile.
6. Dati due linguaggi decidibili $L_1, L_2 \in \mathcal{L}$, il seguente linguaggio è decidibile.

$$L = \{xy : x \in L_1 \wedge y \in L_2\}$$

7. Siano $L_1, L_2, L_3 \in \mathcal{L}$ tre linguaggi semi-decidibili tali che $L_1 \cap L_2 = \emptyset$, $L_1 \cap L_3 = \emptyset$, $L_2 \cap L_3 = \emptyset$ e $L_1 \cup L_2 \cup L_3 = \{0,1\}^*$. Allora L_1, L_2 e L_3 sono decidibili.

Esercizio 3.10. Dimostrare che un linguaggio $L \in \mathcal{L}$ è semi-decidibile se e solo se esiste un linguaggio decidibile $L' \in \mathcal{L}$ tale che, per ogni stringa binaria x, $x \in L$ se e solo se esiste una stringa binaria y per cui $\langle x, y \rangle \in L'$.

Esercizio 3.11. Facendo uso della tecnica della riducibilità, dimostrare che il seguente linguaggio non è decidibile.

$$L = \{c_T \quad : \quad c_T \in \mathcal{C} \text{ e se } T(x) \text{ termina in una configurazione finale, allora } x \text{ è formata da una sequenza di } \texttt{0} \text{ seguita da una sequenza di } \texttt{1}\}$$

Esercizio 3.12. Facendo uso della tecnica della riducibilità, dimostrare che il seguente linguaggio non è decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ e } T(\texttt{0}) \text{ termina in una configurazione finale}\}$$

Esercizio 3.13. Facendo uso della tecnica della riducibilità, dimostrare che il seguente linguaggio non è decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ e se } T(x) \text{ termina in una configurazione finale, allora } |x| \text{ è pari}\}$$

Esercizio 3.14. Facendo uso della tecnica della riducibilità, dimostrare che il seguente linguaggio non è decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ e } |\{x : T(x) \text{termina in una configurazione finale}\}| < \infty\}$$

Esercizio 3.15. Facendo uso della tecnica della riducibilità, dimostrare che il seguente linguaggio non è decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ ed esiste una stringa } x \text{ per cui } T(x) \text{ produce in output la stringa } \mathtt{0}\}$$

Esercizio 3.16. Facendo uso della tecnica di riducibilità dimostrare che il seguente linguaggio non è decidibile.

$$L = \{(c_{T_1}, c_{T_2}) \quad : \quad c_{T_1}, c_{T_2} \in \mathcal{C} \text{ e se } T_1(x) \text{ termina in una configurazione finale, allora } T_2(x) \text{ termina in una configurazione finale}\}$$

Esercizio 3.17. Facendo uso della tecnica di riducibilità dimostrare che il seguente linguaggio non è decidibile.

$$L = \{(c_{T_1}, c_{T_2}) \quad : \quad c_{T_1}, c_{T_2} \in \mathcal{C} \text{ ed esiste una e una sola stringa binaria } x \text{ per cui } T_1(x) \text{ e } T_2(x) \text{ terminano entrambe in una configurazione finale}\}$$

Esercizio 3.18. Abbiamo già osservato, nel capitolo precedente, che un linguaggio è detto essere regolare se può essere deciso da un automa a stati finiti (ovvero da una macchina di Turing che può solo leggere e che può solo spostarsi a destra) e che, come vedremo nella seconda parte di queste dispense, il linguaggio $L = \{\mathtt{0}^n\mathtt{1}^n : n \geqslant 0\}$ non è regolare. Dimostrare che il seguente linguaggio non è decidibile: $L = \{c_T : c_T \in \mathcal{C} \text{ e } T \text{ decide un linguaggio regolare}\}$.

Esercizio 3.19. Dire se la seguente affermazione è vera o falsa: "se $L_1, L_2 \in \mathcal{L}$ sono due linguaggi tali che L_1 è riducibile a L_2 e L_2 è regolare, allora L_1 è regolare". Giustificare la risposta.

Esercizio 3.20. Dimostrare che il linguaggio di Post con il vincolo del pezzo iniziale è riducibile al linguaggio senza tale vincolo.

Esercizio 3.21. È ovvio che se un linguaggio $L_1 \in \mathcal{L}$ è riducibile a un linguaggio $L_2 \in \mathcal{L}$ e se L_1 non è semi-decidibile, allora anche L_2 non è semi-decidibile. È anche evidente che se L_1 è riducibile a L_2, allora anche L_1^c è riducibile a L_2^c. Usando questi due fatti e il Corollario 3.1, dimostrare che il linguaggio L_{eq} definito nell'Esempio 3.9 e il suo complementare L_{eq}^c non sono semi-decidibili.

Esercizio 3.22. Dimostrare che il seguente linguaggio non è semi-decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ e } T(\mathtt{101}) \text{ non termina in una configurazione finale}\}$$

Esercizio 3.23. Si consideri il seguente linguaggio.

$$L = \{c_T \quad : \quad c_T \in \mathcal{C} \text{ ed esiste una stringa } x \text{ per cui } T(x) \text{ termina in una configurazione finale dopo aver eseguito al più 1000 passi}\}$$

Dire se L è decidibile oppure se L è semi-decidibile ma non decidibile oppure se L^c è semi-decidibile e L non è decidibile. Giustificare la risposta.

Esercizio 3.24. Dimostrare che non si perde in generalità se si assume che, date due macchine di Turing T_1 e T_2, la codifica $c_{T_1T_2}$ della composizione di T_1 con T_2 sia uguale alla concatenazione $c_{T_1}c_{T_2}$ della codifica di T_1 con la codifica di T_2.

Esercizio 3.25. Data una macchina di Turing $T \in \mathcal{T}$, diremo che T è *quasi minimale* se non esiste un'altra macchina di Turing $T' \in \mathcal{T}$ equivalente a T tale che $|c_{T'}| < \frac{|c_T|}{2}$. Facendo uso del teorema della ricorsione, dimostrare che il seguente linguaggio non è semi-decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ e } T \text{ è quasi-minimale}\}$$

Esercizio 3.26. Utilizzando il teorema della ricorsione, dimostrare che L_{acc} non è decidibile.

Esercizio 3.27. Facendo uso del teorema della ricorsione, dimostrare che il seguente linguaggio non è decidibile.

$$L = \{c_T \quad : \quad c_T \in \mathcal{C}, T(\texttt{11}) \text{ termina in una configurazione finale e } T(\texttt{00}) \text{ non termina in una configurazione finale}\}$$

Esercizio 3.28. Dire, giustificando la risposta, se è possibile applicare il teorema di Rice per dimostrare che il seguente linguaggio non è decidibile.

$$L = \{c_T : c_T \in \mathcal{C} \text{ e } T(c_T) \text{ termina in al più } |c_T| \text{ passi}\}$$

Capitolo 4

La tesi di Church-Turing

SOMMARIO

Sebbene il concetto di algoritmo abbia avuto una lunga storia nel campo della matematica, la sua definizione formale non fu introdotta prima dell'inizio del diciannovesimo secolo: prima di allora, i matematici avevano una nozione intuitiva di cosa fosse un algoritmo e su di essa facevano affidamento per usare e descrivere algoritmi. Tuttavia, tale nozione intuitiva era insufficiente per comprendere appieno le potenzialità del calcolo algoritmico: per questo motivo, come già detto nell'introduzione, diversi ricercatori nel campo della logica matematica proposero una definizione formale del concetto di algoritmo. In questo capitolo, presenteremo quattro di queste definizioni formali, dimostrando, per ciascuna di esse, come essa non introduca un modello di calcolo più potente delle macchine di Turing e, per tre di esse, la loro equivalenza con il modello di calcolo della macchina di Turing: in base a tali risultati, enunceremo infine quella che tutt'ora è nota coma la tesi di Church-Turing.

4.1 Algoritmi di Markov

All'inizio degli anni cinquanta il matematico e logico russo Markov, propose un modello di calcolo basato sull'elaborazione di stringhe, che dimostrò essere equivalente alle macchine di Turing. A differenza delle macchine di Turing, gli algoritmi di Markov operano esclusivamente in funzione del contenuto della stringa e non in funzione di uno stato interno dell'algoritmo.

In particolare, un **algoritmo di Markov** è definito mediante una serie di **regole di riscrittura** che permettono di trasformare, in modo univoco, il contenuto di una stringa in un'altra stringa. Formalmente, esso consiste di:

- un alfabeto Σ, di cui un sotto-insieme Γ costituisce l'alfabeto dei caratteri di cui può essere costituita la stringa iniziale o di input;

- un insieme finito e ordinato R di regole del tipo $x \to y$ oppure del tipo $x \Rightarrow y$, dove x e y sono stringhe su Σ (le regole del secondo tipo sono anche dette regole **terminali**).

Una regola $x \to y$ o $x \Rightarrow y$ si dice essere **applicabile** a una stringa z su Σ, se $z = z_1 x z_2$, dove z_1 e z_2 sono due stringhe su Σ tali che $z_1 x$ contiene una sola occorrenza della stringa x: **applicare** la regola a z, significa sostituire la prima occorrenza di x in z con y.

Partendo da una stringa $x \in \Gamma^*$ di input, un algoritmo di Markov opera nel modo seguente.

1. Scandisce tutte le regole in R nell'ordine stabilito, alla ricerca di una che sia applicabile alla stringa corrente: se nessuna regola viene trovata, l'algoritmo termina producendo in output la stringa corrente.

2. Se r è la regola trovata al passo precedente, applica r alla stringa corrente: la nuova stringa diviene la stringa corrente.

3. Se r non è terminale, torna al primo passo, altrimenti termina producendo in output la stringa corrente.

Esempio 4.1: algoritmo di Markov per il successore di un numero unario

Dato un numero naturale n codificato in unario mediante la stringa 1^{n+1}, un semplice algoritmo di Markov che produce la codifica del suo successore è costituito dalla sola regola terminale $\lambda \Rightarrow 1$: tale regola è sempre applicabile e inserisce un simbolo 1 in testa alla stringa di input. Ad esempio, partendo dalla stringa 111 che rappresenta il numero naturale 2, l'esecuzione dell'algoritmo di Markov produce, dopo aver applicato l'unica regola, la stringa 1111 che rappresenta il numero naturale 3.

Esempio 4.2: algoritmo di Markov per la somma di due numeri unari

Dati due numeri naturali n e m codificati in unario mediante le due stringhe 1^{n+1} e 1^{m+1} separate da un simbolo 0, un semplice algoritmo di Markov che produce la codifica unaria del numero naturale $n+m$ è costituito dalla sola regola terminale $10 \Rightarrow \lambda$: tale regola elimina il separatore tra i due numeri insieme al simbolo 1 di troppo. Ad esempio, partendo dalla stringa 11110111 che rappresenta i due numeri naturali 3 e 2, l'esecuzione dell'algoritmo di Markov produce, dopo aver applicato l'unica regola, la stringa 111111 che rappresenta il numero naturale 5.

Il prossimo esempio illustra come l'ordine con cui le regole sono presenti nell'insieme R sia determinante per il corretto funzionamento dell'algoritmo. In particolare, osserviamo che se R contiene due regole $x_1 \to y_1$ e $x_2 \to y_2$ (eventualmente terminali) tali che x_1 è un prefisso di x_2, allora la regola $x_2 \to y_2$ deve precedere la regola $x_1 \to y_1$, perché altrimenti essa non verrebbe mai selezionata durante l'esecuzione dell'algoritmo: in particolare, se R include una regola $\lambda \to y$ (eventualmente terminale), questa deve essere la sua ultima regola.

Esempio 4.3: algoritmo di Markov per il prodotto di due numeri unari

Dati due numeri naturali n e m codificati in unario mediante le due stringhe 1^{n+1} e 1^{m+1} separate da un simbolo 0, un algoritmo di Markov che produce la codifica unaria del loro prodotto può operare nel modo seguente. Dopo aver cancellato un simbolo 1 da entrambe le sequenze, l'algoritmo, per ogni simbolo 1 della prima sequenza, esegue una copia della seconda sequenza in fondo alla stringa corrente e, infine, cancella la seconda sequenza: l'insieme di regole che implementa tale strategia è la seguente.

1	$\pi 1 \to \pi_0$	10	$\alpha 0 \to \tau$	19	$1\delta_1 \to \delta_1 1$
2	$\pi_0 1 \to 1\pi_0$	11	$\alpha 1 \to \beta$	20	$2\delta_1 \to 2\gamma$
3	$\pi_0 0 \to 0\rho$	12	$\beta 1 \to 1\beta$	21	$2\mu \to \mu 1$
4	$\rho 1 \to \rho_0$	13	$\beta 0 \to 0\gamma$	22	$0\mu \to \nu 0$
5	$\rho_0 1 \to 1\rho_0$	14	$\gamma 1 \to 2\delta$	23	$1\nu \to \nu 1$
6	$\rho_0 \to \sigma =$	15	$\delta 1 \to 1\delta$	24	$\nu \to \alpha$
7	$1\sigma \to \sigma 1$	16	$\delta = \to \delta_0 = 1$	25	$\tau 1 \to \tau$
8	$0\sigma \to \sigma 0$	17	$1\delta_0 \to \delta_1 1$	26	$\tau = \Rightarrow 1$
9	$\sigma \to \alpha$	18	$2\delta_0 \to \mu 1$	27	$\lambda \to \pi$

Le prime nove regole, insieme alla regola 27, realizzano la cancellazione di un simbolo 1 nelle due sequenze e l'inserimento di un separatore (ovvero il simbolo =) al termine della seconda sequenza. Le successive quindici regole (ovvero, quelle dalla 10 alla 24) eseguono, per ogni simbolo 1 della prima sequenza, una copia della seconda sequenza immediatamente dopo il simbolo =. Infine, le regole 25, 26 e 27 consentono di cancellare la seconda sequenza e il simbolo =. Ad esempio, partendo dalla stringa 1110111 che rappresenta i due numeri naturali 2 e 2, l'esecuzione dell'algoritmo di Markov passa attraverso le stringhe mostrate in Figura 4.1, dove nelle colonne a sinistra di quelle contenenti la stringa corrente sono mostrati i numeri delle regole che sono applicate: come si può vedere la stringa prodotta dall'esecuzione dell'algoritmo è 11111 che rappresenta il numero naturale 4.

Una funzione $f : \Sigma^* \to \Sigma^*$ è detta essere **Markov-calcolabile** se esiste un algoritmo di Markov che, partendo da una stringa $x \in \Sigma^*$, produce in output $f(x)$.

Figura 4.1: esecuzione dell'algoritmo di Markov per il prodotto.

Regola	Stringa	Regola	Stringa	Regola	Stringa	Regola	Stringa
	1110111	11	β1011=	11	β011=11	10	τ11=1111
27	π1110111	12	1β011=	13	0γ11=11	25	τ1=1111
1	$\pi_0$110111	13	10γ11=	14	02δ1=11	25	τ=1111
2	1$\pi_0$10111	14	102δ1=	15	021δ=11	26	11111
2	11$\pi_0$0111	15	1021δ=	16	021δ_0=111		
3	110ρ111	16	1021δ_0=1	17	02$\delta_1$1=111		
4	110$\rho_0$11	17	102$\delta_1$1=1	20	02γ1=111		
5	1101$\rho_0$1	20	102γ1=1	14	022δ=111		
5	11011ρ_0	14	1022δ=1	16	022δ_0=1111		
6	11011σ=	16	1022δ_0=11	18	02μ1=1111		
7	1101σ1=	18	102μ1=11	21	0μ11=1111		
7	110σ11=	21	10μ11=11	22	ν011=1111		
8	11σ011=	22	1ν011=11	24	α011=1111		
7	1σ1011=	23	ν1011=11				
7	σ11011=	24	α1011=11				
9	α11011=						

Teorema 4.1

Se una funzione $f : \Sigma^* \to \Sigma^*$ è Markov-calcolabile, allora f è calcolabile.

Dimostrazione. Sia R l'insieme delle regole di riscrittura dell'algoritmo di Markov che calcola f, con $n = |R|$, e sia $a_i \to b_i$ o $a_i \Rightarrow b_i$ l'i-esima regola di R, per i compreso tra 1 e n. La macchina di Turing T che calcola f include uno stato $q_0^{[i]}$, per ogni i con $1 \leqslant i \leqslant n$: in particolare, $q_0^{[1]}$ è lo stato iniziale di T. Trovandosi nello stato $q_0^{[i]}$, T cerca la prima occorrenza di a_i: se non la trova, scorre il nastro verso sinistra fino a incontrare un simbolo $\square$ e, se $i < n$, si sposta a destra e passa nello stato $q_0^{[i+1]}$, altrimenti (ovvero $i = n$), si sposta a destra e passa nello stato finale q_1.[1] Se, invece, T trova un'occorrenza di a_i, allora la sostituisce con la sequenza b_i, eventualmente operando il necessario spostamento del contenuto del nastro verso sinistra (se $|a_i| > |b_i|$) oppure verso destra (se $|a_i| < |b_i|$). Se l'i-esima regola è terminale, allora T passa nello stato q_2 altrimenti passa nello stato q_3: in entrambi i casi, scorre il nastro verso sinistra fino a trovare un simbolo $\square$. Infine, se si trova nello stato q_2, T si sposta a destra e passa nello stato finale q_1, altrimenti (ovvero si trova nello stato q_3) si sposta

[1] Senza perdita di generalità, possiamo assumere che il simbolo $\square$ non faccia parte dell'alfabeto Σ.

a destra e passa nello stato $q_0^{[1]}$. È facile verificare che, per ogni stringa $x \in \Sigma^*$ per cui $f(x)$ è definita, la macchina di Turing T con input x termina con il nastro contenente la stessa stringa prodotta dall'esecuzione dell'algoritmo di Markov (ovvero, la stringa $f(x)$) e con la testina posizionata sul suo primo simbolo: pertanto, f è una funzione calcolabile e il teorema risulta essere dimostrato. ◇

Abbiamo dunque visto che gli algoritmi di Markov non sono un modello più potente delle macchine di Turing e, quindi, che tutto ciò che non può essere calcolato da una macchina di Turing non può essere calcolato da un algoritmo di Markov. In particolare, tutti i linguaggi non decidibili che abbiamo mostrato nel precedente capitolo non sono decidibili facendo uso degli algoritmi di Markov.

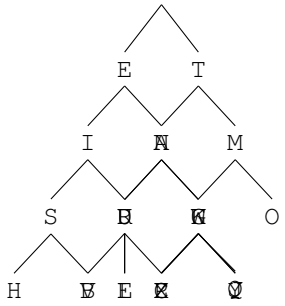
4.2 Macchine di Post

Una **macchina di Post** è simile a una macchina di Turing con la differenza che l'accesso al nastro avviene in modalità FIFO (*First In First Out*), come se quest'ultimo fosse una coda: in una singola transizione, una macchina di Post legge e cancella il simbolo che si trova in cima alla coda e inserisce un simbolo in coda alla coda. Per questo motivo, una transizione da uno stato q a uno stato p contiene come etichetta una lista di coppie di simboli (σ, τ) tale che non siano presenti due coppie con lo stesso primo simbolo: il simbolo σ rappresenta il simbolo in testa alla coda, che deve essere cancellato, mentre il simbolo τ rappresenta il simbolo da appendere in coda alla coda. Se, ad esempio, la macchina di Post si trova nello stato q e la coda contiene la stringa σx, l'esecuzione della transizione porta la macchina nello stato p con la coda contenente la stringa $x\tau$.

L'etichetta di una transizione da uno stato q a uno stato p può anche essere costituita dalla sola coppia del tipo (λ, τ), nel qual caso la transizione viene eseguita indipendentemente dal simbolo in testa alla coda (il quale non viene cancellato): se, ad esempio, la macchina di Post si trova nello stato q e la coda contiene la stringa x, l'esecuzione di una tale transizione porta la macchina nello stato p con la coda contenente la stringa $x\tau$. Infine, l'etichetta di una transizione può contenere una o più coppie del tipo (σ, λ), nel qual caso la transizione viene eseguita senza che alcun simbolo venga appeso in coda alla coda: se, ad esempio, la macchina di Post si trova nello stato q e la coda contiene la stringa σx, l'esecuzione di una tale transizione porta la macchina nello stato p con la coda contenente la stringa x.

Inizialmente, il nastro di una macchina di Post contiene la stringa di input $x = x_1 \cdots x_n$, dove x_1 denota la testa della coda e x_n ne denota la coda: la computazione ha termine quando nessuna transizione può essere applicata e l'output prodotto coincide con il contenuto della coda al termine della computazione.

Figura 4.2: il codice Morse



Per semplicità di programmazione, nel seguito assumeremo che una macchina di Post possa leggere e cancellare una sequenza di simboli (anziché un solo simbolo) che si trovano in cima alla coda e possa inserire una sequenza di simboli (anziché un solo simbolo) in coda alla coda: chiaramente, una tale macchina può essere sempre trasformata in una del tipo originale aggiungendo un opportuno insieme di stati la cui definizione dipende dai simboli delle due sequenze.

Esempio 4.4: macchina di Post per il codice Morse

Il codice Morse consente di codificare una sequenza di lettere alfabetiche mediante una sequenza di punti (simboli $\cdot$) e linee (simboli $-$): tale codice è definito dall'albero binario mostrato nella Figura 4.2, in cui ogni diramazione a sinistra corrisponde a un punto, mentre ogni diramazione a destra corrisponde a una linea (ad esempio, la codifica della stringa `SOS` è la ben nota sequenza $\cdots---\cdots$). Osserviamo che il codice Morse non garantisce la decodifica univoca di una sequenza di punti e linee, in quanto il codice di alcune lettere sono prefisso del codice di altre lettere: ad esempio, la sequenza $\cdots---\cdots$ potrebbe anche corrispondere alla stringa `EIOIE`. Perciò, il codice prevede che al termine della codifica di ogni lettera sia introdotto uno speciale simbolo di separazione (assumiamo che tale simbolo sia `#`). Una macchina di Post che, data una sequenza di lettere alfabetiche, produca in output la sua codifica Morse (inclusi i simboli di separazione) è costituita da un solo stato q e da un'unica transizione da q in se stesso la cui etichetta contiene tutte le coppie del tipo $(\tau,\gamma\#)$ dove τ è una lettera alfabetica e γ è la sua corrispondente codifica Morse: ad esempio, con input la stringa `SOS`, tale macchina passa attraverso le seguenti configurazioni: `OS`$\cdots\#$, `S`$\cdots\#---\#$ e $\cdots\#---\#\cdots\#$.

La macchina di Post dell'esempio precedente è risultata essere molto semplice da realizzare, in quanto il compito richiesto poteva naturalmente essere svolto mediante

una scansione da sinistra verso destra del contenuto iniziale del nastro. Spesso, però, l'elaborazione da parte, ad esempio, di una macchina di Turing implica lo spostamento della testina sia a destra che a sinistra: il prossimo esempio mostra come ciò possa ugualmente essere realizzato mediante una macchina di Post.

Esempio 4.5: macchina di Post per la duplicazione di una stringa binaria

Definiamo una macchina di Post P che, data in input una stringa binaria x, produca in output la stringa binaria xx. Osserviamo che tale compito è più complesso di quello di voler produrre la stringa ottenuta raddoppiando ciascun simbolo di x: in quest'ultimo caso, infatti, una scansione da sinistra verso destra sarebbe sufficiente a ottenere il risultato desiderato. La macchina di Post P, per prima cosa, racchiude la stringa x tra due copie della sequenza `#@`: ciò può essere fatto appendendo anzitutto tali due copie in coda alla coda (ottenendo la sequenza x`#@#@`), spostando la stringa x in coda alla coda (ottenendo la sequenza `#@#@`x) e, infine, spostando la prima copia di `#@` in coda alla coda (ottenendo la sequenza `#@`x`#@`). A questo punto, il simbolo `@` servirà per tenere traccia di quale parte della stringa x è già stata eseguita la copia, mentre il simbolo `#` servirà da delimitatore indicando l'inizio di ciascuna delle due copie. In particolare, P ha tre stati p, p_0 e p_1 e le seguenti transizioni.

- Una transizione da p (rispettivamente, p_0 e p_1) in se stesso la cui etichetta contiene tutte le coppie del tipo (σ, σ) con $\sigma \neq$ `@` (questa transizione consente di spostare in coda alla coda la parte di stringa che precede il prossimo simbolo `@`).
- Una transizione da p a p_0 la cui etichetta contiene la sola coppia (`@0`, `0@`) e una transizione da p a p_1 la cui etichetta contiene la sola coppia (`@1`, `1@`).
- Una transizione da p_0 a p la cui etichetta contiene la sola coppia (`@`, `0@`) e una transizione da p_1 a p la cui etichetta contiene la sola coppia (`@`, `1@`).

Partendo, ad esempio, dalla sequenza `#@011#@`, la macchina P passa attraverso le seguenti configurazioni: `@011#@#`, `11#@#0@`, `@#0@11#` e `#0@11#0@`. Come si può vedere, il risultato ottenuto è quello di aver copiato il primo simbolo della stringa `011`, posizionando le due copie del simbolo `@` immediatamente alla destra delle due copie di tale simbolo. Proseguendo nell'esecuzione, la macchina P arriva a produrre la sequenza `@#011@#011`: per giungere al risultato desiderato (ovvero la stringa `011011`) è sufficiente a questo punto cancellare le due copie della stringa `@#`, attraverso una semplice scansione da sinistra verso destra. In particolare, P cancella anzitutto la copia a sinistra della stringa `@#` (ottenendo la stringa `011@#011`), sposta la prima copia della stringa `011` in fondo alla coda (ottenendo la stringa `@#011011`), e, infine, cancella la copia rimasta della stringa `@#` (ottenendo la stringa `011011`).

Dato un alfabeto Σ, una funzione $f : \Sigma^* \to \Sigma^*$ è detta essere **Post-calcolabile** se esiste una macchina di Post che, partendo da una stringa $x \in \Sigma^*$, produce in output la stringa $f(x) \in \Sigma^*$. Il prossimo risultato afferma che quanto abbiamo visto nel caso delle funzioni di calcolo della codifica Morse e di duplicazione di una stringa binaria,

può in realtà essere fatto nel caso di una qualunque funzione calcolabile: in altre parole, le macchine di Post non sono meno potenti delle macchine di Turing.

Teorema 4.2

Se $f : \Sigma^* \to \Sigma^*$ è una funzione calcolabile, allora f è Post-calcolabile.

Dimostrazione. Sia T una macchina di Turing che calcola f: senza perdita di generalità assumiamo che il nastro di T sia semi-infinito, che la testina di T possa solo muoversi a sinistra e a destra e che l'alfabeto di lavoro Γ di T non includa i simboli $\#$ e $@$ (ricordiamo che a una macchina con nastro semi-infinito non è concesso superare l'estremo sinistro del nastro stesso). L'idea della dimostrazione consiste nel rappresentare una configurazione di T mediante il contenuto della coda della macchina di Post P, in modo che il simbolo in testa alla coda sia sempre il simbolo attualmente letto da T. In particolare, il simbolo $\#$ sarà usato per indicare l'estremità destra della parte di nastro utilizzata da T, mentre il simbolo $@$ sarà sempre posizionato alla sinistra del simbolo in coda alla coda, ovvero del simbolo alla sinistra di quello attualmente letto: quindi, in generale, la configurazione $x\eta\sigma^q\tau\gamma y$, con $x, y \in \Gamma^*$ e $\eta, \sigma, \tau, \gamma \in \Gamma$, sarà rappresentata in P dalla coda contenente la seguente stringa $\tau\gamma y\#x\eta@\sigma$. Se T dalla configurazione $x\eta\sigma^q\tau\gamma y$ passa nella configurazione $x\eta\sigma\delta^p\gamma y$ con $\delta \in \Gamma$, allora P deve passare dallo stato q con la coda contenente la stringa $\tau\gamma y\#x\eta@\sigma$ allo stato p con la coda contenente la stringa $\gamma y\#x\eta\sigma@\delta$. A tale scopo, P per prima cosa può produrre a partire da $\tau\gamma y\#x\eta@\sigma$ la stringa $\gamma y\#x\eta@\sigma\delta$; quindi, può spostare i simboli in testa alla coda mettendoli in coda alla coda fino ad arrivare alla stringa $@\sigma\delta\gamma y\#x\eta$; a questo punto, può invertire i simboli $@$ e σ ottenendo la stringa $\delta\gamma y\#x\eta\sigma@$; infine, P può spostare il simbolo in testa alla coda mettendolo in coda alla coda producendo così la stringa desiderata, ovvero $\gamma y\#x\eta\sigma@\delta$. Se, invece, T dalla configurazione $x\eta\sigma^q\tau\gamma y$ passa nella configurazione $x\eta^p\sigma\delta\gamma y$, allora P deve passare dallo stato q con la coda contenente la stringa $\tau\gamma y\#x\eta@\sigma$ allo stato p con la coda contenente la stringa $\sigma\delta\gamma y\#x@\eta$. A tale scopo, P per prima cosa può produrre a partire da $\tau\gamma y\#x\eta@\sigma$ la stringa $\delta\gamma y\#x\eta@\sigma$; quindi, può spostare i simboli in testa alla coda mettendoli in coda alla coda fino ad arrivare alla stringa $\eta@\sigma\delta\gamma y\#x$; a questo punto, può invertire i simboli η e $@$ producendo così la stringa desiderata, ovvero $\sigma\delta\gamma y\#x@\eta$. Rimane da considerare il caso particolare in cui la testina di T sia posizionata sul simbolo all'estremità destra della parte di nastro attualmente utilizzata e debba eseguire un movimento a destra (per tale motivo, abbiamo introdotto il simbolo $\#$). Se T si trova nella configurazione $x\eta\sigma^q\tau$, con $x \in \Gamma^*$ e $\eta, \sigma, \tau \in \Gamma$ (la quale è rappresentata nella coda di P dalla stringa $\tau\#x\eta@\sigma$) ed esegue un movimento a destra passando nella configurazione $x\eta\delta\sigma^p$, allora P dovrà passare dallo stato q allo stato p con la coda contenente la stringa $\square\#x\eta\sigma@\delta$. A tale scopo, P per prima cosa può produrre a partire da $\tau\#x\eta@\sigma$ la stringa $\#x\eta@\sigma\delta$; quindi, accorgendosi di essere giunta all'estremità

destra del nastro, P produce la stringa $x\eta@\sigma\delta\square\#$; in modo simile a quanto visto in precedenza, può quindi invertire i simboli @ e σ ottenendo la stringa $\delta\square\#x\eta\sigma@$; infine, può spostare il simbolo in testa alla coda mettendolo in coda alla coda fino ad arrivare alla stringa desiderata, ovvero $\square\#x\eta\sigma@\delta$. ◇

Osserviamo che il teorema precedente mostra come una coda sia un modello di calcolo decisamente più potente di una pila. Infatti, sebbene in un esercizio del secondo capitolo si affermi che una macchina costituita da due pile ha almeno lo stesso potere computazionale di una macchina di Turing, è ben noto che una macchina costituita da una sola pila è un modello di calcolo meno potente delle macchine di Turing: in particolare, è possibile dimostrare che il linguaggio costituito dalle stringhe del tipo $0^n1^n2^n$ non può essere deciso da una macchina con una sola pila, mentre è facile definire una macchina di Turing che decida tale linguaggio (torneremo su questo argomento nella prossima parte di queste dispense).

Prima di procedere con un modello di calcolo più funzionale, dimostriamo che i tre modelli di calcolo sinora considerati sono, in realtà, tra di loro equivalenti. A tale scopo, è sufficiente verificare che le macchine di Post possono essere simulate dagli algoritmi di Markov, come afferma il prossimo risultato.

Teorema 4.3

Se $f : \Sigma^* \to \Sigma^*$ è una funzione Post-calcolabile, allora f è Markov-calcolabile.

Dimostrazione. Sia P una macchina di Post che calcola la funzione f e sia Γ il suo alfabeto di lavoro. L'idea della dimostrazione consiste nell'avere una regola di Markov per ogni transizione di P a cui seguano una lista di regole che consentano, eventualmente, di appendere un simbolo in coda alla coda. In particolare, l'algoritmo di Markov che simula P, ha come ultima regola $\lambda \to q_0$ dove q_0 è lo stato iniziale di P. Per ogni transizione di P da uno stato q a uno stato p che avvenga leggendo e cancellando un simbolo σ dalla testa della coda e scrivendo il simbolo τ in coda alla coda, aggiungiamo la regola $q\sigma \to q_\sigma$ e, per ogni $\gamma \in \Gamma$, la regola $q_\sigma\gamma \to \gamma q_\sigma$; inoltre, aggiungiamo la regola $q_\sigma \to q_{\sigma\tau}\tau$ e, per ogni $\gamma \in \Gamma$, la regola $\gamma q_{\sigma\tau} \to q_{\sigma\tau}\gamma$; infine, aggiungiamo la regola $q_{\sigma\tau} \to p$. Infine, successivamente a queste regole, aggiungiamo, per ogni stato q di P, la regola finale $q \Rightarrow \lambda$. ◇

Abbiamo dunque dimostrato che le macchine di Turing, gli algoritmi di Markov e le macchine di Post sono modelli di calcolo equivalenti: quest'affermazione è formalizzata dal seguente risultato, che è un'ovvia conseguenza dei Teoremi 4.1, 4.2 e 4.3.

Corollario 4.1

Una funzione $f : \Sigma^* \to \Sigma^*$ è calcolabile se e solo se f è Post-calcolabile se e solo se f è Markov-calcolabile.

4.3 Funzioni ricorsive

Una funzione ricorsiva è una funzione che può essere ottenuta, a partire da tre funzioni di base, applicando uno di tre possibili operatori, ovvero gli operatori di composizione, ricorsione primitiva e minimalizzazione. In particolare, una funzione $f : \mathbb{N}^n \to \mathbb{N}$ è detta essere **ricorsiva** se una delle seguenti condizioni è vera.

1. f è la funzione **successore** S, tale che, per ogni numero naturale x, $S(x) = x+1$.
2. f è la funzione **zero** Z^n, tale che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$, $Z^n(x_1, \ldots, x_n) = 0$.
3. Esiste un numero naturale k, con $1 \leqslant k \leqslant n$, per cui f è la funzione **selettore** U^n_k, tale che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$, $U^n_k(x_1, \ldots, x_n) = x_k$.
4. f è ottenuta applicando l'operatore di **composizione**, ovvero esistono m funzioni ricorsive $g_1, \ldots, g_m : \mathbb{N}^n \to \mathbb{N}$ e una funzione ricorsiva $h : \mathbb{N}^m \to \mathbb{N}$, tali che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$,
$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$
5. f è ottenuta applicando l'operatore di **ricorsione primitiva**, ovvero esiste una funzione ricorsiva $b : \mathbb{N}^n \to \mathbb{N}$ e una funzione ricorsiva $i : \mathbb{N}^{n+1} \to \mathbb{N}$, tali che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$,
$$f(x_1, \ldots, x_n) = \begin{cases} b(x_1, \ldots, x_n) & \text{se } x_1 = 0, \\ i(x_1 - 1, \ldots, x_n, f(x_1 - 1, \ldots, x_n)) & \text{altrimenti} \end{cases}$$
6. f è ottenuta applicando l'operatore di **minimalizzazione**, ovvero esiste una funzione ricorsiva $p : \mathbb{N}^{n+1} \to \mathbb{N}$, tale che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$,
$$f(x_1, \ldots, x_n) = \min_{t \geqslant 0}[p(x_1, \ldots, x_n, t) = 0]$$

Esempio 4.6: funzione ricorsiva per il calcolo del predecessore

È facile dimostrare che la funzione predecessore `pre` è una funzione ricorsiva (assumendo che il predecessore di 0 sia 0 stesso). Infatti, tale funzione può essere definita nel modo seguente.
$$\texttt{pre}(x) = \begin{cases} Z^1(x) & \text{se } x = 0, \\ U^2_1(x-1, f(x-1)) & \text{altrimenti} \end{cases}$$
In questo caso, la funzione ricorsiva $b : \mathbb{N} \to \mathbb{N}$ coincide la funzione zero Z^1, mentre la funzione ricorsiva $i : \mathbb{N}^2 \to \mathbb{N}$ coincide con la funzione selettore U^2_1.

Esempio 4.7: funzione ricorsiva per il calcolo della somma

È anche facile dimostrare che la funzione `sum` di somma di due numeri naturali è una funzione ricorsiva. Infatti, tale funzione può essere definita nel modo seguente.

$$\texttt{sum}(x,y) = \begin{cases} U_2^2(x,y) & \text{se } x = 0, \\ S(U_3^3(x-1,y,f(x-1,y))) & \text{altrimenti} \end{cases}$$

In questo caso, la funzione ricorsiva $b : N^2 \to N$ coincide la funzione selettore U_2^2, mentre la funzione ricorsiva $i : N^3 \to N$ coincide con la composizione di $h = S$ e di $g_1 = U_3^3$.

Osserviamo che se `f` è ottenuta applicando solo gli operatori di composizione e di ricorsione primitiva, allora `f` è detta essere **ricorsiva primitiva**: in particolare, quindi, la funzione predecessore e la funzione di somma sono due funzioni ricorsive primitive e non è difficile dimostrare che lo stesso vale per la funzione `times` di prodotto di due numeri naturali (si veda l'Esercizio 4.3).

Esempio 4.8: funzione ricorsiva per il calcolo della radice quadrata

Per dimostrare la ricorsività della funzione $\texttt{sqrt} : N \to N$ tale che, per ogni numero naturale x, $\texttt{sqrt}(x) = \lfloor \sqrt{x} \rfloor$, mostriamo anzitutto che la funzione `dif` tale che, dati due numeri interi x e y, $\texttt{dif}(x,y) = y - x$ se $y > x$, altrimenti $\texttt{dif}(x,y) = 0$, è una funzione ricorsiva. Tale funzione può infatti essere definita nel modo seguente.

$$\texttt{dif}(x,y) = \begin{cases} U_2^2(x,y) & \text{se } x = 0, \\ \texttt{pred}(U_3^3(x-1,y,\texttt{dif}(x-1,y))) & \text{altrimenti} \end{cases}$$

In questo caso, la funzione ricorsiva $b : N^2 \to N$ coincide la funzione selettore U_2^2, mentre la funzione ricorsiva $i : N^3 \to N$ coincide con la composizione di `pre` e di $g_1 = U_3^3$. Mostriamo ora che la funzione `gte` tale che, dati due numeri interi x e y, $\texttt{gte}(x,y) = 0$ se $y > x$, altrimenti $\texttt{gte}(x,y) = 1$, è una funzione ricorsiva: tale funzione può infatti essere ottenuta applicando due volte l'operatore di composizione nel modo seguente.

$$\texttt{gte}(x,y) = \texttt{dif}(\texttt{dif}(x,y), S(Z^2(x,y)))$$

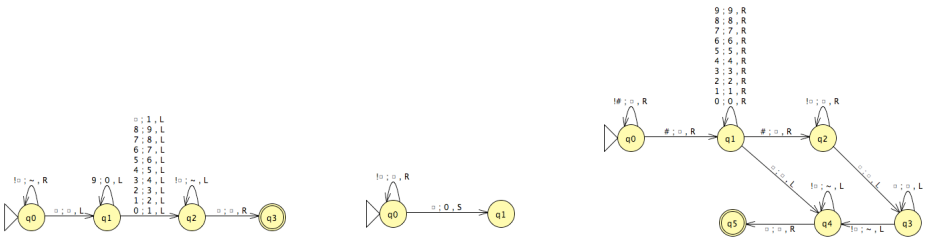
Nella prima composizione, la funzione h coincide con S e la funzione g_1 con Z^2, mentre nella seconda composizione, la funzione h coincide con `dif` e le due funzioni g_1 e g_2 coincidono, rispettivamente, con `dif` e con la funzione ottenuta dalla prima composizione. A questo punto, la funzione `sqrt` che calcola la radice quadrata di un numero naturale x può essere definita applicando più volte l'operatore di composizione e una volta quello di minimalizzazione nel modo seguente.

$$\texttt{sqrt}(x) = \texttt{pre}\left(\min_{t \geqslant 0}\left(\texttt{gte}(U_1^2(x,t), \texttt{times}(U_2^2(x,t), U_2^2(x,t))) = 0\right)\right)$$

Dimostrazioni per induzione strutturale

Dato un insieme X di oggetti definito in modo induttivo, specificando quali siano gli oggetti *elementari* e quali siano le *operazioni* con cui costruire oggetti complessi a partire da quelli elementari, una dimostrazione **per induzione strutturale** che ogni elemento di X soddisfa una determinata proprietà consiste nel dimostrare tale proprietà nel caso degli oggetti elementari e nel dimostrare che un oggetto ottenuto mediante l'applicazione di un'operazione a oggetti per i quali la proprietà è stata già verificata, soddisfa anch'esso la proprietà desiderata.

Figura 4.3: macchine di Turing successore e zero.



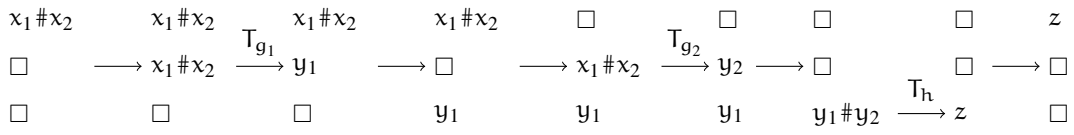
Anche se non lo mostreremo in queste dispense, è possibile provare l'esistenza di funzioni ricorsive che non siano ricorsive primitive. Il prossimo risultato, invece, mostra come le funzioni ricorsive non siano un modello più potente delle macchine di Turing e, quindi, che esistano funzioni che non sono ricorsive.

Teorema 4.4

Ogni funzione ricorsiva $f : \mathbb{N}^n \to \mathbb{N}$ è calcolabile da una macchina di Turing.

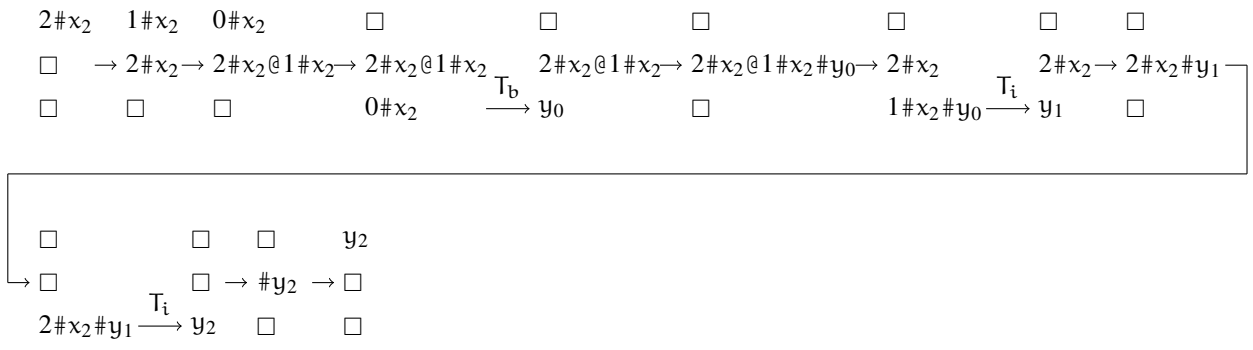
Dimostrazione. La dimostrazione procede per induzione strutturale e consiste anzitutto nel mostrare che le tre funzioni di base sono calcolabili da una macchina di Turing (nel seguito, senza perdita di generalità, assumeremo che n numeri naturali siano separati da un simbolo speciale diverso da $\square$, come, ad esempio, il simbolo $\#$). In particolare, tre macchine di Turing che implementano, rispettivamente, le funzioni S, Z^n (per ogni $n \geqslant 1$) e U_2^n (per ogni $n \geqslant 2$) sono mostrate nella parte sinistra, centrale e destra della Figura 4.3: per ogni $n \geqslant 1$ e per ogni k con $1 \leqslant k \leqslant n$, è facile modificare la terza macchina in modo da ottenerne una che calcoli la funzione U_k^n. Supponiamo, ora, che f sia ottenuta applicando l'operatore di composizione a partire dalle funzioni $h, g_1, \ldots, g_m$. Per ipotesi induttiva, esistono $m+1$ macchine di Turing $T_h, T_{g_1}, \ldots, T_{g_m}$ che calcolano, rispettivamente, le funzioni $h, g_1, \ldots, g_m$ (senza perdita di generalità, possiamo assumere che tali macchine terminino con il solo valore della funzione presente sul nastro). Possiamo allora definire una macchina di Turing

Figura 4.4: macchina di Turing per l'operatore di composizione



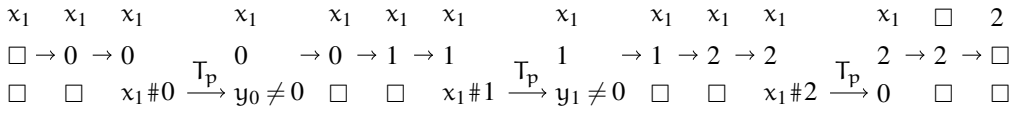
T_f con tre nastri che calcola f e che opera nel modo seguente (si veda la Figura 4.4, in cui per semplicità abbiamo assunto che $m = n = 2$). Con input n numeri naturali $x_1, \ldots, x_n$ presenti sul primo nastro, T_f li copia sul secondo nastro, esegue la macchina T_{g_1} usando il secondo nastro come suo nastro di lavoro e copia (cancellandolo) l'output prodotto da T_{g_1} sul terzo nastro: queste tre operazioni vengono eseguite per le rimanenti $m-1$ macchine $T_{g_2}, \ldots, T_{g_m}$, prestando attenzione a separare gli output prodotti mediante un simbolo $\#$ e a cancellare il contenuto del primo nastro prima dell'esecuzione della macchina T_{g_m}. Infine, T_f esegue T_h usando il terzo nastro come suo nastro di lavoro e copia (cancellandolo) l'output prodotto da T_h sul primo nastro, lasciando la testina di questo nastro sul primo simbolo diverso da $\square$. In base a quanto detto nel primo capitolo relativamente alle macchine di Turing multi-nastro, abbiamo che f è quindi calcolabile da una macchina di Turing con un singolo nastro. Supponiamo, invece, che f sia ottenuta applicando l'operatore di ricorsione primitiva a partire dalle funzioni b e i. Per ipotesi induttiva, esistono due macchine di Turing T_b e T_i che calcolano, rispettivamente, le funzioni b e i (senza perdita di generalità, possiamo assumere che tali macchine terminino con il solo valore della funzione presente sul nastro). Possiamo allora definire una macchina di Turing T_f con tre nastri che calcola f e che opera nel modo seguente (si veda la Figura 4.5, in cui per semplicità abbiamo assunto che $x_1 = n = 2$). Con input n numeri naturali $x_1, \ldots, x_n$ presenti sul primo nastro, fintanto che x_1 è maggiore di zero, T_f copia $x_1, \ldots, x_n$ sul secondo nastro (separando le varie n-tuple mediante un simbolo speciale, come, ad esempio, il simbolo @) e diminuisce di un'unità il valore di x_1 (in altre parole, T_f usa il secondo nastro come una pila su cui memorizzare gli argomenti delle successive invocazioni ricorsive). Nel momento in cui x_1 diventa pari a 0, T_f copia (cancellandolo) il contenuto del primo nastro sul terzo nastro, esegue T_b usando il terzo nastro come suo nastro di lavoro e copia (cancellandolo) l'output prodotto sul secondo nastro separandolo da ciò che lo precede mediante il simbolo $\#$. A questo punto, fintanto che il secondo nastro non contiene un solo numero naturale preceduto dal simbolo $\#$, T_f copia (cancellandoli insieme all'eventuale simbolo @ alla loro sinistra) gli $n+1$ numeri naturali presenti all'estrema destra del secondo nastro sul terzo nastro, esegue T_i usando il terzo nastro come suo nastro di lavoro e copia (cancellandolo) l'output

Figura 4.5: macchina di Turing per l'operatore di ricorsione primitiva



prodotto sul secondo nastro separandolo da ciò che lo precede mediante il simbolo #. Quando il secondo nastro contiene un solo numero naturale preceduto dal simbolo #, T_f copia (cancellandolo) tale numero sul primo nastro e lascia la testina di questo nastro sul primo simbolo diverso da □. Ancora una volta, in base a quanto detto nel primo capitolo relativamente alle macchine di Turing multi-nastro, abbiamo che f è quindi calcolabile da una macchina di Turing con un singolo nastro. Supponiamo, infine, che f sia ottenuta applicando l'operatore di minimalizzazione a partire dalla funzione p. Per ipotesi induttiva, esiste una macchina di Turing T_p che calcola la funzione p (senza perdita di generalità, possiamo assumere che tale macchina termini con il solo valore della funzione presente sul nastro). Possiamo allora definire una macchina di Turing T_f con tre nastri che calcola f e che opera nel modo seguente (si veda la Figura 4.6, in cui per semplicità abbiamo assunto che $n = 1$ e che $p(x_1, 0) \neq 0$, $p(x_1, 1) \neq 0$ e $p(x_1, 2) = 0$). Con input n numeri naturali $x_1, \ldots, x_n$ presenti sul primo nastro, T_f inizializza il contenuto del secondo nastro con il numero 0 e, fintanto che il terzo nastro contiene un numero naturale diverso da 0, cancella il contenuto del terzo nastro, aumenta di un'unità il contenuto del secondo nastro, copia il contenuto del primo nastro e, separato dal simbolo #, il contenuto del secondo nastro sul terzo nastro ed esegue T_p usando il terzo nastro come suo nastro di lavoro (chiaramente, se non esiste un valore t per cui $p(x_1, \ldots, x_n, t)$ sia uguale a 0, allora la macchina di Turing T_f non termina). Nel momento in cui il terzo nastro contiene il numero 0, T_f cancella il contenuto del primo e del terzo nastro e copia (cancellandolo) il contenuto del secondo nastro sul primo nastro, lasciando la testina di questo nastro posizionata sul primo simbolo diverso da □. Sempre in base a quanto detto nel primo capitolo relativamente alle macchine di Turing multi-nastro, abbiamo che f è quindi calcolabile da una macchina di Turing con un singolo nastro. In conclusione, abbiamo provato

Figura 4.6: macchina di Turing per l'operatore di minimalizzazione



che ogni funzione ricorsiva è calcolabile da una macchina di Turing con un singolo nastro: il teorema risulta pertanto essere dimostrato. ◇

Come nel caso degli algoritmi di Markov e delle macchine di Post, è anche possibile provare il viceversa del teorema precedente, ovvero che ogni funzione calcolabile è anche ricorsiva. La dimostrazione di tale risultato (che non includiamo in queste dispense) fa uso di una tecnica introdotta dal logico Kurt Gödel nella dimostrazione del suo famoso teorema e, per questo, chiamata *tecnica di goedilizzazione*: tale tecnica consente di mettere in corrispondenza biunivoca e in modo costruttivo l'insieme dei numeri naturali con quello delle configurazioni della computazione di una macchina di Turing, simulando così attraverso funzioni ricorsive il processo di produzione di una configurazione a partire da un'altra configurazione. Il Corollario 4.1 può dunque essere esteso nel modo seguente.

Corollario 4.2

Una funzione $f : \mathbb{N}^n \to \mathbb{N}$ è calcolabile se e solo se f è Post-calcolabile se e solo se f è Markov-calcolabile se e solo se f è ricorsiva.

4.4 Macchine RAM

La maggior parte dei programmatori sono soliti esprimere i loro algoritmi facendo uso di linguaggi di programmazione di alto livello come `C`, `Java` e `COBOL`, sapendo che un programma (sintatticamente corretto) scritto in uno qualunque di tali linguaggi può essere tradotto in un programma scritto in linguaggio macchina: questo è effettivamente il compito svolto dai compilatori. L'ultimo modello di calcolo che consideriamo in questo capitolo è un modello molto simile a un linguaggio macchina e, quindi, in grado di calcolare tutto quello che possiamo calcolare facendo uso di un linguaggio di programmazione di alto livello. Tale modello risulta essere apparentemente più potente di un linguaggio macchina, non avendo alcun vincolo sulla dimensione di una parola di memoria o di un indirizzo di una cella di memoria: in questo paragrafo, mostreremo che anche questo modello è computazionalmente equivalente

alle macchine di Turing, per cui potremo concludere che esistono funzioni che non possono essere calcolate da alcun programma scritto, ad esempio, in `Java`.

Una **Random Access Machine** (in breve, **RAM**) ha un numero potenzialmente infinito di *registri* e un *contatore di programma*: ogni registro è individuato da un *indirizzo*, ovvero da un numero naturale, e può contenere un numero naturale *arbitrariamente* grande. Nel seguito, indichiamo con `n` un numero naturale, con `(n)` il contenuto del registro il cui indirizzo è `n` e con `[n]` il contenuto del registro il cui indirizzo è contenuto nel registro il cui indirizzo è `n`. Un **programma RAM** è una sequenza ordinata di istruzioni (eventualmente dotate di un'etichetta) di uno dei seguenti tipi (nel seguito, `ox` può indicare `mx`, `(mx)` oppure `[mx]`, per un qualunque numero naturale `mx`).

- `(n) := o1` *operatore* `o2` oppure `[n] := o1` *operatore* `o2` dove *operatore* indica una delle quattro operazioni aritmetiche elementari, ovvero +, −, ⋆ e /.
- `if o1` *operatore* `o2 goto` *etichetta* dove *operatore* indica una delle quattro operazioni relazionali =, <>, <= e <.

All'avvio dell'esecuzione di un programma RAM, i primi n registri contengono gli n numeri naturali che costituiscono l'input mentre tutti gli altri registri contengono 0 e il contatore di programma è uguale a 0: a ogni passo, l'istruzione indicata dal contatore di programma viene eseguita. Se tale istruzione non è un `if` oppure è un `if` la cui condizione booleana non è verificata, il contatore di programma aumenta di 1, altrimenti il contatore di programma viene posto uguale all'indice dell'istruzione corrispondente all'etichetta dell'istruzione `if`. Il programma può anche includere una o più istruzioni `end` che terminano la sua esecuzione: in tal caso, l'output prodotto è uguale al contenuto del registro con indice 0.

Esempio 4.9: programma RAM per la verifica di sequenze palindrome

Supponiamo di voler decidere se una sequenza di 10 numeri naturali è palindroma. Il seguente programma RAM risolve tale problema scorrendo i primi 10 registri in una direzione e nell'altra e verificando che i numeri corrispondenti siano uguali.

```
        (11) := 9 + 0
 loop:  if (11) <= (10) goto yes
        if [10] <> [11] goto no
        (10) := (10) + 1
        (11) := (11) - 1
        if 0 = 0 goto loop
  yes:  (0) := 1 + 0
        end
   no:  (0) := 0 + 0
        end
```

Una funzione $f : \mathbb{N}^n \to \mathbb{N}$ è **RAM-calcolabile** se esiste un programma RAM che, data in input una sequenza $x_1, \ldots, x_n$ di n numeri naturali, termina producendo in output il valore $f(x_1, \ldots, x_n)$. Il prossimo teorema mostra come le macchine RAM non siano più potenti di quelle di Turing e che, quindi, esistano funzioni che non sono RAM-calcolabili.

Teorema 4.5

Ogni funzione $f : \mathbb{N}^n \to \mathbb{N}$ RAM-calcolabile è calcolabile da una macchina di Turing.

Dimostrazione. Definiamo una macchina di Turing T con più nastri, di cui un nastro funge da input e contiene la sequenza di n numeri naturali $x_1, \ldots, x_n$ separati dal simbolo speciale $\#$, e un nastro funge da memoria e contiene inizialmente il solo simbolo speciale $@$ (i rimanenti nastri sono nastri di lavoro, inizialmente vuoti e il cui utilizzo sarà chiarito nel corso della definizione di T). A regime il nastro di memoria contiene la seguente stringa.

$$@\#a_1 : v_1 \# a_2 : v_2 \# a_3 : v_3 \# \cdots a_m : v_m \# \square\square\square\cdots$$

In essa, $:$ è un simbolo speciale, a_i indica l'indirizzo di un registro e v_i ne indica il contenuto (assumiamo che se l'indirizzo di un registro non appare in tale sequenza, il suo contenuto sia uguale a 0). Se l'indirizzo di un registro appare più di una volta nella sequenza, allora il suo contenuto attuale è quello che appare più a destra nella sequenza stessa. All'avvio di T, il nastro di memoria viene dunque inizializzato con la seguente stringa.

$$@\#0 : x_1 \# 1 : x_2 \# 2 : x_3 \# \cdots x_{n-1} : x_n \# \square\square\square\cdots$$

Durante l'esecuzione della macchina di Turing T, il nastro di memoria viene usato per realizzare le seguenti due operazioni.

- Determinare il contenuto di un registro il cui indirizzo a è memorizzato su un nastro di lavoro: a tale scopo, T scorre verso destra il nastro di memoria fino a trovare un $\square$, per poi spostarsi verso sinistra, cercando la prima occorrenza di a che sia seguita dal simbolo speciale $:$. Se T non trova tale occorrenza, il contenuto del registro è 0, altrimenti il contenuto del registro è racchiuso tra il simbolo $:$ e il successivo simbolo $\#$: in entrambi i casi, tale contenuto viene scritto su un altro nastro di lavoro.
- Modificare il contenuto di un registro il cui indirizzo a è memorizzato su un nastro di lavoro, assegnandogli il valore v memorizzato su un altro nastro di lavoro: a tale scopo, T scorre il nastro di memoria fino a trovare un $\square$ e scrive $a : v\#$.

Sia l il numero di istruzioni del programma RAM che calcola la funzione f. Per ogni i con $1 \leqslant i \leqslant l$, l'$i$-esima istruzione del programma RAM è realizzata da una sotto-macchina T_i, la cui definizione dipende dal tipo dell'istruzione nel modo seguente (nel seguito, considereremo solo il caso in cui l'indirizzamento sia esclusivamente indiretto, in quanto questo risulta essere il caso più complesso).

- L'i-esima istruzione è `end`: in tal caso, T_i cancella il contenuto di tutti i nastri, tranne quello di memoria, legge il contenuto del registro con indirizzo 0 e lo scrive sul primo nastro, posizionando la testina di questo nastro sul primo simbolo diverso da $\square$, cancella il contenuto del nastro di memoria e entra nell'unico stato finale della macchina di Turing T.

- L'i-esima istruzione è `[n] := [m1]` *operatore* `[m2]`: in tal caso, T_i legge il contenuto a del registro `n`, legge il contenuto $b1$ del registro `m1`, legge il contenuto $o1$ del registro $b1$, legge il contenuto $b2$ del registro `m2`, legge il contenuto $o2$ del registro $b2$, calcola il valore v ottenuto eseguendo $o1$ *operatore* $o2$ e assegna al registro a il valore v.

- L'i-esima istruzione è `if [n1]` *operatore* `[n2] goto l`: in tal caso, T_i legge il contenuto $a1$ del registro `n1`, legge il contenuto $o1$ del registro $a1$, legge il contenuto $a2$ del registro `n2`, legge il contenuto $o2$ del registro $a2$. Se $o1$ *operatore* $o2$ non è vera, passa il controllo alla sotto-macchina corrispondente all'istruzione successiva, altrimenti passa il controllo alla sotto-macchina corrispondente all'istruzione con etichetta `l`.

È evidente che T termina se e solo se il programma RAM termina e che l'output prodotto da T coincide con l'output prodotto dal programma RAM: pertanto, T calcola la funzione f e, in base a quanto detto nel primo capitolo relativamente alle macchine di Turing multi-nastro, f è calcolabile e il teorema risulta essere dimostrato. ◇

Il prossimo risultato mostra come le macchine RAM e le macchine di Turing siano, in realtà, due modelli di calcolo equivalenti (tale risultato non dovrebbe sorprendere più di tanto, avendo usato ripetutamente, in queste dispense, un programma Java come JFLAP per simulare macchine di Turing): per semplicità, formuleremo tale risultato facendo riferimento solamente a funzioni che decidano l'appartenenza di una stringa su un alfabeto binario a un determinato linguaggio.

Teorema 4.6

Ogni funzione calcolabile $f : \{1,2\}^* \to \{1,2\}$ è RAM-calcolabile.

Dimostrazione. L'idea della dimostrazione consiste nell'associare a ogni cella del nastro usato dalla macchina di Turing T che calcola f un registro della macchina RAM,

mantenendo nel registro con indirizzo 0 l'indirizzo del registro corrispondente alla cella su cui è posizionata la testina (senza perdita di generalità, possiamo assumere che il nastro di T sia semi-infinito e che l'alfabeto di lavoro di T includa i soli simboli 1, 2 e □). Supponendo che gli n simboli della stringa in input siano inizialmente memorizzati nei primi n registri della macchina RAM, il programma RAM, per prima cosa, li sposta tutti di una posizione e scrive nel registro con indirizzo 0 il valore 1. Per ogni stato q di T che non sia finale, il programma RAM include un blocco di istruzioni che inizia nel modo seguente (nel seguito assumiamo che il numero 0 corrisponda al simbolo □).

```
q  if [0] = 0 goto q,0
   if [0] = 1 goto q,1
   if [0] = 2 goto q,2
```

Tale blocco di istruzioni, prosegue con tre sotto-blocchi ciascuno dei quali corrisponde all'esecuzione della transizione eseguita da T trovandosi nello stato q e leggendo uno dei tre simboli del suo alfabeto di lavoro. Supponiamo, ad esempio, che tali transizioni siano definite dalla seguente tabella.

stato	simbolo	stato	simbolo	movimento
q	□	p	□	L
q	1	q	2	R
q	2	q	1	R

Allora, il blocco di istruzioni corrispondente allo stato q prosegue nel modo seguente.

```
q,0  (0) := (0) - 1
     if 0 = 0 goto p
q,1  [0] := [0] + 1
     (0) := (0) + 1
     if 0 = 0 goto q
q,2  [0] := [0] -1
     (0) := (0) + 1
     if 0 = 0 goto q
```

Se, invece, lo stato q è uno stato finale di T, allora il programma RAM include il seguente blocco di istruzioni.

```
q  (0) := [0] + 0
   end
```

È facile verificare che il programma RAM termina se e solo se T termina e che l'output prodotto dal programma RAM coincide con l'output prodotto da T: pertanto, il programma RAM calcola la funzione f e il teorema risulta essere dimostrato. ◇

In conclusione, il Corollario 4.2 può essere ulteriormente esteso, ottenendo il seguente risultato.

Corollario 4.3

Una funzione $f : \mathbb{N}^n \to \mathbb{N}$ è calcolabile se e solo se f è Post-calcolabile se e solo se f è Markov-calcolabile se e solo se f è ricorsiva se e solo se f è RAM-calcolabile.

4.5 La tesi di Church-Turing

I risultati ottenuti nei precedenti paragrafi mostrano come modelli di calcolo alternativi alle macchine di Turing, introdotti all'incirca nello stesso periodo di quello introdotto da Turing oppure simili agli odierni calcolatori, siano in grado di calcolare esattamente lo stesso insieme di funzioni che sono calcolabili mediante una macchina di Turing. Pertanto, possiamo ragionevolmente assumere che tutti i risultati negativi ottenuti nel precedente capitolo valgano *indipendentemente* dal modello di calcolo utilizzato, mostrando la difficoltà computazionale intrinseca ad alcuni linguaggi (come, ad esempio, quello della terminazione).

Il Corollario 4.3 giustifica la **tesi di Church-Turing** già esposta nell'introduzione del corso e che può essere formulata nel modo seguente.

> È CALCOLABILE TUTTO CIÒ CHE PUÒ ESSERE CALCOLATO DA UNA MACCHINA DI TURING.

In altre parole, possiamo concludere questa parte del corso affermando che il concetto di calcolabilità secondo Turing cattura il concetto intuitivo di calcolabilità e può a tutti gli effetti essere usato in sua vece.

Esercizi

Esercizio 4.1. Dimostrare che la funzione successore, definita su numeri interi rappresentati in decimale, è Markov-calcolabile.

Esercizio 4.2. Dimostrare che la funzione successore, definita su numeri interi rappresentati in decimale, è Post-calcolabile.

Esercizio 4.3. Dimostrare che la funzione di prodotto di due numeri naturali è una funzione ricorsiva.

Esercizio 4.4. Dimostrare che la funzione $r : N \to N$ tale che, per ogni numero naturale x, $p(x)$ è uguale all'x-esimo numero primo, è una funzione ricorsiva.
Esercizio 4.5. Dimostrare che la funzione $f : N \to N$, tale che $f(n) = 1$ se e solo se n è un numero primo, è RAM-calcolabile.
Esercizio 4.6. Dimostrare che la funzione $f : N^2 \to N$, tale che $f(n, m) = 1$ se e solo se n e m sono relativamente primi (ovvero se il massimo comune divisore di n e m è uguale a 1), è RAM-calcolabile.
Esercizio 4.7. Dimostrare che la funzione $f : N^4 \to N$, tale che $f(a, b, c, n) = 1$ se e solo se $a^n + b^n = c^n$, è RAM-calcolabile.

Parte II

Teoria dei linguaggi formali

Capitolo 5

La gerarchia di Chomsky

SOMMARIO

La teoria dei linguaggi formali (ovvero la teoria matematica delle grammatiche generative) è stata sviluppata originariamente per i linguaggi naturali (come l'italiano) e solo in seguito fu scoperta la sua utilità nel progetto di compilatori. In questo capitolo, facendo riferimento alla ben nota gerarchia di Chomsky, introduciamo le nozioni di base di tale teoria e mostriamo le connessioni esistenti tra due diversi tipi di grammatiche generative e due diversi tipi di macchine di Turing.

5.1 Grammatiche generative

LE GRAMMATICHE generative furono introdotte dal linguista Noam Chomsky negli anni cinquanta con lo scopo di individuare le procedure sintattiche alla base della costruzione di frasi in linguaggio naturale. Pur essendo stato ben presto chiaro che tali grammatiche non fossero sufficienti a risolvere tale problema, esse risultarono estremamente utili nello sviluppo e nell'analisi di linguaggi di programmazione.

Intuitivamente, una grammatica generativa specifica le regole attraverso le quali sia possibile, a partire da un simbolo iniziale, produrre tutte le stringhe appartenenti a un certo linguaggio. Ad esempio, sappiamo che una frase in italiano è generalmente composta da un sintagma nominale seguito da un sintagma verbale. A sua volta, un sintagma nominale è costituito da un determinante (facoltativo) seguito da un nome e da una sequenza arbitrariamente lunga di aggettivi, mentre un sintagma verbale consiste di un verbo seguito da un sintagma nominale (non considerando, per semplicità, la possibilità che vi sia anche un sintagma preposizionale). Tutto ciò potrebbe essere formalizzato attraverso le seguenti regole di produzione (assumendo, per semplicità, che al più un aggettivo possa seguire un nome).

- $\mathtt{F} \rightarrow \mathtt{SN\ SV}$

- SN $\rightarrow$ DET N A
- SV $\rightarrow$ V SN
- DET $\rightarrow$ il | un | la | una | λ
- N $\rightarrow$ cane | mucca
- A $\rightarrow$ furioso | pigra | λ
- V $\rightarrow$ morde | guarda

Facendo uso di queste regole possiamo, ad esempio, generare la frase

il cane morde la mucca

oppure la frase

una mucca pigra guarda un cane furioso

Ovviamente, il linguaggio italiano (così come tutti i linguaggi naturali) è così complesso che le regole sopra descritte non sono certo in grado di catturare il meccanismo alla base della costruzione di frasi sintatticamente corrette. Tuttavia, quest'esempio motiva la seguente definizione.

Definizione 5.1: grammatica generativa

Una **grammatica** è una quadrupla (V, T, S, P) dove:

1. V è un insieme finito non vuoto di simboli **non terminali**, talvolta anche detti *categorie sintattiche* oppure *variabili sintattiche*;
2. T è un insieme finito non vuoto di simboli **terminali** (osserviamo che V e T devono essere insiemi disgiunti);
3. S è un simbolo **iniziale** (anche detto *simbolo di partenza* oppure *simbolo di frase*) appartenente a V;
4. P è un insieme finito di **produzioni** della forma $\alpha \rightarrow \beta$ dove α e β, dette **forme sentenziali**, sono sequenze di simboli terminali e non terminali con α contenente almeno un simbolo non terminale.

Nel seguito, parlando di grammatiche in generale, i simboli non terminali saranno rappresentati da lettere maiuscole, i terminali da lettere minuscole all'inizio dell'alfabeto, sequenze di terminali da lettere minuscole alla fine dell'alfabeto, e sequenze

miste da lettere greche. Per esempio, parleremo dei non terminali A, B e C, dei terminali a, b e c, delle sequenze di terminali w, x e y e delle forme sentenziali α, β e γ.

Intuitivamente, una produzione significa che ogni occorrenza della sequenza alla sinistra della produzione (ovvero α) può essere sostituita con la sequenza alla destra (ovvero β). In particolare, diremo che una stringa ψ è **direttamente generabile** da una stringa ϕ se $\phi = \gamma\alpha\delta$, $\psi = \gamma\beta\delta$ e $\alpha \to \beta$ è una produzione della grammatica.

Le **frasi** del linguaggio associato a una grammatica sono, dunque, generate partendo da S e applicando le produzioni fino a quando non restano solo simboli terminali. In particolare, diremo che la forma sentenziale α è generabile da S, se esiste una sequenza di forme sentenziali $S = \beta_1, \beta_2, \ldots, \beta_{n-1}, \beta_n = \alpha$ tale che, per ogni i con $1 \leqslant i < n$, β_{i+1} è direttamente generabile da β_i. L'insieme di tutte le sequenze di terminali $x \in T^*$ che sono generabili da S formano il **linguaggio generato** dalla grammatica G, il quale è denotato con $L(G)$.

Esempio 5.1: una grammatica generativa

Consideriamo la grammatica $G = (V, T, S, P)$, dove $V = \{S\}$, $T = \{a, b\}$ e P contiene le seguenti produzioni:

$$S \to aSb$$

$$S \to ab$$

È facile verificare che il linguaggio $L(G)$ coincide con l'insieme delle stringhe $x = a^n b^n$, per $n \geqslant 1$. In effetti, ogni stringa x di tale tipo può essere generata applicando $n-1$ volte la prima produzione e una volta la seconda produzione. Viceversa, possiamo mostrare, per induzione sul numero n di produzioni, che ogni sequenza di terminali x generabile da S deve produrre una sequenza di n simboli a seguita da una sequenza di n simboli b, ovvero $x = a^n b^n$. Se $n = 1$, questo è ovvio, in quanto l'unica produzione applicabile è la seconda. Supponiamo, quindi, che l'affermazione sia vera per $n > 0$ e dimostriamola per $n+1$. Poiché $n+1 > 1$, abbiamo che la prima produzione applicata nel generare x deve essere $S \to aSb$: quindi, $x = ayb$ dove y è generabile da S mediante l'applicazione di n produzioni: per ipotesi induttiva, $y = a^n b^b$, per cui $x = a^{n+1} b^{n+1}$.

5.1.1 La gerarchia di Chomsky

In base alle restrizioni che vengono messe sul tipo di produzioni che una grammatica può contenere, si ottengono classi di grammatiche diverse. In particolare, Chomsky individuò quattro tipologie di grammatiche generative, definite nel modo seguente.

Grammatiche regolari o di tipo 3 In questo caso, ciascuna produzione della grammatica deve essere del tipo $A \to aB$ oppure del tipo $A \to a$ (si veda l'Esempio 5.2).

Grammatiche libere da contesto o di tipo 2 In questo caso, ogni produzione della grammatica deve essere del tipo $A \to \alpha$ (si veda l'Esempio 5.1).

Grammatiche contestuali o di tipo 1 In tal caso, ciascuna produzione della grammatica deve essere del tipo $\alpha \to \beta$ con $|\beta| \geqslant |\alpha|$.

Grammatiche non limitate o di tipo 0 In tal caso, nessun vincolo sussiste sulla tipologia delle produzioni della grammatica.

Esempio 5.2: una grammatica regolare

Consideriamo il linguaggio L costituito da tutte e sole le stringhe binarie contenenti almeno due zeri consecutivi (eventualmente preceduti e seguiti da arbitrarie stringhe binarie non vuote). È facile verificare che tale linguaggio è generato dalla grammatica regolare $G = (V, T, S, P)$, dove $V = \{S, A, B\}$, $T = \{0, 1\}$ e P contiene le seguenti regole di produzione:

$$\begin{array}{lll} S \to 1S & A \to 0B & B \to 1B \\ S \to 0S & A \to 0 & B \to 0 \\ S \to 0A & B \to 0B & B \to 1 \end{array}$$

Per dimostrare che il linguaggio $L(G)$ effettivamente coincide con l'insieme delle stringhe binarie che contengono almeno due zeri consecutivi, osserviamo anzitutto che a partire da B è possibile generare tutte le possibili stringhe binarie non vuote e, quindi, che a partire da A è possibile generare tutte le possibili stringhe binarie che incominciano con un simbolo 0 (inclusa la stringa formata dal solo 0). A partire da S, invece, possiamo produrre una qualunque forma sentenziale formata da una stringa binaria non vuota che termini con uno 0 seguita dal simbolo A: pertanto, a partire da S possiamo produrre tutte e sole le stringhe binarie che contengono almeno due zeri consecutivi (uno prodotto a partire da S e preceduto da una qualunque stringa binaria, eventualmente vuota, l'altro prodotto a partire da A e seguito da una qualunque stringa binaria, eventualmente vuota).

Una grammatica di tipo i, per i compreso tra 1 e 3, è anche una grammatica di tipo $i-1$: in verità, quest'affermazione non è proprio evidente nel caso in cui i sia uguale a 2, in quanto una grammatica di tipo 2 potrebbe contenere regole del tipo $A \to \lambda$ e, quindi, la parte destra della produzione potrebbe essere più corta della parte sinistra. Vedremo però che tali regole possono essere eliminate in modo opportuno, ottenendo quindi sempre una grammatica di tipo 1. Pertanto, l'insieme dei linguaggi

generati da una grammatica di tipo i (anche detti **linguaggi di tipo** i) è contenuto nell'insieme dei linguaggi generati da una grammatica di tipo $i-1$. In questo e nei prossimi capitoli dimostreremo che queste inclusioni sono strette, nel senso che esistono linguaggi di tipo i che non sono di tipo $i+1$, per ogni $i = 0, 1, 2$. In particolare, vedremo che il linguaggio dell'Esempio 5.1 è di tipo 2 ma non di tipo 3 e che il linguaggio del prossimo esempio è di tipo 1 ma non di tipo 2. Nel seguito, diremo che un linguaggio è **regolare** se è di tipo 3, **libero da contesto** se è di tipo 2, e **contestuale** se è di tipo 1.

Esempio 5.3: una grammatica contestuale

Consideriamo il linguaggio L costituito da tutte e sole le stringhe del tipo $0^n1^n2^n$, per $n > 0$. Tale linguaggio è generato dalla grammatica contestuale $G = (V, T, S, P)$, dove $V = \{S, A, B\}$, $T = \{0, 1, 2\}$ e P contiene le seguenti regole di produzione:

$$S \to 012 \quad S \to 0A12 \quad A1 \to 1A \quad A2 \to B122 \quad 1B \to B1 \quad 0B \to 00 \quad 0B \to 00A$$

Chiaramente, mediante la prima produzione possiamo generare la stringa 012. Per generare, invece, la stringa $0^n1^n2^n$ con $n > 1$ possiamo anzitutto applicare la seconda produzione (ottenendo $0A12$), la terza produzione (per spostare il simbolo A dopo il simbolo 1), la quarta produzione (ottenendo $01B122$), la sesta produzione (per spostare il simbolo B prima dei simboli 1) e, infine, la settima produzione (se $n = 2$) oppure l'ottava produzione (ottenendo $00A1122$). Se $n > 2$, tale sequenza di produzioni può essere ripetuta altre $n-2$ volte. Ad esempio, la stringa $0^31^32^3$ può essere generata mediante la seguente sequenza di produzioni: $S \to 0A12 \to 01A2 \to 01B122 \to 0B1122 \to 00A1122 \to 001A122 \to 0011A22 \to 0011B1222 \to 001B11222 \to 00B111222 \to 000111222$. Osserviamo che questo tipo di sequenze di produzioni sono anche le uniche possibili, in quanto in ogni istante il contesto determina univocamente quale produzione può essere applicata: in effetti, ogni forma sentenziale prodotta a partire da S può contenere un solo simbolo non terminale, ovvero A o B, e il carattere che segue o precede tale simbolo non terminale determina quale produzione può essere applicata. Quindi, il linguaggio generato da G coincide con il linguaggio L.

5.2 Linguaggi di tipo 0

IL PRIMO risultato che presentiamo in questa parte delle dispense mostra come la classificazione di Chomsky rientri nell'ambito dei linguaggi semi-decidibili. A tale scopo, però, introduciamo prima una variante delle macchine di Turing che, pur essendo equivalente al modello introdotto nel primo capitolo delle dispense, ci consentirà di dimostrare più agevolmente che la classe dei linguaggi generati da grammatiche di tipo 0 coincide con quello dei linguaggi semi-decidibili.

5.2.1 Macchine di Turing non deterministiche

Una **macchina di Turing non deterministica** è una macchina di Turing a cui non viene imposto il vincolo che, dato uno stato della macchina e un simbolo letto dalla testina, la transizione da eseguire sia univocamente determinata. Quindi, il grafo delle transizioni di una macchina di Turing non deterministica può includere un arco uscente dallo stato q la cui etichetta contiene almeno due triple (σ, τ_1, m_1) e (σ, τ_2, m_2) tali che $\tau_1 \neq \tau_2 \vee m_1 \neq m_2$, oppure due o più archi distinti uscenti dallo stato q le cui etichette includano ognuna almeno una tripla con lo stesso primo simbolo.

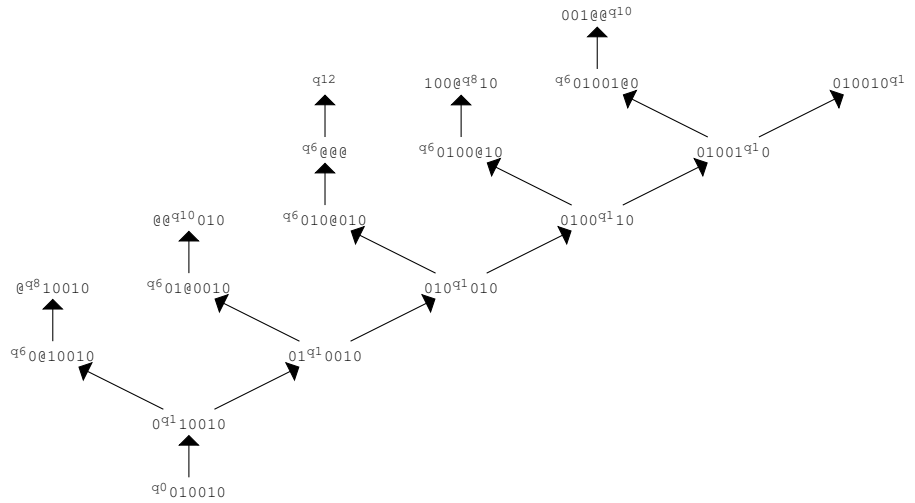
Poiché il comportamento di una macchina di Turing non deterministica, con input una stringa x, non è univocamente determinato da x stesso, dobbiamo modificare le nozioni di calcolabilità e di decidibilità fornite nel terzo capitolo. Per semplicità, nel seguito ci limiteremo a considerare macchine di Turing che decidono (oppure semi-decidono) linguaggi, anche se tutto quello che diremo si può estendere al caso di macchine di Turing che calcolano funzioni (sia pure con la specifica di diversi dettagli di carattere puramente tecnico).

Supponiamo che una macchina di Turing non deterministica si trovi nella configurazione $x^q y$ e sia σ il primo simbolo di y oppure $\square$ se $y = \lambda$ (ovvero, il simbolo σ è quello attualmente scandito dalla testina). Indichiamo con $d_{q,\sigma}$ il **grado di non determinismo dello stato** q **in corrispondenza del simbolo** σ, ovvero il numero di triple contenute in un'etichetta di un arco uscente da q, il cui primo simbolo sia σ. Allora, ognuna delle $d_{q,\sigma}$ configurazioni corrispondenti all'applicazione di tali triple può essere la configurazione prodotta da $x^q y$. Data una macchina di Turing non deterministica T, il **grado di non determinismo** r_T di T denota il massimo numero di scelte non deterministiche disponibili a partire da una qualunque configurazione. Più formalmente, $r_T = \max\{d_{q,\sigma} : q \in Q \wedge \sigma \in \Sigma\}$, dove Q è l'insieme degli stati non finali di T e Σ è il suo alfabeto di lavoro.

Chiaramente, la computazione di una macchina di Turing T non è più rappresentabile mediante una sequenza di configurazioni. Tuttavia, essa può essere vista come un albero, detto **albero delle computazioni**, i cui nodi corrispondono alle configurazioni di T e i cui archi corrispondono alla produzione di una configurazione da parte di un'altra configurazione. Per ciascun nodo dell'albero, il numero massimo dei suoi figli è deteminato dal grado di non determinismo di T.

Ciascuno dei cammini (finiti o infiniti) dell'albero che parte dalla sua radice (ovvero, dalla configurazione iniziale) è detto essere un **cammino di computazione**. Un input x è **accettato** da una macchina di Turing non deterministica T se l'albero delle computazioni corrispondente a x include almeno un **cammino di computazione accettante**, ovvero un cammino di computazione che termini in una configurazione finale. L'insieme delle stringhe accettate da T è detto essere il linguaggio **accettato** da T ed è indicato con $L(T)$.

Figura 5.1: un albero delle computazioni di una macchina non deterministica.



Esempio 5.4: una macchina di Turing non deterministica

Definiamo una macchina di Turing T non deterministica per cui $L(T) = \{xx : x \in \{0,1\}^*\}$. Tale macchina opera nel modo seguente. Sposta la testina di una posizione a destra: se il simbolo letto non è un □, *non deterministicamente* sceglie di ripetere quest'operazione oppure di proseguire spostando il contenuto del nastro a partire dalla cella scandita di una posizione a destra, inserendo un simbolo `@`, e posizionando la testina sul primo simbolo a sinistra diverso dal simbolo □. Successivamente, T verifica se le due stringhe presenti sul nastro (separate da un simbolo `@`) sono uguali: in tal caso, termina nello stato finale. Supponendo che la stringa in input sia `010010`, l'albero delle computazioni corrispondente è mostrato nella Figura 5.1, in cui per brevità la maggior parte dei cammini deterministici è stata contratta in un unico arco e in cui lo stato finale di T è lo stato `q12`. In questo caso, esiste un cammino accettante (il terzo partendo da sinistra), per cui possiamo concludere che la stringa `010010` è accettata da T: in effetti, tale stringa appartiene al linguaggio L.

Equivalenza tra macchine deterministiche e non deterministiche

Una macchina deterministica può chiaramente essere interpretata come una macchina non deterministica con grado di non determinismo pari a 1. Pertanto, tutto quello che è calcolabile da una macchina di Turing deterministica lo è anche da una non deter-

ministica. Il prossimo teorema mostra che l'uso del non determinismo non aumenta il potere computazionale del modello di calcolo delle macchine di Turing.

Teorema 5.1

Sia T una macchina di Turing non deterministica. Allora, esiste una macchina di Turing deterministica T' tale che $L(T) = L(T')$.

Dimostrazione. L'idea della dimostrazione consiste nel costruire una macchina di Turing T' deterministica che, per ogni stringa x, esegua una visita dell'albero delle computazioni di T con input x (in base a quanto detto nel primo capitolo relativamente alle macchine di Turing multi-nastro, senza perdita di generalità possiamo assumere che T' abbia a disposizione due nastri). Nel momento in cui T' incontra una configurazione finale di T, T' termina anch'essa in uno stato finale. In caso ciò non accada, se la visita dell'albero si conclude (ovvero, ogni cammino di computazione termina in una configurazione non finale), T' termina in una configurazione non finale, altrimenti T' non termina. Per poter realizzare la visita dell'albero delle computazioni in modo da essere sicuri che se un cammino accettante esiste, allora tale cammino viene, prima o poi, esplorato interamente, non possiamo fare uso della tecnica di visita in profondità, in quanto con tale tecnica la visita rischierebbe di rimanere "incastrata" in un cammino di computazione che non termina. Dobbiamo invece visitare l'albero in ampiezza o per livelli: la realizzazione di una tale visita può essere ottenuta utilizzando uno dei due nastri come se fosse una coda. In particolare, per ogni input x, T' esegue la visita in ampiezza dell'albero delle computazioni di T con input x, facendo uso dei due nastri nel modo seguente.

- Il primo nastro viene utilizzato come una coda in cui le configurazioni di T (codificate in modo opportuno) vengono inserite, man mano che sono generate, e da cui le configurazioni di T sono estratte per generarne di nuove: tale nastro viene inizializzato inserendo nella coda la configurazione iniziale di T con input x.
- Il secondo nastro viene utilizzato per memorizzare la configurazione di T appena estratta dalla testa della coda e per esaminare tale configurazione in modo da generare le (al più r_T) configurazioni da essa prodotte.

Fintanto che la coda non è vuota, T' estrae la configurazione in testa alla coda e la copia sul secondo nastro: se tale configurazione è finale, allora T' termina nel suo unico stato finale. Altrimenti, ovvero se la configurazione estratta dalla coda non è finale, T' calcola le (al più r_T) configurazioni che possono essere prodotte da tale configurazione e le inserisce in coda alla coda. Se, a un certo punto, la coda si svuota, allora tutti i cammini di computazione sono stati esplorati e nessuno di essi

è terminato in una configurazione finale: in tal caso, quindi, T' può terminare in uno stato non finale. Chiaramente, T' accetta la stringa x se e solo se esiste un cammino accettante all'interno dell'albero delle computazioni di T con input x: quindi, $L(T) = L(T')$ e il teorema risulta essere dimostrato. ◇

Come vedremo nel prossimo capitolo e nella prossima parte di queste dispense, l'utilizzo del non determinismo può, invece, aumentare il potere computazionale degli automi a pila e significativamente migliorare le prestazioni temporali di una macchina di Turing, consentendo di decidere in tempo polinomiale linguaggi per i quali non è noto alcun algoritmo polinomiale deterministico.

5.2.2 Linguaggi di tipo 0 e semi-decidibilità

Il prossimo teorema mostra come un qualunque linguaggio generato da una grammatica non limitata sia semi-decidibile da una macchina di Turing non deterministica e multi-nastro e, in base a quanto detto nel primo capitolo e nel paragrafo precedente, da una macchina di Turing deterministica con un singolo nastro (osserviamo, infatti, che la dimostrazione del Teorema 5.1 può essere estesa al caso di macchine di Turing non deterministiche e multi-nastro).

Teorema 5.2

Per ogni linguaggio L di tipo 0, esiste una macchina di Turing non deterministica a due nastri che semi-decide L.

Dimostrazione. Sia $G = (V, T, S, P)$ una grammatica non limitata che genera L. Una macchina di Turing T non deterministica a due nastri che accetta tutte e sole le stringhe di L può operare nel modo seguente.

1. Inizializza il secondo nastro con il simbolo S.

2. Sia ϕ il contenuto del secondo nastro. Per ogni produzione $\alpha \to \beta$ in P, non deterministicamente applica (tante volte quanto è possibile) tale produzione a ϕ, ottenendo la stringa ψ direttamente generabile da ϕ.

3. Se il contenuto del secondo nastro è uguale a x (che si trova sul primo nastro), termina nell'unico stato finale. Altrimenti torna al secondo passo.

Evidentemente, T accetta tutte e sole le stringhe x che possono essere generate a partire da S: il teorema risulta dunque essere dimostrato. ◇

Il prossimo risultato mostra, invece, che ogni linguaggio semi-deciso da una macchina di Turing può essere generato da una grammatica non limitata: la dimostrazione

del teorema è molto simile a quella con cui abbiamo dimostrato nel terzo capitolo la non decidibilità del problema della corrispondenza di Post e a quella con cui dimostreremo, nel prossimo capitolo, l'esistenza di linguaggi NP-completi.

Teorema 5.3

Se T è una macchina di Turing che semi-decide un linguaggio L, allora L è di tipo 0.

Dimostrazione. L'idea della dimostrazione consiste nel definire la grammatica $G = (V, T, S, P)$ in modo tale che le produzioni della grammatica siano in grado di "simulare" la computazione di T con input una qualunque stringa x. In base a quanto dimostrato nel secondo capitolo, possiamo assumere, senza perdita di generalità, che l'alfabeto di lavoro di T sia $\Sigma = \{0, 1, \square\}$, che T faccia uso di un nastro semi-infinito, che i soli movimenti consentiti alla testina del nastro siano il movimento a destra e quello a sinistra, che lo stato iniziale di T sia q_0 e che l'unico stato finale sia q^*. L'insieme delle produzioni di G consentiranno di simulare il comportamento di T operando in tre fasi distinte. La prima fase consisterà nella generazione della configurazione iniziale di T con input la stringa $x = x_1 \cdots x_n$, ovvero di una forma sentenziale corrispondente alla configurazione $q_0 x_1 \cdots x_n$. La seconda fase permetterà alla grammatica di simulare passo dopo passo la computazione di T. La terza fase, in cui la grammatica entra se T ha raggiunto lo stato finale q^*, permetterà a G di ripulire il lavoro svolto producendo al termine della fase stessa la stringa x. Poiché durante la seconda fase, nel simulare la computazione di T, le produzioni della grammatica potrebbero modificare la stringa di input iniziale, sarà necessario, per poter realizzare la terza fase, far ricordare alla grammatica in qualche modo quale fosse tale stringa. A tale scopo, faremo uso di simboli non terminali formati da una coppia di simboli in Σ: il primo elemento della coppia permetterà di rappresentare sempre il contenuto del nastro all'inizio della computazione, mentre il secondo elemento consentirà di rappresentare il contenuto del nastro durante la computazione. Inoltre, l'insieme dei simboli non terminali includerà un simbolo per ogni possibile stato di T. Infine, V includerà, oltre al simbolo iniziale S, altri cinque simboli speciali B, E, I, L e R. Per realizzare la prima fase, la grammatica G contiene le produzioni necessarie a produrre una qualunque sequenza di simboli $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ e $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$, preceduta dal simbolo q_0 e seguita da un numero arbitrario di $\square$ (osserviamo, infatti, che non possiamo sapere a priori quante celle del nastro saranno utilizzate dalla computazione di T). Tali produzioni sono le seguenti.

$$S \to LBR \quad B \to B\begin{bmatrix} \square \\ \square \end{bmatrix} \quad B \to I\begin{bmatrix} \square \\ \square \end{bmatrix} \quad I \to I\begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad I \to I\begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad I \to q_0$$

Notiamo come le produzioni sopra elencate consentono di generare anche la configurazione iniziale corrispondente alla stringa di input vuota. I due simboli L e R

sono messi all'estremità sinistra e destra e verranno utilizzati nella terza e ultima fase. Le produzioni che realizzano la seconda fase dipendono, chiaramente, dal grafo delle transizioni di T. In particolare, se tale grafo include una transizione dallo stato $q \neq q^*$ allo stato p la cui etichetta include la tripla $(a, b, \mathtt{R})$, allora, per ogni $c \in \Sigma$, la grammatica include la seguente produzione.

$$q \begin{bmatrix} c \\ a \end{bmatrix} \to \begin{bmatrix} c \\ b \end{bmatrix} p$$

Invece, se il grafo delle transizioni include una transizione dallo stato $q \neq q^*$ allo stato p la cui etichetta include la tripla $(a, b, \mathtt{L})$, allora, per ogni $c, d, e \in \Sigma$, la grammatica include la produzione

$$\begin{bmatrix} d \\ e \end{bmatrix} q \begin{bmatrix} c \\ a \end{bmatrix} \to p \begin{bmatrix} d \\ e \end{bmatrix} \begin{bmatrix} c \\ b \end{bmatrix}$$

La grammatica è ora in grado di simulare la computazione di T fino a quando non viene raggiunto lo stato finale q^* (nel caso in cui ciò non avvenga, le produzioni non consentiranno di produrre alcuna stringa di simboli terminali). In particolare, se T raggiunge lo stato q^*, allora le produzioni della grammatica incluse in precedenza consentono di generare una sequenza del seguente tipo.

$$L \begin{bmatrix} a_1 \\ b_1 \end{bmatrix} \cdots \begin{bmatrix} a_k \\ b_k \end{bmatrix} q^* \begin{bmatrix} a_{k+1} \\ b_{k+1} \end{bmatrix} \cdots \begin{bmatrix} a_h \\ b_h \end{bmatrix} R \qquad \text{con } h > n$$

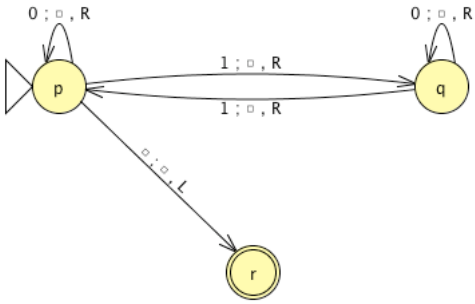
In tale sequenza, per i compreso tra 1 e n, $a_i = x_i$ e, per i compreso tra $n+1$ e h, $a_i = \square$. Per ripulire il lavoro svolto e produrre, infine, la stringa in input, possiamo consentire alla grammatica di spostare il simbolo dello stato all'estremità destra e, quindi, di attraversare l'intera forma sentenziale da destra verso sinistra, estraendo ogni simbolo diverso da $\square$ che appare come primo simbolo di uno non terminale. Questa terza fase si realizza includendo nella grammatica, per ogni $a, b \in \Sigma$, le seguenti produzioni.

$$q^* \begin{bmatrix} a \\ b \end{bmatrix} \to \begin{bmatrix} a \\ b \end{bmatrix} q^* \quad q^* R \to E \qquad \begin{bmatrix} \square \\ b \end{bmatrix} E \to E$$

$$\begin{bmatrix} 0 \\ b \end{bmatrix} E \to E0 \qquad \begin{bmatrix} 1 \\ b \end{bmatrix} E \to E1 \quad LE \to \lambda$$

È facile verificare che la grammatica così definita può generare tutte e sole le stringhe accettate da T e il teorema risulta dunque essere dimostrato. ◇

Figura 5.2: una macchina di Turing da simulare con una grammatica.



Un esempio

Consideriamo la macchina di Turing mostrata nella Figura 5.2 (in cui $q^* = r$), la quale decide il linguaggio formato da tutte le stringhe binarie contenenti un numero pari di simboli `1`: osserviamo che tale linguaggio è generabile da una grammatica regolare (si veda l'Esercizio 5.1) e, quindi, a maggior ragione da una di tipo 0. Supponiamo che la stringa di input sia $x = \texttt{0101}$. Utilizzando le regole di produzione relative alla prima fase, la grammatica definita nella dimostrazione del teorema precedente consente di generare la forma sentenziale

$$\mathrm{Lp}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}\mathrm{R}$$

nel modo seguente.

$$\begin{aligned}
S &\to \mathrm{LBR} \to \mathrm{LI}\begin{bmatrix}\square\\\square\end{bmatrix}\mathrm{R} \to \mathrm{LI}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}\mathrm{R}\\
&\to \mathrm{LI}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}\mathrm{R} \to \mathrm{LI}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}\mathrm{R}\\
&\to \mathrm{LI}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}\mathrm{R} \to \mathrm{Lp}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}\mathrm{R}
\end{aligned}$$

Come si può vedere, la stringa x è stata prodotta in doppia copia: come abbiamo già detto, la simulazione della computazione avrà luogo facendo uso della copia di sotto, mentre la copia di sopra sarà utilizzata nell'eventuale terza fase per produrre la stringa x. Osserviamo inoltre che un solo simbolo non terminale $\begin{bmatrix}\square\\\square\end{bmatrix}$ è sufficiente,

in quanto la macchina che stiamo simulando termina non appena incontra il primo $\square$ alla destra dell'input. Possiamo, a questo punto, usare le regole di produzione relative alla seconda fase, che nel caso della macchina che stiamo simulando sono le seguenti, per ogni $c, d, e \in \Sigma$.

$$p\begin{bmatrix} c \\ 0 \end{bmatrix} \to \begin{bmatrix} c \\ \square \end{bmatrix} p \quad p\begin{bmatrix} c \\ 1 \end{bmatrix} \to \begin{bmatrix} c \\ \square \end{bmatrix} q \quad \begin{bmatrix} d \\ e \end{bmatrix} p \begin{bmatrix} c \\ \square \end{bmatrix} \to r \begin{bmatrix} d \\ e \end{bmatrix} \begin{bmatrix} c \\ \square \end{bmatrix}$$

$$q\begin{bmatrix} c \\ 0 \end{bmatrix} \to \begin{bmatrix} c \\ \square \end{bmatrix} q \quad q\begin{bmatrix} c \\ 1 \end{bmatrix} \to \begin{bmatrix} c \\ \square \end{bmatrix} p$$

In particolare, la produzione delle forme sentenziali può proseguire nel modo seguente.

$$\begin{aligned} & Lp\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} R \to L\begin{bmatrix} 0 \\ \square \end{bmatrix} p \begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} R \\ \to \; & L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix} q \begin{bmatrix} 0 \\ 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} R \to L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix} q \begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} R \\ \to \; & L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix} p \begin{bmatrix} \square \\ \square \end{bmatrix} R \to L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix} r \begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} R \end{aligned}$$

Essendo stato raggiunto lo stato finale, la grammatica può entrare nella terza fase della produzione di x (osserviamo che se la macchina non termina nello stato finale, allora la grammatica non è in grado di generare una sequenza di soli simboli terminali). In particolare, usando le prime due regole di produzione relative alla terza fase, possiamo proseguire la produzione delle forme sentenziali nel modo seguente.

$$\begin{aligned} & L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix} r \begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} R \to L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix} r \begin{bmatrix} \square \\ \square \end{bmatrix} R \\ \to \; & L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} rR \to L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} E \end{aligned}$$

Attraverso le successive quattro regole di produzione relative alla terza fase, la produzione può proseguire e terminare producendo x nel modo seguente.

$$\begin{aligned} & L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix} E \to L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix} E \\ \to \; & L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix}\begin{bmatrix} 0 \\ \square \end{bmatrix} E1 \to L\begin{bmatrix} 0 \\ \square \end{bmatrix}\begin{bmatrix} 1 \\ \square \end{bmatrix} E01 \\ \to \; & L\begin{bmatrix} 0 \\ \square \end{bmatrix} E101 \to LE0101 \to 0101 \end{aligned}$$

5.3 Linguaggi di tipo 1

NELLA DIMOSTRAZIONE del Teorema 5.3 abbiamo fatto uso di regole di produzione la cui parte destra è più corta di quella sinistra: in altre parole, la grammatica definita in tale dimostrazione non è di tipo 1. Questo suggerisce che le macchine di Turing siano un modello di calcolo più potente delle grammatiche contestuali e che un diverso modello vada ricercato per poter caratterizzare la classe dei linguaggi generati da tale tipo di grammatiche. Per individuare tale modello di calcolo, proviamo a analizzare nuovamente la dimostrazione del Teorema 5.2 supponendo che la grammatica di partenza sia una grammatica di tipo 1. Il fatto che in tale grammatica ogni produzione $\alpha \to \beta$ soddisfi il vincolo $|\beta| \geqslant |\alpha|$, ci consente di modificare la macchina di Turing non deterministica definita nella dimostrazione in modo che essa termini per ogni input. Questo è quanto afferma il prossimo risultato.

Teorema 5.4

Se un linguaggio L è di tipo 1 allora esiste una macchina di Turing non deterministica T a tre nastri tale che $L(T) = L$ e, per ogni input x, ogni cammino di computazione di T con input x termina.

Dimostrazione. Sia $G = (V, T, S, P)$ una grammatica dipendente dal contesto che genera L. Una macchina di Turing T non deterministica a tre nastri che termina per ogni input e che accetta tutte e sole le stringhe di L opera in modo simile alla macchina non deterministica definita nella dimostrazione del Teorema 5.2, ma ogni qualvolta cerca di applicare (in modo non deterministico) una produzione di G, verifica se il risultato dell'applicazione sia una stringa di lunghezza minore oppure uguale alla lunghezza della stringa x di input. Se così non è, allora T può terminare in uno stato non finale, in quanto siamo sicuri (in base alla proprietà delle grammatiche di tipo 1) che non sarà possibile da una stringa di lunghezza superiore a $|x|$ generare x. Rimane anche da considerare la possibilità che la forma sentenziale generata da T rimanga sempre della stessa lunghezza a causa dell'applicazione ciclica della stessa sequenza di produzioni: anche in questo caso, dobbiamo fare in modo che T termini in uno stato non finale. A tale scopo, osserviamo che il numero di possibili forme sentenziali distinte di lunghezza k, con $1 \leqslant k \leqslant |x|$, è pari a $|V \cup T|^k$. La macchina T può evitare di entrare in un ciclo senza termine, mantenendo sul terzo nastro un contatore che viene inizializzato ogni qualvolta una forma sentenziale di lunghezza k viene generata sul secondo nastro per la prima volta. Per ogni produzione che viene applicata, se la lunghezza della forma sentenziale generata non aumenta, allora il contatore viene incrementato di 1: se il suo valore supera $|V \cup T|^k$, allora siamo sicuri

che la computazione non avrà termine e, quindi, possiamo far terminare T in uno stato non finale. In conclusione, ogni cammino di computazione della macchina T termina e tutte e sole le stringhe generate da G sono accettate da almeno un cammino di computazione di T. Quindi, $L(T) = L$ e il teorema risulta essere dimostrato. ◇

La macchina T definita nella dimostrazione del teorema precedente, oltre a terminare su qualunque cammino dell'albero delle computazioni, ha un'ulteriore proprietà: le celle utilizzate da ogni cammino di computazione su ogni singolo nastro è limitato dalla lunghezza della stringa x di input. In effetti, questo è chiaro per quanto riguarda i primi due nastri, che contengono, rispettivamente, x e la forma sentenziale corrente (che abbiamo detto non può superare in lunghezza x). Il terzo nastro contiene un contatore da 1 a, al massimo, $|V \cup T|^{|x|}$: operando in aritmetica in base $|V \cup T|$, anche questo contatore richiede al più un numero di celle limitato da $|x|$. Quest'osservazione suggerisce l'introduzione del seguente modello di calcolo.

Definizione 5.2: macchine di Turing lineari

Una **macchina di Turing lineare** è una macchina di Turing non deterministica con un singolo nastro che opera esclusivamente sulla porzione di nastro contenente la stringa di input (più le due celle adiacenti).

Poiché ogni macchina di Turing multi-nastro può essere simulata da una con un singolo nastro, sembrerebbe plausibile concludere che il Teorema 5.4 implichi il fatto che ogni linguaggio di tipo 1 sia decidibile da una macchina di Turing lineare. Tuttavia, ciò non è del tutto corretto, in quanto la simulazione di una macchina di Turing multi-nastro che abbiamo mostrato nel primo capitolo usa un numero di celle del nastro singolo pari alla somma del numero di celle utilizzate su ogni singolo nastro della macchina multi-nastro. Pertanto, la macchina di Turing con un singolo nastro non sarebbe una macchina di Turing lineare. È possibile però ovviare a questo problema, ideando una diversa tecnica di simulazione di una macchina multi-nastro, che utilizzi un numero di celle pari al massimo numero di celle usate su uno dei nastri della macchina multi-nastro: questa idea è alla base della dimostrazione del prossimo risultato.

Teorema 5.5

Se un linguaggio L è di tipo 1 allora esiste una macchina di Turing lineare T tale che $L(T) = L$ e, per ogni input x, ogni cammino di computazione di T con input x termina.

Dimostrazione. La dimostrazione consiste nel realizzare una simulazione più efficiente (dal punto di vista dello spazio) della macchina di Turing non deterministica T con 3 nastri definita nella dimostrazione del Teorema 5.4 da parte di una non deterministica

con un solo nastro. A tale scopo possiamo modificare la simulazione descritta nel primo capitolo, usando un alfabeto più "ricco" che consenta di rappresentare mediante un solo simbolo il contenuto delle k celle che si trovano nella stessa posizione dei k nastri. Ad esempio, supponendo che la macchina multi-nastro abbia due nastri e che il suo alfabeto di lavoro sia $\{0, 1, \square\}$, allora l'alfabeto della macchina con un solo nastro, per rappresentare il contenuto di due celle su cui non è posizionata la testina, potrebbe contenere i seguenti simboli.

$$\left[\begin{array}{c}0\\0\end{array}\right],\left[\begin{array}{c}0\\1\end{array}\right],\left[\begin{array}{c}0\\\square\end{array}\right],\left[\begin{array}{c}1\\0\end{array}\right],\left[\begin{array}{c}1\\1\end{array}\right],\left[\begin{array}{c}1\\\square\end{array}\right],\left[\begin{array}{c}\square\\0\end{array}\right],\left[\begin{array}{c}\square\\1\end{array}\right],\left[\begin{array}{c}\square\\\square\end{array}\right]$$

Per rappresentare, invece, il contenuto di due celle di cui solo la prima è attualmente scandita dalla testina, l'alfabeto potrebbe contenere i seguenti simboli.

$$\left[\begin{array}{c}0\text{h}\\0\end{array}\right],\left[\begin{array}{c}0\text{h}\\1\end{array}\right],\left[\begin{array}{c}0\text{h}\\\square\end{array}\right],\left[\begin{array}{c}1\text{h}\\0\end{array}\right],\left[\begin{array}{c}1\text{h}\\1\end{array}\right],\left[\begin{array}{c}1\text{h}\\\square\end{array}\right],\left[\begin{array}{c}\square\text{h}\\0\end{array}\right],\left[\begin{array}{c}\square\text{h}\\1\end{array}\right],\left[\begin{array}{c}\square\text{h}\\\square\end{array}\right]$$

Per rappresentare, poi, il contenuto di due celle di cui solo la seconda è attualmente scandita dalla testina, la grammatica conterebbe i seguenti simboli.

$$\left[\begin{array}{c}0\\0\text{h}\end{array}\right],\left[\begin{array}{c}0\\1\text{h}\end{array}\right],\left[\begin{array}{c}0\\\square\text{h}\end{array}\right],\left[\begin{array}{c}1\\0\text{h}\end{array}\right],\left[\begin{array}{c}1\\1\text{h}\end{array}\right],\left[\begin{array}{c}1\\\square\text{h}\end{array}\right],\left[\begin{array}{c}\square\\0\text{h}\end{array}\right],\left[\begin{array}{c}\square\\1\text{h}\end{array}\right],\left[\begin{array}{c}\square\\\square\text{h}\end{array}\right]$$

Per rappresentare, infine, il contenuto di due celle entrambe scandite dalla testina, la grammatica potrebbe contenere i seguenti simboli.

$$\left[\begin{array}{c}0\text{h}\\0\text{h}\end{array}\right],\left[\begin{array}{c}0\text{h}\\1\text{h}\end{array}\right],\left[\begin{array}{c}0\text{h}\\\square\text{h}\end{array}\right],\left[\begin{array}{c}1\text{h}\\0\text{h}\end{array}\right],\left[\begin{array}{c}1\text{h}\\1\text{h}\end{array}\right],\left[\begin{array}{c}1\text{h}\\\square\text{h}\end{array}\right],\left[\begin{array}{c}\square\text{h}\\0\text{h}\end{array}\right],\left[\begin{array}{c}\square\text{h}\\1\text{h}\end{array}\right],\left[\begin{array}{c}\square\text{h}\\\square\text{h}\end{array}\right]$$

La raccolta delle informazioni per decidere quale transizione eseguire e la simulazione della transizione stessa può essere fatta in modo simile a quanto detto nel primo capitolo: osserviamo che, in base alle proprietà di T, per realizzare lo spostamento della testina non sarà mai necessario spostare il contenuto dell'intero nastro di una posizione a destra (come accadeva nel caso della simulazione descritta nel primo capitolo) e, quindi, la macchina così ottenuta è una macchina di Turing lineare. Il teorema risulta dunque essere dimostrato. ◇

L'ultimo risultato di questo capitolo mostra che un linguaggio deciso da una macchina di Turing lineare è generabile da una grammatica dipendente dal contesto.

Teorema 5.6

Se T è una macchina di Turing lineare, allora L(T) è di tipo 1.

Dimostrazione. La dimostrazione è molto simile a quella del Teorema 5.3, ma chiaramente non identica in quanto, come abbiamo già osservato, la grammatica definita in quella dimostrazione non è di tipo 1: in particolare, non sono di tipo 1 le produzioni della grammatica che consentono di realizzare la terza fase della simulazione. L'idea della dimostrazione consiste dunque nel definire nuovamente la grammatica G in modo tale che le produzioni della grammatica rispettino tutte il vincolo che la parte destra della produzione sia almeno lunga quanto la parte sinistra. In particolare, i simboli non terminali e l'insieme delle produzioni di G consentiranno ancora di simulare il comportamento di T operando in tre fasi distinte, che però avranno un risultato intermedio diverso da quello della dimostrazione del Teorema 5.3. La prima fase consisterà sempre nella generazione della configurazione iniziale di T con input la stringa $x = x_1 \cdots x_n$, ma attraverso la seguente forma sentenziale (in cui # indica un simbolo speciale che non appartiene all'alfabeto di lavoro di T).

$$\begin{bmatrix} \# \\ q_0 \\ x_1 \\ x_1 \end{bmatrix} \begin{bmatrix} \\ x_2 \\ x_2 \end{bmatrix} \cdots \begin{bmatrix} \\ x_{n-1} \\ x_{n-1} \end{bmatrix} \begin{bmatrix} \# \\ \\ x_n \\ x_n \end{bmatrix}$$

Durante la seconda fase la grammatica simulerà passo dopo passo la computazione di T con input x: durante la simulazione, però, non sarà consentito alla testina di spostarsi alla sinistra del simbolo non terminale più alla sinistra contenente come primo simbolo # né alla destra del simbolo non terminale più alla destra contenente come primo simbolo #. La terza e ultima fase, che viene eseguita se e solo se T ha raggiunto lo stato finale, consisterà nel ripulire il lavoro svolto producendo al termine della fase stessa la stringa x. Lasciamo al lettore il compito (relativamente facile) di definire le produzioni della grammatica G che consentano di realizzare le tre fasi (si veda l'Esercizio 5.14). ◇

Quanto esposto in questo capitolo ci consente di riassumere nella seguente tabella, la classificazione dei linguaggi in base alla loro tipologia, al tipo di grammatiche che li generano e in base al modello di calcolo corrispondente.

Tipo di linguaggio	**Tipo di produzioni**	**Modello di calcolo**
Tipo 0	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$ e $\beta \in (V \cup T)^*$	Macchina di Turing
Contestuale	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$, $\beta \in (V \cup T)(V \cup T)^*$ e $\lvert\beta\rvert \geqslant \lvert\alpha\rvert$	Macchina di Turing lineare
Libero da contesto	$A \to \beta$ con $A \in V$ e $\beta \in (V \cup T)(V \cup T)^*$	
Regolare	$A \to aB$ e $A \to a$ con $A, B \in V$ e $a \in T$	

I prossimi due capitoli ci consentiranno di completare tale tabella, determinando i modelli di calcolo corrispondenti ai linguaggi regolari e a quelli liberi da contesto.

Esercizi

Esercizio 5.1. Descrivere una grammatica regolare che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti un numero pari di simboli `1`.
Esercizio 5.2. Dimostrare che il linguaggio costituito da tutte e sole le stringhe binarie contenenti almeno due simboli `0` e almeno un simbolo `1` è regolare.
Esercizio 5.3. Dimostrare che il linguaggio costituito da tutte e sole le stringhe binarie contenenti un numero pari di simboli `0` oppure esattamente due simboli `1` è regolare.
Esercizio 5.4. Descrivere una grammatica regolare che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti almeno tre simboli `1`.
Esercizio 5.5. Descrivere una grammatica libera da contesto che generi il linguaggio costituito da tutte e sole le stringhe binarie del tipo $0^n 1 2^n$.
Esercizio 5.6. Descrivere una grammatica libera da contesto che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti lo stesso numero di `1` e di `0`.
Esercizio 5.7. Descrivere una grammatica libera da contesto che generi il linguaggio costituito da tutte e sole le stringhe binarie contenenti più simboli `1` che simboli `0`.
Esercizio 5.8. Si consideri la grammatica $G = (V, T, S, P)$ dove $V = \{S, X, Y\}$, $T = \{a, b\}$ e P contiene le seguenti produzioni: $S \to aXb$, $X \to bYa$, $X \to ba$, $Y \to aXb$ e $Y \to ab$. Determinare il linguaggio generato da G.
Esercizio 5.9. A quale classe nella gerarchia di Chomsky appartiene il linguaggio delle stringhe sull'alfabeto $\{a, b, c\}$ che contengono un numero dispari di `a`, un numero dispari di `b` e un numero pari di `c`? Giustificare la risposta.
Esercizio 5.10. Sia T una macchina di Turing non deterministica per la quale $r_T > 2$. Dimostrare che esiste una macchina di Turing non deterministica T' con $r_{T'} = 2$ tale che $L(T) = L(T')$.
Esercizio 5.11. Fornire un esempio di linguaggio che non sia di tipo 0. Dimostrare che l'esempio è corretto.
Esercizio 5.12. Descrivere una macchina di Turing lineare che decida il linguaggio costituito da tutte e sole le stringhe binarie del tipo $0^n 1^n 2^n$ con $n > 0$.
Esercizio 5.13. Applicando il Teorema 5.6 e la macchina di Turing lineare dell'esercizio precedente, derivare una grammatica contestuale per il linguaggio costituito da tutte e sole le stringhe binarie del tipo $0^n 1^n 2^n$ con $n > 0$.
Esercizio 5.14. Completare la dimostrazione del Teorema 5.6.

Capitolo 6

Linguaggi regolari

SOMMARIO

Una delle principali componenti di un compilatore è l'analizzatore lessicale, il cui compito è quello di analizzare un programma per identificarne al suo interno le unità significanti, anche dette token*. La risorsa principale nello sviluppo di un analizzatore lessicale sono le espressioni regolari, che, come vedremo in questo capitolo, corrispondono in modo biunivoco agli automi a stati finiti e alle grammatiche regolari. Al termine del capitolo, presenteremo anche il principale strumento per dimostrare che un linguaggio non è regolare, ovvero il* pumping lemma *per linguaggi regolari.*

6.1 Introduzione

UN **COMPILATORE** è un programma che traduce un programma scritto in un linguaggio ad alto livello (come JAVA) in uno scritto in linguaggio macchina. Il linguaggio macchina è il linguaggio originale del calcolatore sul quale il programma deve essere eseguito ed è letteralmente l'unico linguaggio per cui è appropriato affermare che il calcolatore lo "comprende". Agli albori del calcolo automatico, le persone programmavano in binario e scrivevano sequenze di bit, ma ben presto furono sviluppati primitivi traduttori, detti "assembler", che permettevano al programmatore di scrivere istruzioni in uno specifico codice mnemonico anzichè mediante sequenze binarie. Generalmente, un'istruzione assembler corrisponde a una singola istruzione in linguaggio macchina. Linguaggi come JAVA, invece, sono noti come linguaggi ad alto livello ed hanno la proprietà che una loro singola istruzione corrisponde a più di un'istruzione in linguaggio macchina. Il vantaggio principale dei linguaggi ad alto livello è la produttività. È stato stimato che il programmatore medio può produrre 10 linee di codice senza errori in una giornata di lavoro (il punto chiave è "senza errori": possiamo tutti scrivere enormi quantità di codice in minor tempo, ma il tempo addizionale richiesto per verificare e correggere riduce tale quadro drasticamente). Si

è anche osservato che questo dato è essenzialmente indipendente dal linguaggio di programmazione utilizzato. Poichè una tipica istruzione in linguaggio ad alto livello può corrispondere a 10 istruzioni in linguaggio assemblatore, ne segue che approssimativamente possiamo essere 10 volte più produttivi se programmiamo, ad esempio, in JAVA invece che in linguaggio assembler.

Nel seguito, il linguaggio ad alto livello che il compilatore riceve in ingresso è chiamato *linguaggio sorgente*, ed il programma in linguaggio sorgente che deve essere compilato è detto *codice sorgente*. Il particolare linguaggio macchina che viene generato è il *linguaggio oggetto*, e l'uscita del compilatore è il *codice oggetto*. Abbiamo detto che il linguaggio oggetto è un linguaggio macchina ma non è sempre detto che tale macchina obiettivo sia reale. Ad esempio, il linguaggio JAVA viene normalmente compilato nel linguaggio di una macchina virtuale, detto "byte code". In tal caso, il codice oggetto va ulteriormente compilato o interpretato prima di ottenere il linguaggio macchina. Questo può sembrare un passo seccante ma il linguaggio byte code è relativamente facile da interpretare e l'utilizzo di una macchina virtuale consente di costruire compilatori per JAVA che siano indipendenti dalla piattaforma finale su cui il programma deve essere eseguito. Non a caso, JAVA si è imposto negli ultimi anni come linguaggio di programmazione per sviluppare applicazioni da diffondere attraverso reti di calcolatori.

Un compilatore è un programma molto complesso e, come molti programmi complessi, è realizzato mediante un numero separato di parti che lavorano insieme: queste parti sono note come *fasi* e sono cinque.

- Analisi lessicale: in questa fase, il compilatore scompone il codice sorgente in unità significanti dette **token**. Questo compito è relativamente semplice per la maggior parte dei moderni linguaggi di programmazione, poichè il programmatore deve separare molte parti dell'istruzione con spazi o caratteri di tabulazione; questi spazi rendono più facile per il compilatore determinare dove un token finisce ed il prossimo ha inizio. Di questa fase ci occuperemo in questo capitolo e gli strumenti di cui faremo uso sono le espressioni regolari e gli automi a stati finiti.

- Analisi sintattica: in questa fase, il compilatore determina la struttura del programma e delle singole istruzioni. Di questa fase ci occuperemo nel prossimo capitolo, in cui vedremo che la costruzione di analizzatori sintattici poggia, in generale, sulle tecniche e i concetti sviluppati nella teoria dei linguaggi formali e, in particolare, sulle grammatiche generative libere da contesto.

- Generazione del codice intermedio: in questa fase, il compilatore crea una rappresentazione interna del programma che riflette l'informazione non coperta dall'analizzatore sintattico.

- Ottimizzazione: in questa fase, il compilatore identifica e rimuove operazioni ridondanti dal codice intermedio.
- Generazione del codice oggetto: in questa fase, infine, il compilatore traduce il codice intermedio ottimizzato nel linguaggio della macchina obiettivo.

Le ultime tre fasi vanno ben al di là degli obiettivi di queste dispense: rimandiamo il lettore ai tanti libri disponibili sulla progettazione e la realizzazione di compilatori (uno per tutti il famoso *dragone rosso* di Aho, Sethi e Ullman).

6.2 Analisi lessicale di linguaggi di programmazione

IL PRINCIPALE compito dell'analizzatore lessicale consiste nello scandire la sequenza del codice sorgente e scomporla in parti significanti, ovvero nei token di cui abbiamo parlato in precedenza. Per esempio, data l'istruzione JAVA

```
if (x == y*(b - a)) x = 0;
```

l'analizzatore lessicale deve essere in grado di isolare la parole chiave `if`, gli identificatori `x`, `y`, `b` ed `a`, gli operatori ==, `*`, − e =, le parentesi, il letterale `0` ed il punto e virgola finale. In questo capitolo svilupperemo un modo sistematico per estrarre tali token dalla sequenza sorgente (in realtà l'analizzatore lessicale può prendersi cura anche di altre cose, come, ad esempio, la rimozione dei commenti, la conversione dei caratteri, la rimozione degli spazi bianchi e l'interpretazione delle direttive di compilazione).

Abbiamo già osservato come nell'esempio precedente siano presenti quattro identificatori. Dal punto di vista sintattico, tuttavia, gli identificatori giocano tutti lo stesso ruolo ed è sufficiente dire in qualche modo che il prossimo oggetto nel codice sorgente è un identificatore (d'altro canto, sarà chiaramente importante, in seguito, essere in grado di distinguere i vari identificatori). Analogamente, dal punto di vista sintattico, un letterale intero è equivalente ad un altro letterale intero: in effetti, la struttura grammaticale dell'istruzione nel nostro esempio non cambierebbe se `0` fosse sostituito con `1` oppure con `1000`. Così tutto quello che in qualche modo dobbiamo dire è di aver trovato un letterale intero (di nuovo, in seguito, dovremo distinguere tra i vari letterali interi, poiché essi sono funzionalmente differenti anche se sintatticamente equivalenti).

Trattiamo questa distinzione nel modo seguente: il tipo generico, passato all'analizzatore sintattico, è detto token e le specifiche istanze del tipo generico sono dette **lessemi**. Possiamo dire che un token è il nome di un insieme di lessemi che hanno lo stesso significato grammaticale per l'analizzatore sintattico. Così nel nostro esempio

abbiamo quattro istanze (ovvero i lessemi `x`, `y`, `b` ed `a`) del token "identificatore", abbreviato come `id`, e un'istanza (ovvero il lessema `0`) del token "letterale intero", abbreviato come `int`. In molti casi, un token può avere una sola valida istanziazione: per esempio, le parole chiave non possono essere raggruppate in un solo token, in quanto hanno diversi significati per l'analizzatore sintattico. In questi casi, dunque, il token coincide con il lessema (lo stesso accade, nell'esempio precedente, con i token `(`, `)`, `==`, `*`, `-` e `;`). In conclusione, nel nostro esempio, la sequenza di token trasmessa dall'analizzatore lessicale a quello sintattico è la seguente:

```
if (id == id * (id - id)) id = int;
```

L'analizzatore lessicale deve quindi fare due cose: deve isolare i token ed anche prendere nota dei particolari lessemi. In realtà, l'analizzatore lessicale ha anche un compito aggiuntivo: quando un identificatore viene trovato, deve dialogare con il gestore della **tabella dei simboli**. Se l'identificatore viene dichiarato, un nuovo elemento verrà creato, in corrispondenza del lessema, nella tabella stessa. Se l'identificatore viene invece usato, l'analizzatore deve chiedere al gestore della tabella dei simboli di verificare che esista un elemento per il lessema nella tabella.

6.3 Automi a stati finiti

CHI SVILUPPA l'analizzatore lessicale definisce i token del linguaggio mediante espressioni regolari. Queste costituiscono una notazione compatta che indica quali caratteri possono far parte dei lessemi appartenenti a un particolare token e come devono essere messi in sequenza. Ad esempio, l'espressione regolare:

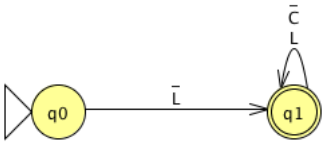
$$(0+\{1,2,3,4,5,6,7,8,9\}\{0,1,2,3,4,5,6,7,8,9\}^*)(\lambda+\mathtt{l}+\mathtt{L})$$

specifica il token letterale intero decimale in JAVA. L'analizzatore lessicale, tuttavia, viene meglio realizzato come un **automa a stati finiti**. Abbiamo già introdotto tali automi nella prima parte delle dispense: si tratta, infatti, di macchine di Turing che possono solo leggere e spostarsi a destra e che al momento in cui incontrano il primo simbolo $\square$ devono terminare accettando o rigettando la stringa di input.

Esempio 6.1: un automa a stati finiti per gli identificatori

Un identificatore consiste di una lettera o di un segno di sottolineatura seguito da una combinazione di lettere, segno di sottolineatura e cifre. In Figura 6.1 mostriamo un automa a stati finiti che identifica un token di questo tipo (per semplicità, abbiamo indicato con `L` e `C` l'insieme delle lettere e delle cifre, rispettivamente).

Figura 6.1: un automa per il token identificatore.



Osserviamo che, come mostrato nell'esempio precedente, quando si descrive un automa a stati finiti non è necessario specificare, nelle etichette degli archi, il simbolo scritto e il movimento, in quanto in un tale automa la scrittura non è consentita e l'unico movimento possibile è quello a destra. Osserviamo inoltre che, essendo il criterio di terminazione applicabile solo nel momento in cui l'intera stringa in input sia stata letta, uno stato finale può avere transizioni in uscita (come nel caso dello stato `q1` dell'esempio precedente). Infine, nel seguito indicheremo con $L(T)$ il linguaggio accettato dall'automa a stati finiti T, ovvero l'insieme delle stringhe x per cui esiste un cammino nel grafo, che parte dallo stato iniziale e finisce in uno stato finale, le etichette dei cui archi formano x. Due automi a stati finiti T_1 e T_2 sono detti essere **equivalenti** se $L(T_1) = L(T_2)$.

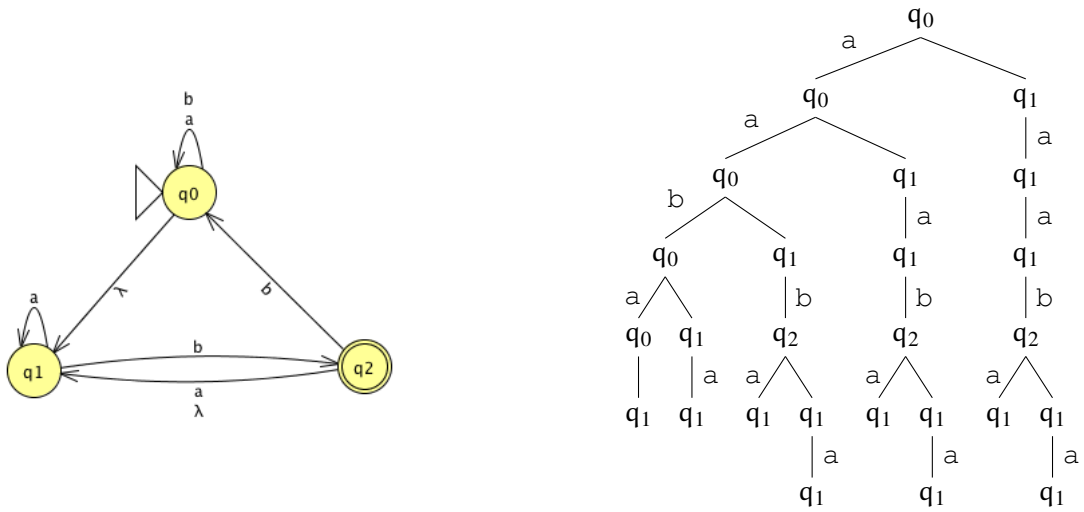
6.3.1 Automi a stati finiti non deterministici

Analogamente a quanto abbiamo fatto con le macchine di Turing nel precedente capitolo, possiamo modificare la definizione di automa a stati finiti consentendo un grado di non prevedibilità delle sue transizioni. Più precisamente, la transizione di un automa a stati finiti **non deterministico** può portare l'automa stesso in più stati diversi tra di loro. Come già osservato con le macchine di Turing non deterministiche, la computazione di un automa a stati finiti non deterministico corrisponde a un albero di possibili cammini di computazione: una stringa x è accettata dall'automa se tale albero include almeno un cammino che, dallo stato iniziale, conduce a uno stato finale e le etichette dei cui archi formino x. Nel caso degli automi a stati finiti, tuttavia, il non determinismo è generalmente esteso includendo la possibilità di λ-**transizioni**, ovvero transizioni che avvengono senza leggere alcun simbolo di input.

Esempio 6.2: un automa a stati finiti con transizioni nulle

Nella parte destra della Figura 6.2 è mostrato un automa non deterministico con λ-transizioni. L'albero delle computazioni corrispondente alla stringa `aaba` è mostrato nella parte destra della figura: come si può vedere, la stringa non viene accettata.

Figura 6.2: un automa a stati finiti non deterministico e un suo albero delle computazioni.



6.3.2 Automi deterministici e non deterministici

Nel capitolo precedente abbiamo dimostrato che una macchina di Turing non deterministica può essere simulata da una deterministica, facendo uso della tecnica di visita in ampiezza dell'albero delle computazioni. Chiaramente, tale tecnica non è realizzabile mediante un automa a stati finiti. Si potrebbe quindi pensare che gli automi a stati finiti non deterministici siano computazionalmente più potenti di quelli deterministici. Il prossimo risultato mostra come ciò non sia vero: l'idea alla base della dimostrazione consiste nello sfruttare il fatto che, per quanto le transizioni tra i singoli stati siano imprevedibili, le transizioni tra sottoinsiemi di stati sono al contrario univocamente determinate.

Teorema 6.1

Sia T un automa a stati finiti non deterministico senza λ-transizioni. Allora, esiste un automa a stati finiti T' equivalente a T.

Dimostrazione. La dimostrazione procede definendo uno dopo l'altro gli stati di T' sulla base degli stati già definiti e delle transizioni di T (l'alfabeto di lavoro di T' sarà uguale a quello di T): in particolare, ogni stato di T' denoterà un sottoinsieme degli stati di T. Lo stato iniziale di T' è lo stato $\{q_0\}$ dove q_0 è lo stato iniziale di T. La costruzione delle transizioni e degli altri stati di T' procede nel modo seguente.

Sia $Q = \{s_1, \ldots, s_k\}$ uno stato di T' per cui non è ancora stata definita la transizione corrispondente a un simbolo σ dell'alfabeto di lavoro di T e, per ogni i con $i = 1, \ldots, k$, sia $N(s_i, \sigma)$ l'insieme degli stati raggiungibili da s_i leggendo il simbolo σ (osserviamo che S potrebbe anche essere l'insieme vuoto). Definiamo allora $S = \bigcup_{i=1}^{k} N(s_i, \sigma)$ e introduciamo la transizione di T' dallo stato Q allo stato S leggendo il simbolo σ. Inoltre, aggiungiamo S all'insieme degli stati di T', nel caso non ne facesse già parte. Al termine di questo procedimento, identifichiamo gli stati finali di T' come quegli stati corrispondenti a sottoinsiemi contenenti almeno uno stato finale di T. È facile dimostrare che una stringa x è accettata da T se e solo se è accettata anche da T', ovvero $L(T) = L(T')$. ◇

Esempio 6.3: trasformazione da automa non deterministico ad automa deterministico

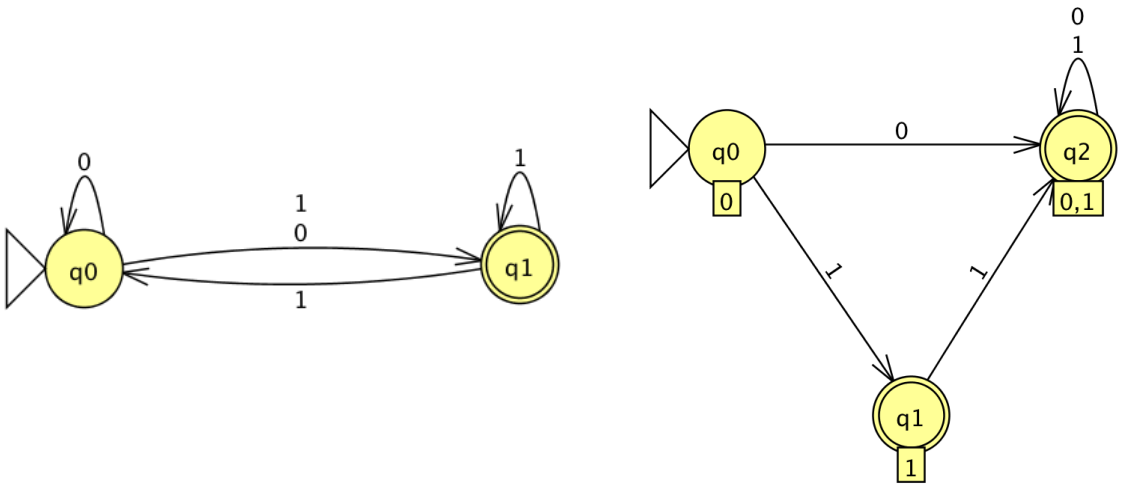
Consideriamo l'automa non deterministico T definito dalla seguente tabella delle transizioni, in cui, per ogni riga, specifichiamo l'insieme degli stati in cui l'automa può andare leggendo uno specifico simbolo a partire da un determinato stato (assumiamo che `q0` sia lo stato iniziale e `q1` quello finale, come mostrato nella parte sinistra della Figura 6.3).

stato	simbolo	insieme di stati
q0	0	{q0, q1}
q0	1	{q1}
q1	1	{q0, q1}

L'automa deterministico T' equivalente a T include lo stato $\{q0\}$. Relativamente a esso abbiamo che $N(q0, 0) = \{q0, q1\}$ e che $N(q0, 1) = \{q1\}$. Questi due stati non sono ancora stati generati: li aggiungiamo all'insieme degli stati di T' e definiamo una transizione verso di essi a partire dallo stato $\{q0\}$ leggendo il simbolo `0` e il simbolo `1`, rispettivamente. Dobbiamo ora definire la transizione a partire da $\{q0, q1\}$ leggendo il simbolo `0`. In questo caso, $N(q0, 0) = \{q0, q1\}$ e $N(q1, 0) = \{\}$: l'unione di questi due insiemi è uguale a $\{q0, q1\}$, che già esiste nell'insieme degli stati di T'. È sufficiente quindi definire la transizione da $\{q0, q1\}$ a se stesso leggendo `0`. Procedendo, abbiamo che $N(q0, 1) = \{q1\}$ e che $N(q1, 1) = \{q0, q1\}$: non dobbiamo aggiungere nessuno stato ma solo la transizione da $\{q0, q1\}$ a se stesso leggendo `1`. In modo analogo possiamo calcolare le transizioni a partire dallo stato $\{q1\}$: la tabella finale delle transizioni di T' è la seguente in cui lo stato iniziale `Q0` corrisponde all'insieme $\{q0\}$, lo stato finale `Q1` all'insieme $\{q0,q1\}$ e lo stato finale `Q2` all'insieme $\{q1\}$ (si veda la parte destra della Figura 6.3).

stato	simbolo	stato
Q0	0	Q1
Q0	1	Q2
Q1	0	Q1
Q1	1	Q1
Q2	1	Q1

Figura 6.3: trasformazione di un automa a stati finiti non deterministico in uno deterministico.

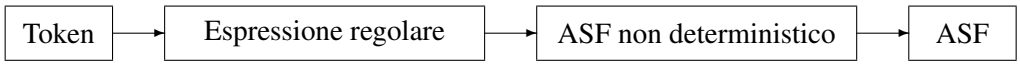


La dimostrazione del Teorema 6.1 può essere opportunamente modificata in modo da estendere il risultato al caso in cui l'automa a stati finiti non deterministico contenga λ-transizioni. In effetti, abbiamo bisogno di una sola modifica che consiste, per lo stato iniziale di T' e per ogni nuovo stato, nell'includere tutti gli stati che possono essere raggiunti da tale stato mediante una o più λ-transizioni. Tale operazione è anche detta λ-**chiusura** di uno stato. Formalmente, la λ-*estensione* di un insieme di stati A di un automa non deterministico è definita come l'insieme degli stati che sono collegati direttamente a uno stato di A mediante una transizione con etichetta λ. La λ-chiusura di A si ottiene allora calcolando ripetutamente la λ-estensione di A fino a quando non vengono aggiunti nuovi stati. L'algoritmo di costruzione mediante sottoinsiemi dell'automa T' descritto nella dimostrazione del Teorema 6.1 può dunque essere modificato nel modo seguente: lo stato iniziale di T' corrisponde alla λ-chiusura di $\{q_0\}$ e lo stato S è definito come la λ-chiusura di $\bigcup_{i=1}^{k} N(s_i, \sigma)$.

6.4 Espressioni regolari

NEL PARAGRAFO precedente abbiamo visto come gli automi a stati finiti non deterministici non sono computazionalmente più potenti di quelli deterministici. In tal caso, a che cosa servono gli automi a stati finiti non deterministici? La risposta è che il nostro scopo consiste nel trovare un modo di costruire le tabelle degli stati di

Figura 6.4: dal token all'automa a stati finiti equivalente



automi a stati finiti a partire dalle definizioni dei token. In questo paragrafo vedremo come le espressioni regolari risultino un modo molto conveniente ed economico di specificare i token. Vedremo anche che è relativamente semplice, anche se laborioso, trovare un automa a stati finiti non deterministico equivalente a una data espressione regolare: da quest'automa possiamo costruirne uno deterministico equivalente che può quindi essere usato dall'analizzatore lessicale. Il processo di generazione della tabella degli stati finale consiste quindi dei passi mostrati in Figura 6.4. Noi siamo partiti dalla fine della catena mostrata in figura e stiamo ora per tornare all'inizio. L'intero processo è laborioso ma non richiede molto esercizio mentale e può quindi essere delegato a un calcolatore.

Definizione 6.1: espressioni regolari

L'insieme delle espressioni regolari su di un alfabeto Σ è definito induttivamente come segue.

- Ogni carattere in Σ è un'espressione regolare.
- λ è un'espressione regolare.
- Se R ed S sono due espressioni regolari, allora
  - La *concatenazione* $R \cdot S$ (o, semplicemente, RS) è un'espressione regolare.
  - La *selezione* $R + S$ è un'espressione regolare.
  - La *chiusura di Kleene* R^* è un'espressione regolare.
- Solo le espressioni formate da queste regole sono regolari.

Nel valutare un'espressione regolare supporremo nel seguito che la chiusura di Kleene abbia priorità maggiore mentre la selezione abbia priorità minore: le parentesi verrano anche usate per annullare le priorità nel modo usuale.

Un'espressione regolare R genera un linguaggio $L(R)$ definito anch'esso in modo induttivo nel modo seguente.

- Se $R = a \in \Sigma$, allora $L(R) = \{a\}$.
- Se $R = \lambda$, allora $L(R) = \{\lambda\}$.
- Se $R = S_1 \cdot S_2$, allora $L(R) = \{xy : x \in L(S_1) \wedge y \in L(S_2)\}$.
- Se $R = S_1 + S_2$, allora $L(R) = L(S_1) \cup L(S_2)$.
- Se $R = S^*$, allora $L(R) = \{x_1 x_2 \dots x_n : n \geqslant 0 \wedge x_i \in L(S)\}$.

Osserviamo che la chiusura di Kleene consente zero concatenazioni, per cui il linguaggio generato contiene la stringa vuota.

Due espressioni regolari R e S sono equivalenti se $L(R) = L(S)$. Alcune equivalenze, la cui dimostrazione è lasciata come esercizio (si veda l'Esercizio 6.3) sono utili talvolta per analizzare espressioni regolari.

Teorema 6.2

Se R, S e T sono tre espressioni regolari, allora le seguenti affermazioni sono vere.

Associatività $R(ST)$ è equivalente a $(RS)T$.

Associatività $R + (S + T)$ è equivalente a $(R + S) + T$.

Commutatività $R + S$ è equivalente a $S + R$.

Distributività $R(S + T)$ è equivalente a $RS + RT$.

Identità $R\lambda$ ed λR sono equivalenti a R.

Osserviamo che la concatenazione non è commutativa in quanto, in generale, $L(RS) \neq L(SR)$ (si veda l'Esercizio 6.4).

Le espressioni regolari sono in grado di generare solo un insieme limitato di linguaggi ma sono abbastanza potenti da poter essere usate per definire i token. Abbiamo già visto, infatti, che un token può essere visto come un linguaggio che include i suoi lessemi: rappresentando il token mediante un espressione regolare, definiamo esattamente in che modo i lessemi sono riconosciuti come elementi del linguaggio. Abbiamo già osservato, ad esempio, come la seguente espressione regolare

$$(0 + \{1,2,3,4,5,6,7,8,9\}\{0,1,2,3,4,5,6,7,8,9\}^*)(\lambda + l + L)$$

specifichi il token letterale intero decimale in JAVA. Infatti, tale token è definito come un numerale decimale eventualmente seguito dal suffisso `l` oppure `L` allo scopo di indicare se la rappresentazione deve essere a 64 bit. Un numerale decimale, a sua volta, può essere uno `0` oppure una cifra da `1` a `9` eventualmente seguita da una o più cifre da `0` a `9`.

6.4.1 Espressioni regolari ed automi a stati finiti

Siamo ora in grado di descrivere come può essere realizzato il secondo passo della catena che porta dalla definizione del token al corrispondente automa a stati finiti non deterministico.

Teorema 6.3

Per ogni espressione regolare R esiste un automa a stati finiti non deterministico T tale che $L(R) = L(T)$.

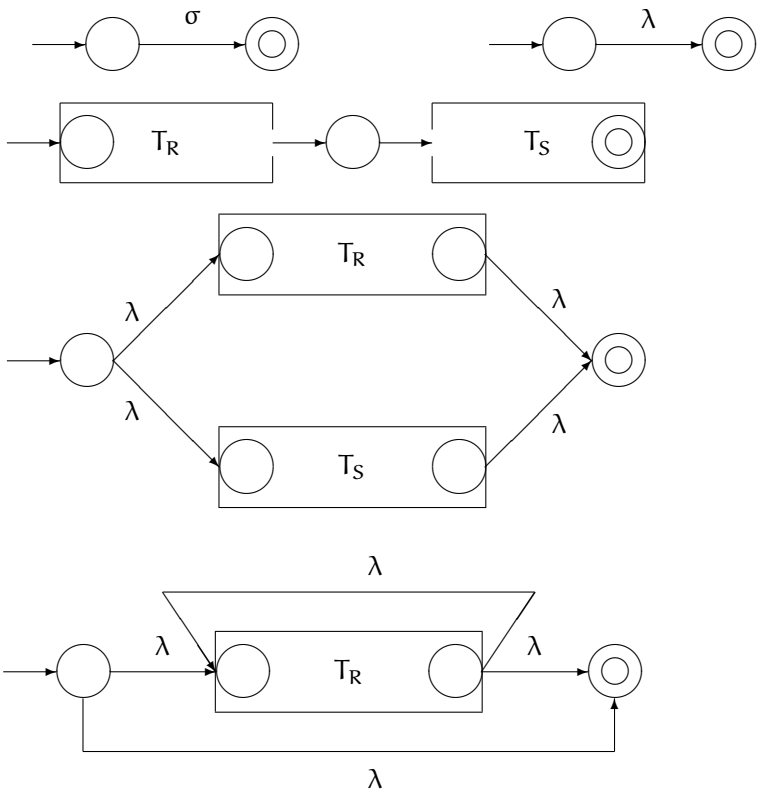
Dimostrazione. La costruzione si basa sulla definizione induttiva delle espressioni regolari fornendo una macchina oppure un'interconnessione di macchine corrispondente a ogni passo della definizione. La costruzione, che tra l'altro assicura che l'automa ottenuto avrà un solo stato finale diverso dallo stato iniziale e senza transizioni in uscita, è la seguente.

- La macchina mostrata in alto a sinistra della Figura 6.5 accetta il carattere $\sigma \in \Sigma$ mentre quella mostrata in alto a destra accetta λ.
- Date due espressioni regolari R e S, la macchina mostrata nella seconda riga della Figura 6.5 accetta $L(RS)$ dove T_R e T_S denotano due macchine che decidono, rispettivamente, $L(R)$ ed $L(S)$ e lo stato finale di T_R è stato fuso con lo stato iniziale di T_S in un unico stato.
- Date due espressioni regolari R e S, la macchina mostrata nella terza riga della Figura 6.5 accetta $L(R+S)$ dove T_R e T_S denotano due macchine che riconoscono, rispettivamente, $L(R)$ ed $L(S)$, un nuovo stato iniziale è stato creato, due λ-transizioni da questo nuovo stato agli stati iniziali di T_R ed T_S sono state aggiunte, un nuovo stato finale è stato creato e due λ-transizioni dagli stati finali di T_R ed T_S a questo nuovo stato sono state aggiunte.
- Data un'espressione regolare R, la macchina mostrata nella quarta riga della Figura 6.5 accetta $L(R^*)$ dove T_R denota una macchina che riconosce $L(R)$, un nuovo stato iniziale e un nuovo stato finale sono stati creati, due λ-transizioni dal nuovo stato iniziale al nuovo stato finale e allo stato iniziale di T_R sono state aggiunte e due λ-transizioni dallo stato finale di T_R allo stato iniziale di T_R e al nuovo stato finale sono state aggiunte.

È facile verificare che la costruzione sopra descritta permette di definire un automa a stati finiti non deterministico equivalente a una specifica espressione regolare. ◇

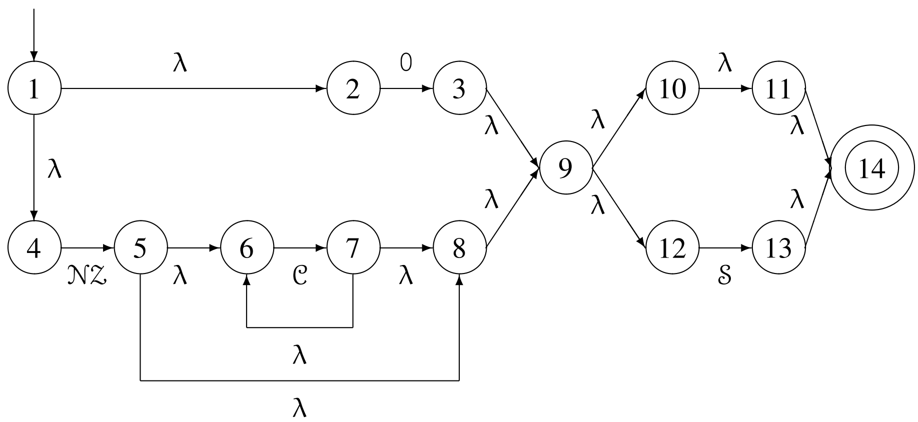
Dall'automa a stati finiti non deterministico ottenuto dalla costruzione precedente possiamo ottenere un automa a stati finiti equivalente all'espressione regolare di

Figura 6.5: da espressioni regolari ad automi non deterministici



partenza che può quindi essere incorporato nell'analizzatore lessicale. Ciò completa dunque la nostra catena: sappiamo ora come ottenere, partendo da un'espressione regolare, l'equivalente automa a stati finiti facendo uso dell'automa a stati finiti non deterministico nel passo intermedio. Il ben noto programma `Lex` genera una tabella degli stati per un analizzatore lessicale essenzialmente in questo modo: l'utente specifica i token mediante espressioni regolari, `Lex` calcola gli equivalenti automi a stati finiti non deterministici e da quelli genera la tabella degli stati per gli automi a stati finiti. In realtà, quando costruiamo un automa a stati finiti equivalente a uno non deterministico (a sua volta ottenuto da un'espressione regolare) si crea generalmente un'esplosione esponenziale di stati, per la maggior parte ridondanti: sebbene ciò sia

Figura 6.6: automa non deterministico per il token letterale intero decimale



al di là degli scopi di queste dispense, è bene sapere che è possibile identificare ed eliminare questi stati non necessari attraverso un processo di minimizzazione degli stati, di cui lo stesso programma `Lex` fa uso.

Osserviamo inoltre che la catena mostrata nella Figura 6.4 può in realtà essere trasformata in un ciclo dimostrando che a ogni automa a stati finiti corrisponde un'espressione regolare a esso equivalente: la dimostrazione di tale risultato va comunque al di là degli obiettivi di queste dispense.

Un esempio

Abbiamo già osservato che il token letterale intero decimale in JAVA può essere definito mediante la seguente espressione regolare:

$$(\mathtt{0}+\{\mathtt{1},\mathtt{2},\mathtt{3},\mathtt{4},\mathtt{5},\mathtt{6},\mathtt{7},\mathtt{8},\mathtt{9}\}\{\mathtt{0},\mathtt{1},\mathtt{2},\mathtt{3},\mathtt{4},\mathtt{5},\mathtt{6},\mathtt{7},\mathtt{8},\mathtt{9}\}^*)(\lambda+\mathtt{l}+\mathtt{L}).$$

Nel seguito indicheremo con $\mathcal{C}$ l'insieme delle cifre da `0` a `9`, con $\mathcal{NZ}$ l'insieme delle cifre da `1` a `9` e con $\mathcal{S}$ l'insieme $\{\mathtt{l}, \mathtt{L}\}$. Quindi l'espressione regolare associata al token letterale intero decimale diviene la seguente:

$$(\mathtt{0}|\mathcal{NZ}(\mathcal{C})^*)(\epsilon|\mathcal{S}).$$

Da quest'espressione regolare intendiamo ora ottenere l'equivalente automa a stati finiti mediante le tecniche viste nei paragrafi precedenti. Anzitutto, costruiamo l'au-

toma a stati finiti non deterministico T equivalente all'espressione regolare facendo uso della dimostrazione del Teorema 6.3: tale automa è mostrato in Figura 6.6.

Applicando poi la trasformazione di un automa non deterministico in uno deterministico mediante la tecnica utilizzata nella dimostrazione del Teorema 6.1 (estesa in modo da gestire anche le λ-transizioni), otteniamo l'automa a stati finiti T′ definito dalla seguente tabella delle transizioni:

stato	simbolo	stato
Q0	0	Q1
Q0	$\mathcal{NZ}$	Q2
Q1	$\mathcal{S}$	Q3
Q2	0	Q4
Q2	$\mathcal{NZ}$	Q4
Q2	$\mathcal{S}$	Q3
Q4	0	Q4
Q4	$\mathcal{NZ}$	Q4
Q4	$\mathcal{S}$	Q3

In tale tabella, lo stato iniziale Q0 corrisponde all'insieme $\{1,2,4\}$ degli stati di T, mentre gli stati Q1, Q2, Q3 e Q4 sono tutti finali e corrispondono rispettivamente agli insiemi $\{3,9,10,11,12,14\}$, $\{5,6,8,9,10,11,12,14\}$, $\{13,14\}$ e $\{6,7,8,9,10,11,12,14\}$ degli stati di T. Osserviamo come gli stati Q2 e Q4 siano equivalenti: eliminando lo stato Q4 otteniamo l'automa a stati finiti finale la cui tabella degli stati è la seguente:

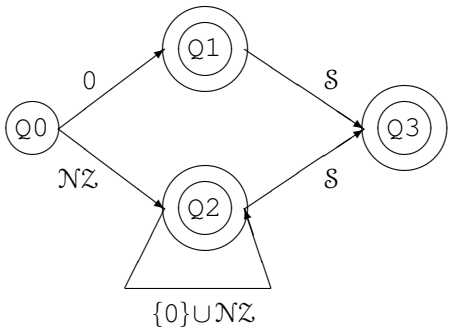
stato	simbolo	stato
Q0	0	Q1
Q0	$\mathcal{NZ}$	Q2
Q1	$\mathcal{S}$	Q3
Q2	0	Q2
Q2	$\mathcal{NZ}$	Q2
Q2	$\mathcal{S}$	Q3

Tale automa è mostrato anche in Figura 6.7.

6.5 Automi a stati finiti e grammatiche regolari

COME ABBIAMO visto nel capitolo precedente, le grammatiche regolari o di tipo 3 sono grammatiche le cui regole di produzione sono **lineari a destra**, ovvero del tipo $X \to a$ oppure del tipo $X \to aY$. Il prossimo risultato mostra come

Figura 6.7: automa deterministico per il token letterale intero decimale



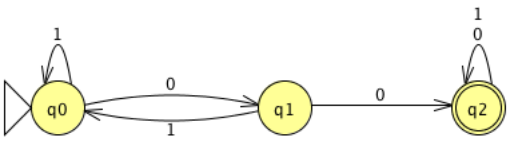
tali grammatiche siano equivalenti agli automi a stati finiti e, quindi, all'espressioni regolari.

Teorema 6.4

Un linguaggio L è regolare se e solo se esiste un automa a stati finiti che decide L.

Dimostrazione. Sia T un automa a stati finiti e sia $L = L(T)$ il linguaggio deciso da T. Costruiamo ora una grammatica regolare G che genera tutte e sole le stringhe di L. Tale grammatica avrà un simbolo non terminale per ogni stato di T: il simbolo iniziale sarà quello corrispondente allo stato iniziale. Per ogni transizione che dallo stato q fa passare l'automa nello stato p leggendo il simbolo a, la grammatica include la regola di produzione $Q \to aP$, dove Q e P sono i due simboli non terminali corrispondenti agli stati q e p, rispettivamente. Inoltre, se p è uno stato finale, allora la grammatica include anche la transizione $Q \to a$. È facile verificare che $L(T) = L(G)$ (si veda ad esempio, la Figura 6.8). Viceversa, supponiamo che G sia una grammatica regolare e che $L = L(G)$ sia il linguaggio generato da G. Definiamo un automa a stati finito non deterministico T tale che $L(T) = L$: in base a quanto visto nei paragrafi precedenti, questo dimostra che esiste un automa a stati finiti equivalente a G. L'automa T ha uno stato per ogni simbolo non terminale di G: lo stato iniziale sarà quello corrispondente al simbolo iniziale di G. Per ogni produzione del tipo $X \to aY$ di G, T avrà una transizione dallo stato corrispondente a X allo stato corrispondente a Y leggendo il simbolo a (in questo modo, possiamo avere che T sia non deterministico). Inoltre, per ogni produzione del tipo $X \to a$ di G, T avrà una transizione dallo stato corrispondente a X all'unico stato finale F leggendo il simbolo a. Di nuovo, è facile verificare che $L(T) = L(G)$ (si veda ad esempio, la Figura 6.9). ◇

Figura 6.8: un automa a stati finiti e la corrispondente grammatica regolare.



$$Q_0 \to 1Q_0 \quad Q_0 \to 0Q_1$$
$$Q_1 \to 1Q_0 \quad Q_1 \to 0Q_2 \quad Q_1 \to 0$$
$$Q_2 \to 0Q_2 \quad Q_2 \to 0 \quad Q_2 \to 1Q_2 \quad Q_2 \to 1$$

6.5.1 Linguaggi non regolari

Il teorema precedente, oltre ad essere di per sé interessante, fornisce un'immediata intuizione su come cercare di dimostrare che esistono linguaggi non regolari. In effetti, se un linguaggio L è deciso da un automa a stati finiti T, quest'ultimo deve necessariamente avere un numero finito n di stati. Ciò significa che tutte le stringhe in L di lunghezza superiore a n devono corrispondere a una computazione di T che visita lo stesso stato più di una volta. Quindi, T deve contenere un ciclo, ovvero un cammino da uno stato q ad altri stati che ritorna poi in q: tale ciclo può essere "pompato" a piacere tante volte quante lo desideriamo, producendo comunque sempre stringhe in L. Formalizzando tale ragionamento, possiamo dimostrare il seguente risultato.

Lemma 6.1

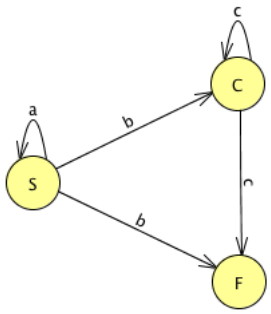
Se L è un linguaggio infinito regolare, allora esiste un numero intero $n_L > 0$, tale che, per ogni stringa $x \in L$ con $|x| > n_L$, x può essere decomposta nella concatenazione di tre stringhe y, v e z per cui valgono le seguenti affermazioni.

1. $|v| > 0$.
2. $|yv| < n_L$.
3. Per ogni $i \geqslant 0$, $yv^i z \in L$.

Dimostrazione. Sia T un automa a stati finiti tale che $L = L(T)$: poniamo n_L uguale al numero di stati di T. Sia x una stringa in L di lunghezza n maggiore di n_L: consideriamo la computazione di T con input x. Sia $q_0, q_1, q_2, \ldots, q_n$ la sequenza di stati attraverso cui passa tale computazione (q_0 è lo stato iniziale, mentre q_n è uno degli stati finali): poiché $n > n_L$, tale sequenza deve contenere uno stato ripetuto nei primi $n_L + 1$ elementi. Sia q_j il primo stato ripetuto e supponiamo che si ripeta dopo $m > 0$ passi (ovvero, $q_j = q_{j+m}$ con $j + m \leqslant n_L$): definiamo allora y come la sequenza delle etichette delle transizioni eseguite nel passare da q_0 a q_j, v come la sequenza

Figura 6.9: una grammatica regolare e il corrispondente automa a stati finiti.

$S \to aS \quad S \to bC \quad S \to b$
$C \to cC \quad C \to c$



delle etichette delle transizioni eseguite nel passare da q_j a q_{j+m} e con z il resto della stringa x. Abbiamo che $|v| = m > 0$ e che $|yv| = j + m - 1 < n_L$. Inoltre, y fa passare T da q_0 a q_j, v fa passare T da q_j a q_j e z fa passare T da q_j a q_n (che è uno stato finale): pertanto, per ogni $i \geqslant 0$, la stringa yv^iz fa passare T da q_0 a q_n e, quindi, appartiene a L. ◇

Il risultato precedente fornisce una condizione necessaria affinché un linguaggio sia regolare: per questo motivo, esso viene principalmente utilizzato per dimostrare che un linguaggio *non* è regolare.

Esempio 6.4: un linguaggio non regolare

Consideriamo il linguaggio L dell'Esempio 5.1 costituito da tutte e sole le stringhe del tipo $a^n b^n$, per $n > 0$. Chiaramente questo linguaggio è infinito. Se L fosse regolare, allora esisterebbe il numero $n_L > 0$ definito nel Lemma 6.1: consideriamo la stringa $a^{n_L} b^{n_L}$. Dal lemma, segue che tale stringa può essere decomposta nella concatenazione di tre stringhe y, v e z tale che $|v| > 0$ e $yv^2z \in L$. D'altra parte, poiché $|yv| < n_L$, abbiamo che v deve essere costituita dal solo simbolo a, per cui, assumendo che $yvz \in L$, yv^2z conterrebbe un numero di simboli a maggiore del numero di simboli b, contraddicendo il fatto che $yv^2z \in L$: pertanto, L non può essere un linguaggio regolare.

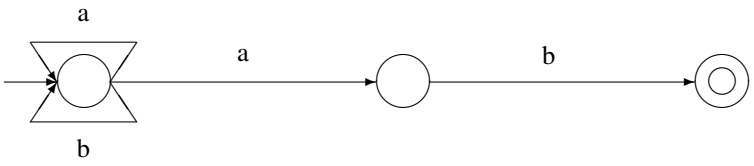
In conclusione, quanto esposto in questo capitolo ci consente di aggiornare la tabella di classificazione dei linguaggi in base alla loro tipologia, al tipo di grammatiche che li generano e in base al modello di calcolo corrispondente: la nuova tabella è la seguente.

Tipo di linguaggio	Tipo di produzioni	Modello di calcolo
Tipo 0	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$ e $\beta \in (V \cup T)^*$	Macchina di Turing
Contestuale	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$, $\beta \in (V \cup T)(V \cup T)^*$ e $\lvert\beta\rvert \geqslant \lvert\alpha\rvert$	Macchina di Turing lineare
Libero da contesto	$A \to \beta$ con $A \in V$ e $\beta \in (V \cup T)(V \cup T)^*$	
Regolare	$A \to aB$ e $A \to a$ con $A, B \in V$ e $a \in T$	Automa a stati finiti

Esercizi

Esercizio 6.1. Trasformare l'automa a stati finiti non deterministico della Figura 6.2 in un automa deterministico equivalente, facendo uso della tecnica di costruzione mediante sottoinsiemi.

Esercizio 6.2. Facendo uso della tecnica di costruzione mediante sottoinsiemi, trasformare il seguente automa a stati finiti non deterministico in un automa deterministico equivalente.



Esercizio 6.3. Dimostrare il Teorema 6.2.

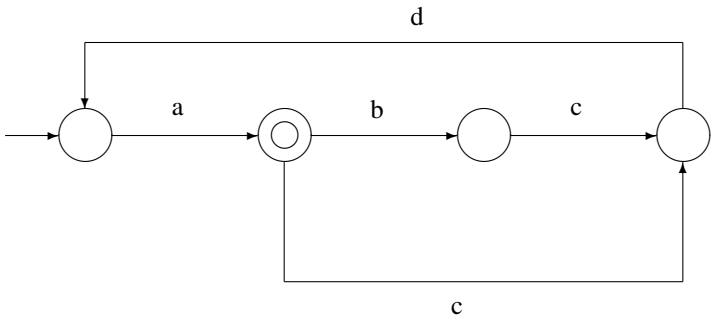
Esercizio 6.4. Dimostrare che la concatenazione non è commutativa.

Esercizio 6.5. Derivare un'espressione regolare che generi l'insieme di tutte le sequenze di 0 ed 1 che contengono un numero di 0 divisibile per 3.

Esercizio 6.6. Sia rev un operatore che inverte le sequenze di simboli: ad esempio, $rev(abc) = cba$. Sia L un linguaggio generato da un'espressione regolare e sia $L^r = \{x : rev(x) \in L\}$. Dimostrare che esiste un'espressione regolare che genera L^r.

Esercizio 6.7. Definire un automa a stai finiti non deterministico che accetta il linguaggio generato dalla seguente espressione regolare: $(((1 \cdot 1)^* 0)^* + 0 \cdot 0)^*$. Quindi derivare l'equivalente automa a stati finiti.

Esercizio 6.8. Definire un'espressione regolare che generi il linguaggio accettato dal seguente automa a stati finiti.



Esercizio 6.9. Dimostrare che il linguaggio complementare di quello considerato nell'Esempio 6.4 non è regolare.

Esercizio 6.10. Dire, giustificando la risposta, quali delle seguenti affermazioni sono vere.

1. Se L è un linguaggio regolare e L' è un linguaggio non regolare, allora $L \cap L'$ è un linguaggio regolare se e solo se $L \cup L'$ è un linguaggio regolare.
2. Se $L_1 \subseteq L_2$ e L_2 è un linguaggio regolare, allora L_1 è un linguaggio regolare.
3. Se L_1 e L_2 sono due linguaggi non regolari, allora $L_1 \cup L_2$ non può essere un linguaggio regolare.
4. Se L_1 è un linguaggio non regolare, allora il complemento di L_1 non può essere un linguaggio regolare.

Esercizio 6.11. Utilizzando il Lemma 6.1 dimostrare che il linguaggio costituito da tutte e sole le stringhe binarie del tipo ww, con $w \in \{0,1\}^*$, non è regolare.

Esercizio 6.12. Utilizzando il Lemma 6.1 dimostrare che il linguaggio costituito da tutte e sole le stringhe del tipo 0^{2^n}, con $n \geqslant 0$, non è regolare.

Esercizio 6.13. Facendo uso del pumping lemma per i linguaggi regolari, dimostrare che il seguente linguaggio non è regolare.

$$L = \{0^i 1^j : i > j\}$$

Esercizio 6.14. Facendo uso del pumping lemma per i linguaggi regolari, dimostrare che il seguente linguaggio non è regolare.

$$L = \{x2y : x, y \in \{0,1\}^* \text{ e il numero di } 0 \text{ in } x \text{ è uguale al numero di } 1 \text{ in } y\}$$

Esercizio 6.15. Facendo uso del pumping lemma, dimostrare che il seguente linguaggio non è regolare.

$$L = \{0^p : p \text{ è un numero primo}\}$$

Esercizio 6.16. Dimostrare che il linguaggio

$$L = \{0^i 1^j 2^k : i = 1 \Rightarrow j = k\}$$

soddisfa le condizioni del Lemma 6.1, pur non essendo regolare.

Esercizio 6.17. Si consideri la seguente dimostrazione.

> Sia L il linguaggio costituito da tutte e sole le stringhe binarie di lunghezza pari a 1000. Supponiamo che L sia regolare. Sia $x = 0^{1000}$ una stringa in L: in base al pumping lemma per i linguaggi regolari, x può essere decomposta nella concatenazione di tre stringhe y, v e z tali che $|v| > 0$ e, per ogni $i > 0$, $yv^iz \in L$. Ma questo contraddice il fatto che L contiene solo stringhe di lunghezza pari a 1000. Quindi, L non è regolare.

Tale dimostrazione è certamente sbagliata. Perché? Dire, inoltre, qual è l'errore contenuto nella dimostrazione.

Capitolo 7

Analisi sintattica top-down

SOMMARIO

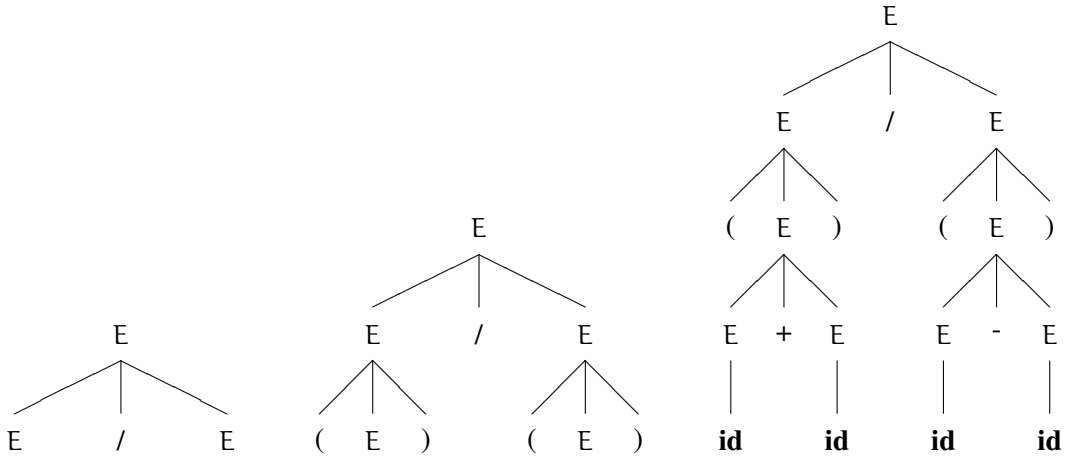
Una delle principali componenti di un compilatore è il parser, il cui compito è quello di analizzare la struttura di un programma e delle sue istruzioni componenti e di verificare l'esistenza di errori. La risorsa principale nello sviluppo di un parser sono le grammatiche libere da contesto. In questo capitolo, analizzeremo tali grammatiche e mostreremo una tecnica per utilizzarle per effettuare l'analisi sintattica di un programma, basata sulla costruzione top-down degli alberi di derivazione.

7.1 Introduzione

L'analizzatore sintattico, anche detto *parser*, è il cuore della prime tre fasi di un compilatore. Il suo compito principale è quello di analizzare la struttura del programma e delle sue istruzioni componenti e di verificare l'esistenza di errori. A tale scopo interagisce con l'analizzatore lessicale, che fornisce i token in risposta alle richieste del parser: allo stesso tempo, il parser può supervisionare le operazioni del generatore di codice intermedio.

La risorsa principale nello sviluppo del parser sono le grammatiche libere da contesto, le quali sono molto spesso utilizzate per definire uno specifico linguaggio di programmazione. A essere precisi, molti linguaggi di programmazione reali non possono essere descritti completamente da questo tipo di grammatiche: in tali linguaggi, infatti, vi sono spesso delle restrizioni che non possono essere imposte da grammatiche libere da contesto. Per esempio, i linguaggi fortemente tipati richiedono che ogni variabile sia dichiarata prima di essere usata: le grammatiche libere da contesto non sono in grado di imporre tale vincolo (d'altra parte, le grammatiche che possono farlo sono troppo complesse per essere usate nella costruzione di compilatori). In ogni caso, eccezioni come queste sono rare e possono essere gestite semplicemente con altri mezzi: pertanto le grammatiche libere da contesto, essendo

Figura 7.1: lo sviluppo di un albero di derivazione.



così comode da usare e fornendo così facilmente metodi efficienti per la costruzione di analizzatori sintattici, sono generalmente utilizzate per il progetto dei compilatori.

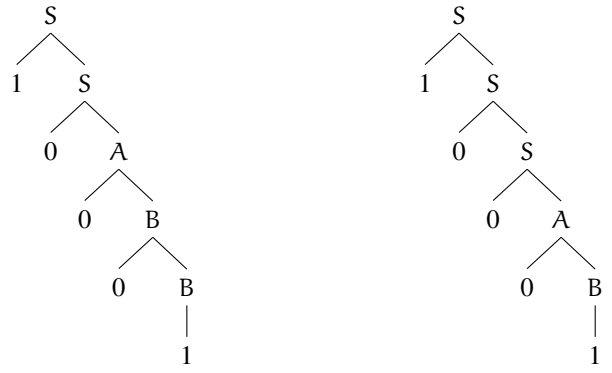
7.2 Alberi di derivazione

Consideriamo le espressioni aritmetiche in un linguaggio di programmazione come JAVA formate facendo uso delle sole operazioni di somma, sottrazione, moltiplicazione e divisione. Una semplice grammatica libera da contesto per tali espressioni è la seguente:

$$E \to E+E \quad E \to E-E \quad E \to E*E$$
$$E \to E/E \quad E \to (E) \quad E \to \mathbf{id}$$

Facendo uso di tale grammatica, analizziamo l'espressione $(a+b)/(a-b)$. L'analizzatore lessicale passerà quest'espressione in forma di token, sostituendo i lessemi a e b con il token **id**. Quindi la sequenza di simboli terminali da dover generare è $(\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})$. Quest'espressione è una frazione il cui numeratore è $(\mathbf{id}+\mathbf{id})$ e il cui denominatore è $(\mathbf{id}-\mathbf{id})$: pertanto, la prima produzione che usiamo è $E \to E/E$ come mostrato nella parte sinistra della Figura 7.1. Osserviamo che la produzione è rappresentata mediante un albero in cui la parte a sinistra della produzione è associata al nodo padre e i simboli della parte destra ai nodi figli. Il numeratore e il denominatore della frazione sono entrambi espressioni parentesizzate: quindi, li sostituiamo

Figura 7.2: alberi di derivazione.



entrambi facendo uso della regola $\mathtt{E} \to (\mathtt{E})$ come mostrato nella parte centrale della Figura 7.1. Il contenuto della parentesi al numeratore è una somma: quindi, sostituiamo la $\mathtt{E}$ all'interno della parentesi facendo uso della produzione $\mathtt{E} \to \mathtt{E} + \mathtt{E}$. Il denominatore invece è una differenza: quindi, sostituiamo la $\mathtt{E}$ all'interno della parentesi con la produzione $\mathtt{E} \to \mathtt{E} - \mathtt{E}$. A questo punto, le $\mathtt{E}$ rimanenti devono produrre i token identificatori terminali: quindi, facciamo uso della regola $\mathtt{E} \to$ **id** come mostrato nella parte destra della Figura 7.1.

Data una grammatica $G = (V, N, S, P)$, un **albero di derivazione** è un albero la cui radice è etichettata con S (ovvero il simbolo iniziale della grammatica), le cui foglie sono etichettate con simboli in V (ovvero simboli terminali) e tale che, per ogni nodo con etichetta un simbolo non terminale A, i figli di tale nodo siano etichettati con i simboli della parte destra di una produzione in P la cui parte sinistra sia A. Un albero di derivazione è detto essere un albero di derivazione della stringa x se i simboli che etichettano le foglie dell'albero, letti da sinistra verso destra, formano la stringa x.

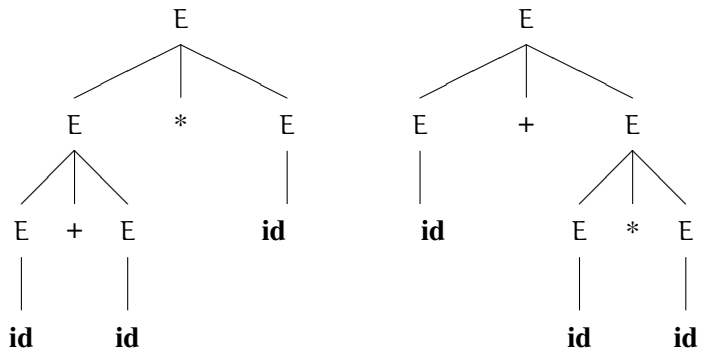
Esempio 7.1: alberi di derivazione

Consideriamo la grammatica regolare contenente le seguenti produzioni:

$$S \to 1S \quad S \to 0S \quad S \to 0A \quad A \to 0B \quad A \to 0 \quad B \to 0B \quad B \to 1B \quad B \to 0 \quad B \to 1.$$

Un esempio di albero di derivazione della stringa `10001` è mostrato nella parte sinistra della Figura 7.2, mentre un altro esempio di albero di derivazione della stessa stringa è mostrato nella parte destra della figura.

Figura 7.3: due alberi di derivazione in una grammatica ambigua.

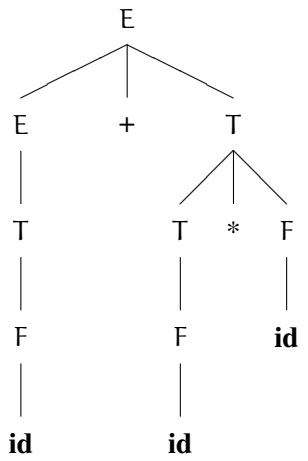


7.2.1 Grammatiche ambigue

L'esempio precedente mostra un'altra importante caratteristica delle grammatiche generative, ovvero il fatto che tali grammatiche possano essere ambigue in quanto la stessa stringa può essere prodotta in diversi modi. Nell'esempio precedente ciò non sembra comportare particolari problemi, ma, come ulteriore esempio di tale fenomeno e facendo riferimento alla grammatica delle espressioni aritmetiche, consideriamo l'espressione $\mathbf{id} + \mathbf{id} * \mathbf{id}$. Possiamo generarla usando la produzione $\mathtt{E} \to \mathtt{E} * \mathtt{E}$, quindi applicando la produzione $\mathtt{E} \to \mathtt{E} + \mathtt{E}$ e, infine, applicando ripetutamente la produzione $\mathtt{E} \to \mathbf{id}$ per ottenere l'albero mostrato nella parte sinistra della Figura 7.2. Ma avremmo anche potuto iniziare applicando $\mathtt{E} \to \mathtt{E} + \mathtt{E}$, quindi $\mathtt{E} \to \mathtt{E} * \mathtt{E}$ e, infine, $\mathtt{E} \to \mathbf{id}$ per ottenere l'albero mostrato nella parte destra della figura. Questi due alberi sono chiaramente diversi: qual'è quello giusto?

Non possiamo rispondere a questa domanda se conosciamo solo la grammatica, in quanto entrambi gli alberi sono stati costruiti usando le sue produzioni e non vi è ragione perchè entrambi non possano essere accettabili. Una grammatica nella quale è possibile analizzare anche una sola sequenza in due o più modi diversi è detta **ambigua**: quest'ambiguità se non viene risolta in qualche modo è chiaramente inaccettabile per un compilatore. Sebbene non esistano metodi automatici per eliminare le ambiguità di una grammatica, esistono comunque due principali tecniche per trat-

Figura 7.4: albero di derivazione in una grammatica non ambigua.



tare questo problema. Facendo riferimento alla prima, quando guardiamo dal di fuori la grammatica delle espressioni e consideriamo le regole di precedenza degli operatori in JAVA e in molti altri linguaggi ad alto livello, vediamo che uno solo dei due alberi della Figura 7.2 può essere quello giusto. L'albero di sinistra, infatti, dice che l'espressione è un prodotto e che uno dei due fattori è una somma, mentre l'albero di destra dice che l'espressione è una somma e che uno dei due addendi è un prodotto: quest'ultima è la normale interpretazione dell'espressione $\mathbf{id} + \mathbf{id} * \mathbf{id}$. Se possiamo in qualche modo incorporare all'interno del parser il fatto che la moltiplicazione ha una maggiore priorità rispetto all'addizione, questo risolverà l'ambiguità.

La seconda alternativa è quella di riscrivere la grammatica in modo da eliminare le ambiguità. Per esempio, se distinguiamo tra *espressioni*, *termini* e *fattori*, possiamo definire la seguente grammatica alternativa:

$$\begin{array}{lll} E \to E + T & E \to E - T & E \to T \\ T \to T * F & T \to T/F & T \to F \\ F \to (E) & F \to \mathbf{id} & \end{array}$$

In altre parole, questa grammatica esplicita il fatto che, se vogliamo usare la somma o sottrazione di due addendi come fattore di una moltiplicazione o di una divisione, allora dobbiamo prima racchiudere tale somma o sottrazione tra parentesi. Facendo uso

di questa grammatica, il nostro esempio può essere analizzato in un solo modo come mostrato nella Figura 7.4 (considereremo di nuovo tale grammatica nel seguito).

7.2.2 Derivazioni destre e sinistre

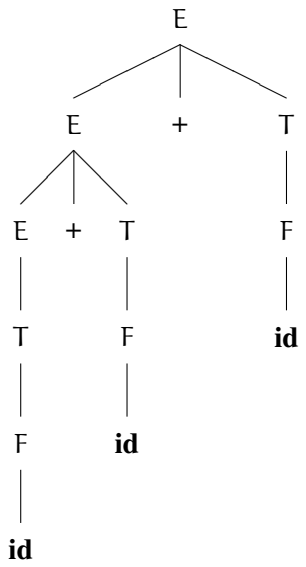
Consideriamo ancora la grammatica delle espressioni aritmetiche introdotta all'inizio di questo paragrafo e supponiamo di voler generare la sequenza $(\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})$. Come abbiamo già osservato, esistono diversi modi per poterlo fare. Uno di questi è la seguente derivazione:

$$\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{E}/\mathtt{E} \\
&\rightarrow \mathtt{E}/(\mathtt{E}) \\
&\rightarrow \mathtt{E}/(\mathtt{E}-\mathtt{E}) \\
&\rightarrow \mathtt{E}/(\mathtt{E}-\mathbf{id}) \\
&\rightarrow \mathtt{E}/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathtt{E})/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathtt{E}+\mathtt{E})/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathtt{E}+\mathbf{id})/(\mathbf{id}-\mathbf{id}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})
\end{aligned}$$

Osserviamo che, a ogni passo della derivazione precedente, abbiamo scelto il non terminale più a destra come quello da sostituire. Ad esempio, nella forma sentenziale $\mathtt{E}/(\mathtt{E}-\mathtt{E})$ avevamo tre scelte di $\mathtt{E}$ da poter sostituire e abbiamo scelto quella più a destra. Una tale derivazione è detta **derivazione destra**. Alternativamente, avremmo potuto selezionare il non terminale più a sinistra a ogni passo e avremmo ottenuto una **derivazione sinistra**, come quella seguente:

$$\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{E}/\mathtt{E} \\
&\rightarrow (\mathtt{E})/\mathtt{E} \\
&\rightarrow (\mathtt{E}+\mathtt{E})/\mathtt{E} \\
&\rightarrow (\mathbf{id}+\mathtt{E})/\mathtt{E} \\
&\rightarrow (\mathbf{id}+\mathbf{id})/\mathtt{E} \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathtt{E}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathtt{E}-\mathtt{E}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathtt{E}) \\
&\rightarrow (\mathbf{id}+\mathbf{id})/(\mathbf{id}-\mathbf{id})
\end{aligned}$$

Figura 7.5: l'albero di derivazione di **id** + **id** + **id**.



Notiamo che mentre è possibile costruire diverse derivazioni corrispondenti allo stesso albero di derivazione, le derivazioni sinistre e destre sono uniche. Ogni forma sentenziale che occorre in una derivazione sinistra è detta *forma sentenziale sinistra* e ogni forma sentenziale che occorre in una derivazione destra è detta *forma sentenziale destra*. La distinzione tra derivazioni sinistre e destre non è puramente accademica: esistono, infatti, due tipi di analizzatori sintattici, di cui un tipo genera derivazioni sinistre e un altro derivazioni destre, e le differenze tra questi due tipi incide direttamente sui dettagli della costruzione del parser e sulle sue operazioni. In queste dispense ci limiteremo ad analizzare nel prossimo paragrafo gli analizzatori del primo tipo, anche detti parser top-down.

7.3 Parser top-down

Un parser **top-down** parte dalla radice dell'albero di derivazione e cerca di ricostruire la crescita dell'albero che porta alla data sequenza di simboli terminali: nel fare ciò, ricostruisce una derivazione sinistra. Il parser top-down deve iniziare dalla radice

dell'albero e determinare in base alla sequenza di simboli terminali come far crescere l'albero di derivazione: inoltre, deve fare questo in base solo alla conoscenza delle produzioni nella grammatica e dei simboli terminali in arrivo da sinistra verso destra. Quest'approccio fa sorgere diversi problemi che analizzeremo e risolveremo, in questo paragrafo, man mano che si presenteranno fino a sviluppare gradualmente il metodo generale per la costruzione di un parser top-down.

7.3.1 Ricorsione sinistra

La scansione della sequenza di simboli terminali da sinistra a destra ci crea subito dei problemi. Supponiamo che la nostra grammatica sia la seguente:

$$\begin{array}{ll} \mathsf{E} \to \mathsf{E} + \mathsf{T} & \mathsf{E} \to \mathsf{T} \\ \mathsf{T} \to \mathsf{T} * \mathsf{F} & \mathsf{T} \to \mathsf{F} \\ \mathsf{F} \to (\mathsf{E}) & \mathsf{F} \to \mathbf{id} \end{array}$$

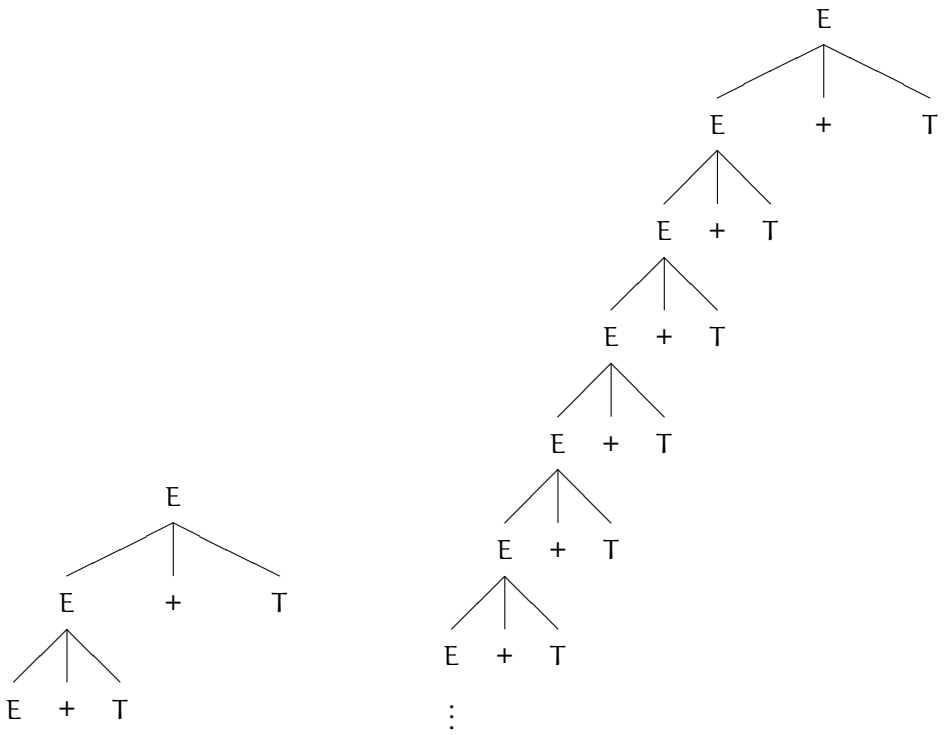
(si tratta di una versione semplificata della grammatica precedentemente introdotta per risolvere il problema delle ambiguità). Supponiamo di stare analizzando l'espressione $\mathbf{id} + \mathbf{id} + \mathbf{id}$, il cui albero di derivazione è mostrato nella Figura 7.5. Un parser top-down per questa grammatica partirà tentando di espandere E con la produzione $\mathsf{E} \to \mathsf{E} + \mathsf{T}$. Quindi tenterà di espandere il nuovo E nello stesso modo: questo ci darà l'albero mostrato nella parte sinistra della Figura 7.6 che finora è corretto. Ora sappiamo che la E più a sinistra dovrebbe essere sostituita mediante la regola $\mathsf{E} \to \mathsf{T}$ invece di usare di nuovo $\mathsf{E} \to \mathsf{E} + \mathsf{T}$. Ma come può l'analizzatore sintattico saperlo? Il parser non ha nulla per andare avanti se non la grammatica e la sequenza di simboli terminali in input e nulla nell'input è cambiato fino a ora. Per questo motivo, non vi è modo di evitare che il parser faccia crescere l'albero di derivazione indefinitamente come mostrato nella parte destra della figura.

In effetti, non esiste soluzione a questo problema finchè la grammatica è nella sua forma corrente. Produzioni della forma

$$\mathsf{A} \to \mathsf{A}\alpha$$

sono produzioni **ricorsive a sinistra** e nessun parser top-down è in grado di gestirle. Infatti, osserviamo che il parser procede "consumando" i simboli terminali: ognuno di tali simboli guida il parser nella sua scelta di azioni. Quando il simbolo terminale è usato, un nuovo simbolo terminale diviene disponibile e ciò porta il parser a fare una diversa mossa. I simboli terminali vengono consumati quando si accordano con i terminali nelle produzioni: nel caso di una produzione ricorsiva a sinistra, l'uso ripetuto di tale produzione non usa simboli terminali per cui, a ogni mossa nuova, il parser ha di fronte lo stesso simbolo terminale e quindi farà la stessa mossa.

Figura 7.6: il problema della ricorsione sinistra.



Questo problema affligge tutti i parser top-down comunque essi siano implementati. La soluzione consiste nel riscrivere la grammatica in modo che siano eliminate le ricorsioni sinistre. A tale scopo distinguiamo tra due tipi di tali ricorsioni: le ricorsioni sinistre **immediate** generate da produzioni del tipo

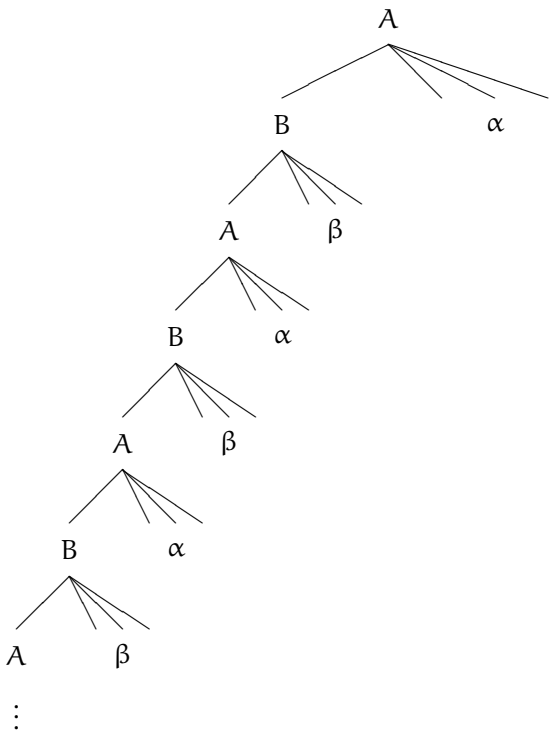
$$A \to A\alpha$$

e quelle **non immediate** generate da produzioni del tipo

$$A \to B\alpha \qquad B \to A\beta$$

In quest'ultimo caso, A produrrà $B\alpha$, B produrrà $A\beta$ e così via: il parser costruirà quindi l'albero mostrato nella Figura 7.7. Nel seguito, discuteremo anzitutto l'eliminazione delle ricorsioni dirette, poichè questa è in un certo senso l'operazione di

Figura 7.7: ricorsione sinistra non immediata.



base, facendo uso della quale potremo poi descrivere la tecnica generale che rimuove tutti i tipi di ricorsione sinistra.

Eliminazione di ricorsioni sinistre immediate

Per rimuovere le ricorsioni sinistre immediate, procediamo nel modo seguente (nel seguito assumiamo che la grammatica originale non contenga λ-produzioni, ovvero produzioni la cui parte destra sia uguale a λ). Per ogni non terminale A nella grammatica che presenta almeno una produzione ricorsiva sinistra immediata, eseguiamo le seguenti operazioni.

1. Separiamo le produzioni ricorsive a sinistra immediate dalle altre. Supponiamo che le produzioni siano le seguenti:

$$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \cdots$$

e

$$A \to \delta_1 \mid \delta_2 \mid \delta_3 \mid \cdots$$

2. Introduciamo un nuovo non terminale A'.
3. Sostituiamo ogni produzione non ricorsiva $A \to \delta_i$ con la produzione $A \to \delta_i A'$.
4. Sostituiamo ogni produzione ricorsiva immediata $A \to A\alpha_i$ con la produzione $A' \to \alpha_i A'$.
5. Aggiungiamo la produzione $A' \to \lambda$.

Osserviamo che la rimozione di ricorsioni immediate introduce produzioni del tipo $A \to \lambda$. L'idea dell'algoritmo appena descritto è quella di anticipare l'uso delle produzioni non ricorsive che in ogni caso dovranno essere prima o poi utilizzate: così facendo, le ricorsioni sinistre immediate vengono trasformate, mediante l'uso di un nuovo simbolo non terminale, in ricorsioni destre immediate, le quali non creano problemi a un parser di tipo top-down.

Esempio 7.2: eliminazione di ricorsioni sinistre immediate

Consideriamo la seguente grammatica:

$$S \to Sa \qquad S \to b$$

Tutte le derivazioni in questa grammatica hanno la forma

$$S \to Sa \to Saa \to Saaa \to \cdots \to ba^n$$

(il processo ha termine quando viene scelta la produzione $S \to b$). La grammatica trasformata è la seguente:

$$S \to bS' \qquad S' \to aS' \qquad S' \to \lambda$$

Tutte le derivazioni in questa grammatica hanno la forma

$$S \to bS' \to baS' \to baaS' \to baaaS' \to \cdots \to ba^n$$

(in questo caso il processo ha termine quando scegliamo $S' \to \lambda$). Come detto in precedenza, la procedura di eliminazione delle ricorsioni immediate ha anticipato l'uso della produzione non ricorsiva che in ogni caso doveva essere prima o poi utilizzata.

Osserviamo che l'eliminazione di ricorsioni immediate sinistre mediante l'introduzione di ricorsioni immediate destre può causare qualche problema quando si ha a che fare con operatori associativi a sinistra (come nel caso della sottrazione): questi problemi devono essere tenuti in considerazione al momento della produzione del codice intermedio.

Eliminazione di ricorsioni sinistre non immediate

Per rimuovere tutte le ricorsioni sinistre in una grammatica procediamo invece nel modo seguente.

1. Creiamo una lista ordinata $A_1, \ldots, A_m$ di tutti i simboli non terminali.
2. Per $j = 1, \ldots, m$, eseguiamo le seguenti operazioni:
   (a) per $h = 1, \ldots, j-1$, sostituiamo ogni produzione del tipo $A_j \to A_h\beta$ con l'insieme delle produzioni $A_j \to \gamma\beta$ per ogni produzione del tipo $A_h \to \gamma$ (facendo riferimento alle produzioni di A_h già modificate);
   (b) facendo uso della tecnica descritta in precedenza, rimuoviamo le eventuali ricorsioni sinistre immediate a partire da A_j (i nuovi non terminali che eventualmente vengono introdotti in questo passo non sono inseriti nella lista ordinata).

Esempio 7.3: eliminazione di ricorsioni sinistre non immediate

Consideriamo la seguente grammatica:

$$S \to aA \qquad S \to b \qquad S \to cS \qquad A \to Sd \qquad A \to e$$

Supponendo che S preceda A, abbiamo che le produzioni per S non richiedono alcuna modifica. Per il simbolo non terminale A, abbiamo una parte destra che inizia con S. Quindi sostituiamo tale produzione con le seguenti produzioni:

$$A \to aAd \qquad A \to bd \qquad A \to cSd$$

Quello che è successo è di avere inserito i passi iniziali di tutte le possibili derivazioni sinistre a partire da A tramite S nelle nuove parti destre. Infatti, a partire da A possiamo ottenere solo le seguenti forme sentenziali:

$$A \to Sd \to \begin{cases} aAd \to \cdots \\ bd \\ cSd \to \cdots \end{cases}$$

Nel riscrivere le produzioni di A, abbiamo semplicemente saltato il primo passo.

L'algoritmo appena descritto garantisce l'eliminazione di tutte le ricorsioni sinistre se si assume non solo che la grammatica non contenga λ-produzioni ma che non contenga nemmeno cicli, ovvero non sia possibile derivare il simbolo non terminale A a partire da A stesso. Il motivo per cui l'algoritmo funziona è che, all'iterazione j dell'algoritmo, ogni produzione del tipo $A_h \to A_i\eta$ con $h < j$ deve avere $i > h$ (in quanto A_h è già stato analizzato): pertanto, la produzione $A_j \to A_i\eta\beta$ che viene eventualmente aggiunta tra quelle in sostituzione di $A_j \to A_h\beta$ verrà successivamente elaborata durante la stessa iterazione e sostituita a sua volta con nuove produzioni. Prima o poi l'indice del simbolo non terminale che appare in prima posizione di una produzione la cui parte sinistra è A_j dovrà essere non inferiore a j: l'eliminazione delle ricorsioni sinistre immediate garantisce che, al termine dell'iterazione, quest'indice sarà strettamente maggiore di j. Il prossimo esempio mostra che l'algoritmo può funzionare anche nel caso la grammatica contenga λ-produzioni.

Esempio 7.4: eliminazione di ricorsioni sinistre non immediate con produzioni nulle

Consideriamo la seguente grammatica: $S \to Aa$, $S \to b$, $A \to Ac$, $A \to Sd$, $A \to \lambda$. Supponiamo che S preceda A. Analizzando il simbolo non terminale A, otteniamo le seguenti produzioni:

$$A \to Ac \qquad A \to Aad \qquad A \to bd \qquad A \to \lambda$$

Eliminando le ricorsioni sinistre immediate, otteniamo la nuova grammatica che non include alcuna ricorsione sinistra: $S \to Aa$, $S \to b$, $A \to bdA'$, $A \to A'$, $A' \to cA'$, $A' \to adA'$ e $A' \to \lambda$.

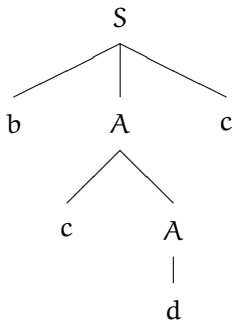
7.3.2 Backtracking

Un modo per sviluppare un parser top-down consiste semplicemente nel fare in modo che il parser tenti esaustivamente tutte le produzioni applicabili fino a trovare l'albero di derivazione corretto. Questo è talvolta detto il metodo della "forza bruta". Tale metodo può far sorgere tuttavia alcuni problemi. Per esempio, consideriamo la grammatica seguente:

$$S \to ee \qquad S \to bAc \qquad S \to bAe \qquad A \to d \qquad A \to cA$$

e la sequenza di simboli terminali $bcde$. Se il parser tenta tutte le produzioni esaustivamente, incomincerà considerando $S \to bAc$ poichè il b all'inizio dell'input impedisce di usare la produzione $S \to ee$. Il prossimo simbolo di input è c: questo elimina $A \to d$ e quindi viene tentata $A \to cA$. Proseguendo in questo modo, il parser genera l'albero mostrato nella Figura 7.8. Ma quest'albero è sbagliato! Esso genera la stringa $bcdc$ invece di $bcde$. Chiaramente ciò è dovuto al fatto che il parser è partito con

Figura 7.8: un esempio di derivazione sbagliata.

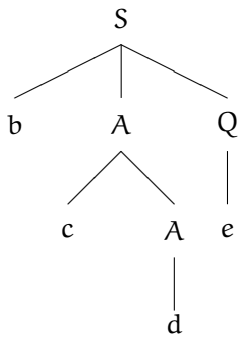


la parte destra sbagliata: se fosse stato capace di guardare in avanti al simbolo di input finale, non avrebbe fatto quest'errore. Purtroppo, i parser top-down scandiscono i simboli terminali da sinistra verso destra. Per rimediare al danno, dobbiamo tornare indietro (in inglese *backtrack*) fino a trovare una produzione alternativa: in questo caso, il parser deve ritornare alla radice e tentare la terza produzione $S \rightarrow \mathrm{b}A\mathrm{e}$.

Il backtrack è sostanzialmente una visita in profondità di un grafo. La ricerca procede in avanti da nodo a nodo finchè viene trovata la soluzione oppure viene raggiunto un vicolo cieco. Se questo è il caso, deve tornare indietro fino a trovare una biforcazione lungo la strada e tentare quella possibilità. Similmente, il metodo della forza bruta lavora in avanti di produzione in produzione fino al successo oppure fino a un vicolo cieco. Se questo è il caso, deve tornare indietro fino a trovare una biforcazione ovvero una produzione con una parte destra non ancora tentata.

Ma con il procedere dell'analisi, l'input viene consumato. Ad esempio, nel generare l'albero precedente quando viene scelta $S \rightarrow \mathrm{b}A\mathrm{c}$, il parser supera la b nella sequenza di input e quando sceglie $A \rightarrow \mathrm{c}A$ supera la c. Quindi, una volta constatato di essere giunto a un vicolo cieco, non solo deve demolire l'albero sbagliato ma deve anche tornare indietro nell'input al simbolo terminale che stava esaminando quando ha sbagliato strada. In alcuni casi, questo è relativamente facile, in altri può essere proibitivo. Se l'analisi lessicale viene eseguita come un passo separato e l'intero programma è stato ridotto in forma di token, allora può essere semplice tornare indietro lungo la lista di token. Se invece l'analizzatore lessicale è sotto il controllo del parser e crea i token man mano che si procede, tornare indietro può essere difficile perchè il lettore deve tornare indietro al punto corrispondente della sequenza in input. Questi problemi possono tutti essere risolti ma rallentano le operazioni del compilatore.

Figura 7.9: albero di derivazione per la grammatica fattorizzata.



L'operazione è particolarmente lenta se il codice sorgente contiene un errore poiché il compilatore deve tornare indietro ripetutamente per tentare tutte le possibilità e vedere che tutte falliscono, prima di concludere che l'input è sbagliato. Per questi motivi, ogni metodo basato sul backtrack non è molto attraente.

Il problema è in parte nello sviluppo del parser e in parte nello sviluppo del linguaggio. L'esempio che abbiamo usato aveva una produzione che sembrava promettente ma che aveva una trappola alla fine. Osserviamo quanti meno problemi avremmo con la seguente grammatica:

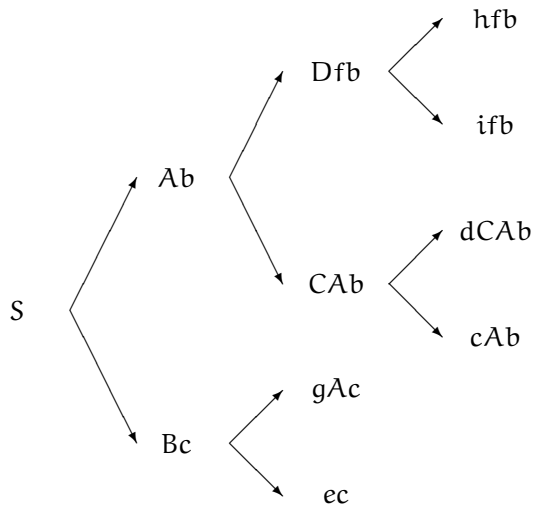
$$S \to ee \qquad S \to \mathrm{b}AQ \qquad Q \to \mathrm{c} \qquad Q \to e \qquad A \to \mathrm{d} \qquad A \to \mathrm{c}A$$

In questo caso abbiamo fattorizzato il prefisso comune $\mathrm{b}A$ e usato un nuovo non terminale per permettere la scelta finale tra c ed e. Questa grammatica genera lo stesso linguaggio di quella precedente ma il parser può ora generare l'albero di derivazione giusto senza backtrack come mostrato nella Figura 7.9. La trasformazione appena mostrata è nota come fattorizzazione sinistra ed è la prima di tante tecniche che rendono i parser top-down praticabili: un altro importante modo di evitare il backtrack è quello di cui parleremo nel prossimo paragrafo.

7.3.3 Parser predittivi

Se nella grammatica al più una produzione inizia con un simbolo non terminale, allora la strategia del parser potrebbe essere facile: tenta le parti destre che iniziano con un terminale e se fallisce tenta quella che inizia con il non terminale. Ma supponiamo

Figura 7.10: terminali derivabili dal simbolo iniziale.



di avere la grammatica seguente:

$$S \to Ab \qquad S \to Bc \qquad A \to Df \qquad A \to CA$$

$$B \to gA \qquad B \to \qquad C \to dC \qquad C \to c \qquad D \to h \qquad D \to i$$

In questo caso le parti destre delle produzioni da S e A non cominciano con simboli terminali. Di conseguenza il parser non ha una guida immediata dalla sequenza di input. Ad esempio, se la sequenza di input è gchfc, un parser deve sperimentare e fare molto backtrack prima di trovare la seguente derivazione: $S \to Bc \to gAc \to gCAc \to gcAc \to gcDfc \to gchfc$. Questo esempio non è irrealistico: frequentemente le grammatiche hanno diverse produzioni le cui parti destre iniziano con simboli non terminali.

Il backtrack potrebbe essere evitato se il parser avesse la capacità di guardare avanti nella grammatica in modo da anticipare quali simboli terminali sono derivabili (mediante derivazioni sinistre) da ciascuno dei vari simboli non terminali nelle parti destre delle produzioni. Per esempio, se seguiamo le parti destre di S, considerando tutte le possibili derivazioni sinistre, troviamo le possibilità mostrate nella Figura 7.10. Da questa figura vediamo che se la sequenza di input inizia con c, d, h

oppure i dobbiamo scegliere $S \to Ab$ mentre se inizia con e oppure g dobbiamo scegliere $S \to Bc$. Se inizia con qualcosa di diverso, dobbiamo dichiarare un errore. La figura ci dice infatti quali simboli terminali possono iniziare forme sentenziali derivabili da Ab e quali terminali possono iniziare sequenze derivabili da Bc: questi insiemi sono noti come insiemi FIRST e sono generalemente indicati come $\text{FIRST}(Ab)$ (che è uguale a $\{c, d, h, i\}$) e $\text{FIRST}(Bc)$ (che è uguale a $\{e, g\}$). Data questa informazione, il parser tenterà la produzione $S \to Ab$ se il simbolo terminale in input appartiene a $\text{FIRST}(Ab)$ e la produzione $S \to Bc$ se appartiene a $\text{FIRST}(Bc)$. Se tale simbolo terminale non appartiene a $\text{FIRST}(S) = \text{FIRST}(Ab) \cup \text{FIRST}(Bc)$, allora la sequenza è grammaticalmente scorretta e può essere rifiutata.

Data una grammatica $G = (V, N, S, P)$, possiamo in generale definire formalmente la funzione $\text{FIRST} : (V \cup N)^+ \to 2^V$ come segue: se α è una sequenza di simboli terminali e non terminali e se X è l'insieme di tutte le forme sentenziali derivabili da α mediante derivazioni sinistre, allora, per ogni $\beta \in X$ che inizia con un terminale x, x appartiene a $\text{FIRST}(\alpha)$. Per convenzione, inoltre, assumiamo che se la stringa λ è generabile a partire da α, allora $\lambda \in \text{FIRST}(\alpha)$.

In grammatiche di dimensione moderata, gli insiemi FIRST possono essere trovati a mano in modo simile a quanto fatto nella Figura 7.10. In generale, tuttavia, dovremo definire un metodo di calcolo di tali insiemi. A tale scopo, osserviamo che dalla definizione di FIRST seguono le seguenti proprietà:

1. se α inizia con un terminale x, allora $\text{FIRST}(\alpha) = x$;
2. $\text{FIRST}(\lambda) = \{\lambda\}$;
3. se α inizia con un non terminale A, allora $\text{FIRST}(\alpha)$ include $\text{FIRST}(A) - \{\lambda\}$.

La terza proprietà contiene una trappola nascosta. Supponiamo che α sia $AB\delta$ e che sia possibile generare λ a partire da A. Allora, per calcolare $\text{FIRST}(\alpha)$, dobbiamo anche seguire le possibilità a partire da B. Inoltre, se è possibile generare λ anche a partire da B, allora dobbiamo anche seguire le possibilità a partire da δ.

Esempio 7.5: simboli non terminali annullabili

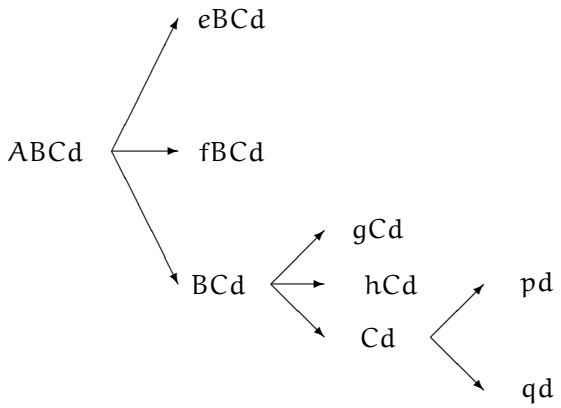
Supponiamo che la grammatica G includa le seguenti produzioni:

$$S \to ABCd \qquad A \to e \qquad A \to f \qquad A \to \lambda$$

$$B \to g \qquad B \to h \qquad B \to \lambda \qquad C \to p \qquad C \to q$$

e che vogliamo trovare $\text{FIRST}(S) = \text{FIRST}(ABCd)$. Esplorando questa forma sentenziale, troviamo le possibilità mostrate nella Figura 7.11. Quindi $\text{FIRST}(ABCd) = \{e, f, g, h, p, q\}$.

Figura 7.11: simboli terminali annullabili e funzione FIRST.



Se è possibile generare λ a partire da un non terminale A, diciamo che A è **annullabile** (nell'esempio precedente sia A che B sono annullabili). Osserviamo che sebbene $\text{FIRST}(A)$ e $\text{FIRST}(B)$ includano λ, questo non è vero per $\text{FIRST}(ABCd)$: in effetti, $\text{FIRST}(\alpha)$ include λ solo se è possibile generare λ a partire da α e ciò può accadere solo se ogni elemento di α è annullabile.

Questo problema non sorge molto spesso, ma se vogliamo programmare la procedura di calcolo degli insiemi FIRST dobbiamo risolverlo. A tale scopo, supponiamo che α è del tipo $\beta X\delta$ dove β è una sequenza di zero o più simboli non terminali annullabili, X è un terminale oppure un non terminale non annullabile, e δ è tutto quello che segue. Allora $\text{FIRST}(\alpha) = (\text{FIRST}(\beta) - \{\lambda\}) \cup \text{FIRST}(X)$ (se tutto in α è annullabile, ovvero $\alpha = \beta$, allora $\text{FIRST}(\alpha) = \text{FIRST}(\beta)$).

Possiamo ora definire un algoritmo per il calcolo della funzione FIRST distinguendo i seguenti due casi.

1. α è un singolo carattere oppure $\alpha = \lambda$.
   (a) Se α è un terminale y allora $\text{FIRST}(\alpha) = y$.
   (b) Se $\alpha = \lambda$ allora $\text{FIRST}(\alpha) = \{\lambda\}$.
   (c) Se α è un non terminale A e $A \to \beta_i$ sono le produzioni a partire da A, per $i = 1, \ldots, k$, allora

$$\text{FIRST}(\alpha) = \bigcup_k \text{FIRST}(\beta_k).$$

2. $\alpha = X_1 X_2 \cdots X_n$ con $n > 1$.
   (a) Poniamo $\mathrm{FIRST}(\alpha) = \emptyset$ e $j = 1$.
   (b) Poniamo $\mathrm{FIRST}(\alpha) = \mathrm{FIRST}(\alpha) \cup \mathrm{FIRST}(X_j) - \{\lambda\}$.
   (c) Se X_j è annullabile e $j < n$, allora poniamo $j = j + 1$ e ripetiamo il passo precedente.
   (d) Se $j = n$ e X_n è annullabile, allora includiamo λ in $\mathrm{FIRST}(\alpha)$.

Osserviamo come i due casi sopra descritti siano mutuamente ricorsivi.

Esempio 7.6: calcolo della funzione FIRST

Considerando l'esempio precedente, si ha che nel caso di $\alpha = ABCd$, C è il primo non terminale non annullabile. Quindi abbiamo che

$$\begin{aligned} \mathrm{FIRST}(ABCd) &= \{e, f\} \quad (\text{ovvero } \mathrm{FIRST}(A) - \{\lambda\}) \\ &\cup \{g, h\} \quad (\text{ovvero } \mathrm{FIRST}(B) - \{\lambda\}) \\ &\cup \{p, q\} \quad (\text{ovvero } \mathrm{FIRST}(C)) \\ &= \{e, f, g, h, p, q\} \end{aligned}$$

in accordo con quanto avevamo trovato in precedenza esplorando le produzioni a mano.

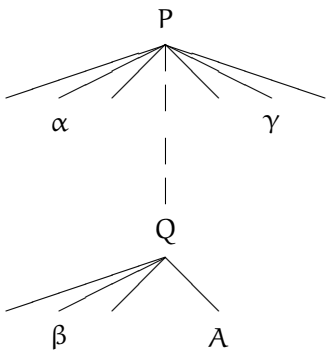
I parser che usano gli insiemi FIRST sono detti parser **predittivi**, in quanto hanno la capacità di vedere in avanti e prevedere che il primo passo di una derivazione ci darà prima o poi un certo simbolo terminale.

Grammatiche LL(1)

La tecnica basata sul calcolo della funzione FIRST non sempre può essere utilizzata. Talvolta la struttura della grammatica è tale che il simbolo terminale successivo non ci dice quale parte destra utilizzare (ad esempio, nel caso in cui $\mathrm{FIRST}(\alpha)$ e $\mathrm{FIRST}(\beta)$ non siano disgiunti). Inoltre, quando le grammatiche acquisiscono λ-produzioni come risultato della rimozione della ricorsione sinistra, gli insiemi FIRST non ci dicono quando scegliere $A \to \lambda$. Per gestire questi casi, abbiamo bisogno di una seconda funzione, FOLLOW. Nel definire tale funzione, assumeremo che una sequenza di simboli terminali, prima di essere passata al parser, abbia un segno di demarcazione appeso, che indicheremo con \$.

Data una grammatica $G = (V, N, S, P)$, definiamo la funzione $\mathrm{FOLLOW}: N \to 2^V$ come segue: se A è un simbolo non terminale, allora, per ogni $x \in V$ che può seguire A in una forma sentenziale, x appartiene a $\mathrm{FOLLOW}(A)$. Per convenzione, inoltre,

Figura 7.12: calcolo della funzione FOLLOW.



assumiamo che se A può apparire come ultimo simbolo di una forma sentenziale, allora $\$ \in \text{FOLLOW}(A)$. Per ogni non terminale A, possiamo calcolare $\text{FOLLOW}(A)$ nel modo seguente.

1. Se A è il simbolo iniziale, allora includiamo \$ in $\text{FOLLOW}(A)$.
2. Cerchiamo attraverso la grammatica le occorrenze di A nelle parti destre delle produzioni. Sia $Q \to xAy$ una di queste produzioni. Distinguiamo i seguenti tre casi:
   (a) se y inizia con un terminale q, allora includiamo q in $\text{FOLLOW}(A)$;
   (b) se y non inizia con un terminale, allora includiamo $\text{FIRST}(y) - \{\lambda\}$ in $\text{FOLLOW}(A)$;
   (c) se $y = \lambda$ (ovvero A è in fondo) oppure se y è annullabile, allora includiamo $\text{FOLLOW}(Q)$ in $\text{FOLLOW}(A)$.

Osserviamo che se la prima regola si applica, non possiamo fermarci ma dobbiamo procedere e applicare la seconda regola. Inoltre notiamo che escludiamo la stringa vuota λ perchè essa non non apparirà mai esplicitamente.

L'ultima regola richiede qualche spiegazione. Anzitutto, osserviamo che se A è alla fine della parte destra, questo significa che A può venire alla fine di una forma sentenziale derivabile da Q. Inoltre, se $Q \to \beta AB$ e B è annullabile, anche in questo caso A può venire alla fine di una forma sentenziale derivabile da Q. In questi casi, cosa può venire dopo A? Per rispondere consideriamo da dove viene Q stesso e per

semplicità supponiamo che $B = \lambda$, come mostrato nella Figura 7.12. Se guardiamo in alto l'albero, vediamo che a partire da S abbiamo prima prodotto la forma sentenziale $\alpha Q \gamma$ e quindi la forma sentenziale $\alpha\beta A\gamma$. Quindi quello che viene dopo A è γ, che è esattamente ciò che viene dopo Q: per cui γ deve avere contribuito a $\mathsf{FOLLOW}(Q)$. Questo è il motivo per cui tutto ciò che è in $\mathsf{FOLLOW}(Q)$ deve anche stare in $\mathsf{FOLLOW}(A)$. Nel caso particolare in cui la parte sinistra sia anche A, ovvero $A \to xA$, ignoriamo questa regola poichè non aggiungerebbe nulla di nuovo.

Esempio 7.7: la grammatica delle espressioni

Consideriamo la grammatica delle espressioni che qui riformuliamo dopo avere eliminato le ricorsioni sinistre.

$$\begin{array}{lllll} E \to TQ & Q \to +TQ & Q \to -TQ & Q \to \lambda & T \to FR \\ R \to *FR & R \to /FR & R \to \lambda & F \to (E) & F \to \textbf{id} \end{array}$$

In questo caso , è facile verificare che gli insiemi FIRST e FOLLOW sono i seguenti:

$$\mathsf{FIRST}(E) = \mathsf{FIRST}(F) = \mathsf{FIRST}(T) = \mathsf{FIRST}(TQ) = \mathsf{FIRST}(FR) = \{(, \textbf{id}\}$$

$$\mathsf{FIRST}(Q) = \{+, -, \epsilon\} \qquad \mathsf{FIRST}(R) = \{*, /, \epsilon\} \qquad \mathsf{FIRST}(+TQ) = \{+\}$$

$$\mathsf{FIRST}(-TQQ) = \{-\} \qquad \mathsf{FIRST}(*RF) = \{*\} \qquad \mathsf{FIRST}(/RF) = \{/\}$$

$$\mathsf{FOLLOW}(E) = \mathsf{FOLLOW}(Q) = \{\$,)\} \qquad \mathsf{FOLLOW}(F) = \{+, -, *, /, \$,)\}$$

$$\mathsf{FOLLOW}(T) = \mathsf{FOLLOW}(R) = \{+, -, \$,)\}$$

In un parser predittivo, l'insieme FOLLOW ci dice quando usare le λ-produzioni. Supponiamo di dover espandere un non terminale A. Inizialmente vediamo se il simbolo terminale in arrivo appartiene all'insieme FIRST di una parte destra di una produzione la cui parte sinistra è A. Se così non è questo normalmente vuol dire che si è verificato un errore. Ma se una delle produzioni da A è $A \to \lambda$, allora dobbiamo vedere se il simbolo terminale appartiene a $\mathsf{FOLLOW}(A)$. Se così è, allora può non trattarsi di un errore e $A \to \lambda$ è la produzione scelta.

Grammatiche per cui questa tecnica può essere usata sono note come **grammatiche LL(1)** e i parser che usano questa tecnica sono detti *parser LL(1)*. In questa notazione, la prima L sta per "left" per indicare che la scansione è da sinistra a destra, la seconda L sta per "left" per indicare che la derivazione è sinistra, e '(1)' indica che si guarda in avanti di un carattere. Le grammatiche LL(1) assicurano che guardando un carattere in avanti il token in arrivo determina univocamente quale parte destra scegliere (nelle grammatiche LL(2) bisognerebbe guardare a coppie di simbo-

li terminali). Perchè una grammatica sia LL(1) richiediamo che per ogni coppia di produzioni $A \to \alpha$ e $A \to \beta$, valgano le seguenti due proprietà:

1. $\mathsf{FIRST}(\alpha) - \{\lambda\}$ e $\mathsf{FIRST}(\beta) - \{\lambda\}$ siano disgiunti;
2. se α è annullabile, allora $\mathsf{FIRST}(\beta)$ e $\mathsf{FOLLOW}(A)$ devono essere disgiunti.

Se la prima regola è violata, allora ogni simbolo terminale in $\mathsf{FIRST}(\alpha) \cap \mathsf{FIRST}(\beta)$ non riuscirà a dirci quale parte destra scegliere. Se la regola 2 è violata, allora ogni token in $\mathsf{FIRST}(\beta) \cap \mathsf{FOLLOW}(A)$ non riuscirà a dirci se scegliere tra β oppure λ (ottenibile a partire da α).

7.4 Costruzione di un parser predittivo

Se possiamo evitare il backtrack, allora vi sono diversi modi di implementare il parser. Uno è quello di racchiudere ogni insieme di produzioni a partire da un simbolo non terminale in una funzione Booleana. Vi è una tale funzione per ogni non terminale della grammatica. La funzione tenterà ciascuna parte destra fino a che una corrispondenza non viene trovata. In tal caso ritorna il valore `true`, altrimenti ritorna il valore `false`. Nel caso di un parser LL(1) ciascuna di queste funzioni dovrà scegliere la parte destra in base agli insiemi FIRST e FOLLOW. La rimozione delle ricorsioni sinistre e l'utilizzo degli insiemi FIRST e FOLLOW rendono un tale parser praticabile. Il problema è che dobbiamo scrivere una funzione per ogni produzione. Se sorge qualche problema che rende necessario un cambiamento della grammatica, dobbiamo riprogrammare una o più di queste funzioni.

Una forma più conveniente di parser predittivo consiste di una semplice procedura di controllo che utilizza una tabella. Parte dell'attrattiva di questo approccio è che la procedura di controllo è generale: se la grammatica va cambiata, solo la tabella deve essere riscritta creando meno problemi che la riprogrammazione. La tabella può essere costruita a mano per piccole grammatiche o mediante il calcolatore per grammatiche grandi. La tabella della versione non ricorsiva dice quale parte destra scegliere e i simboli terminali sono usati nel modo naturale.

Il nostro esempio sarà un parser per l'espressioni aritmetiche, usando la grammatica vista nell'esempio precedente, ovvero la grammatica delle espressioni aritmetiche ottenuta dopo aver avere eliminato le ricorsioni sinistre:

$$
\begin{array}{lll}
\mathsf{E} \to \mathsf{TQ} & & \\
\mathsf{Q} \to +\mathsf{TQ} & \mathsf{Q} \to -\mathsf{TQ} & \mathsf{Q} \to \lambda \\
\mathsf{T} \to \mathsf{FR} & & \\
\mathsf{R} \to *\mathsf{FR} & \mathsf{R} \to /\mathsf{FR} & \mathsf{R} \to \lambda \\
\mathsf{F} \to (\mathsf{E}) & \mathsf{F} \to \textbf{id} &
\end{array}
$$

È più facile vedere prima la tabella e come viene utilizzata, per poi mostrare come costruirla. In particolare, la tabella per questa grammatica è la seguente (gli spazi bianchi indicano condizioni di errore):

	id	+	-	*	/	(	)	$
E	TQ					TQ		
Q		+TQ	−TQ				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	*FR	/FR		ϵ	ϵ
F	**id**					(E)		

Intuitivamente, ogni elemento non vuoto della tabella indica quale produzione debba essere scelta dal parser trovandosi a dover espandere il simbolo non terminale che etichetta la riga della tabella e leggendo in input il simbolo terminale che etichetta la colonna della tabella.

Pertanto, nel caso dei parser guidati da una tabella siffatta, è utile mantenere una pila che viene usata nel modo seguente.

- Inseriamo $ nella pila e alla fine della sequenza e inseriamo il simbolo iniziale nella pila.
- Fintantoché la pila non è vuota
  - Sia x l'elemento in cima alla pila e a il simbolo terminale in input.
  - Se $x \in V$ allora:
    - se $x = a$ allora estraiamo x dalla pila e avanziamo di un simbolo terminale, altrimenti segnaliamo un errore.
  - Se $x \notin V$, allora:
    - se $\texttt{table}[x, a]$ non è vuoto, allora estraiamo x dalla pila e inseriamo $\texttt{table}[x, a]$ nella pila in *ordine inverso*, altrimenti segnaliamo un errore.

Notiamo che, nell'ultimo caso, inseriamo la parte destra di una produzione in ordine inverso in modo che il simbolo più a sinistra sarà in cima alla pila pronto per essere eventualmente espanso o cancellato. Ciò è dovuto al fatto che il parser sta cercando di generare una derivazione sinistra, in cui il simbolo non terminale più a sinistra è quello da espandere e al fatto che la pila è una struttura dati che consente l'inserimento e l'estrazione di un elemento dallo stesso punto di accesso.

Esempio 7.8: analisi di un'espressione aritmetica

Supponiamo che la sequenza in input sia **(id+id)*id**. Lo stato iniziale sarà il seguente:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$		

La sequenza ha il simbolo $ appeso in fondo e la pila ha tale simbolo e il simbolo iniziale inseriti (scriviamo il contenuto della pila in modo che cresca da sinistra verso destra). A questo punto l'analizzatore entra nel ciclo principale. Il simbolo in cima alla pila è E e $table[E, (] = TQ$: quindi la nostra produzione è $E \to TQ$ e abbiamo

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$		

In questo caso, E è stato estratto dalla pila e la parte destra TQ inserita nella pila in ordine inverso. Ora abbiamo un non terminale T in cima alla pila e il simbolo terminale in input è ancora (. Quindi abbiamo $table[T, (] = FR$ e la produzione è $T \to FR$:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$	$T \to FR$	$\to FRQ$
$QRF	**(id+id)*id**$		

Proseguendo abbiamo $table[F, (] = (E)$ e la produzione è $F \to (E)$:

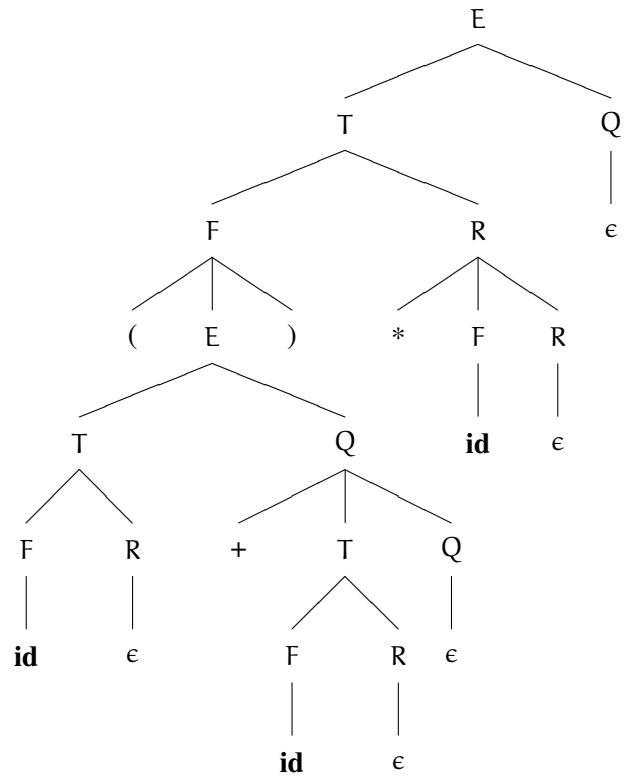
PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$	$T \to FR$	$\to FRQ$
$QRF	**(id+id)*id**$	$F \to (E)$	$\to (E)RQ$
$QR)E(	**(id+id)*id**$		

Ora abbiamo un terminale in cima alla pila. Lo confrontiamo con il simbolo in input e, poichè coincidono, estraiamo il simbolo terminale dalla pila e ci spostiamo al prossimo simbolo della sequenza in input:

PILA	INPUT	PRODUZIONE	DERIVAZIONE
$E	**(id+id)*id**$	$E \to TQ$	$E \to TQ$
$QT	**(id+id)*id**$	$T \to FR$	$\to FRQ$
$QRF	**(id+id)*id**$	$F \to (E)$	$\to (E)RQ$
$QR)E(	**(id+id)*id**$		
$QR)E	**id+id)*id**$		

Così andando avanti si arriva a consumare l'intero input con la pila vuota e possiamo annunciare il successo dell'analisi. Possiamo costruire l'albero a partire dalle produzioni lette nell'ordine in cui appaiono come mostrato nella Figura 7.13.

Figura 7.13: albero di derivazione generato dal parser predittivo.



Costruzione della tabella del parser predittivo

Ricordiamo che in un parser top-down, se non vi deve essere backtrack, allora il simbolo terminale in arrivo deve sempre dirci cosa fare. La stessa cosa si applica alla costruzione della tabella. Supponiamo che abbiamo X in cima alla pila e che a sia il simbolo terminale in arrivo. Vogliamo selezionare una parte destra che incominci con a oppure che possa portare a una forma sentenziale che inizi con a.

Per esempio, all'inizio del nostro esempio, avevamo E nella pila e (come input. Avevamo bisogno di una produzione della forma $E \to (\cdots$ Ma una tale produzione non esiste nella grammatica. Poichè non era disponibile, avremmo dovuto tracciare un cammino di derivazione che ci conducesse a una forma sentenziale che iniziasse

con (. L'unico tale cammino è

$$E \to TQ \to FRQ \to (E)RQ$$

e se guardiamo alla tabella vediamo che essa contiene esattamente la parte destra che determina il primo passo del cammino.

Ciò ci sta conducendo verso un terreno familiare: vogliamo selezionare una parte destra α se il token appartiene a $\mathrm{FIRST}(\alpha)$; quindi per una riga A e una produzione $A \to \alpha$, la tabella deve avere la parte destra α in ogni colonna etichettata con un terminale in $\mathrm{FIRST}(\alpha)$. Ciò funzionerà in tutti i casi eccetto quello in cui $\mathrm{FIRST}(\alpha)$ include λ poichè la tabella non ha una colonna etichettata λ. Per questi casi, seguiamo gli insiemi FOLLOW.

La regola per costruire la tabella è dunque la seguente.

- Visita tutte le produzioni. Sia $X \to \beta$ una di esse.
  - Per tutti i terminali a in $\mathrm{FIRST}(\beta)$, poniamo $\mathtt{table}[X, a] = \beta$.
  - Se $\mathrm{FIRST}(\beta)$ include λ, allora, per ogni $a \in \mathrm{FOLLOW}(X)$, $\mathtt{table}[X, a] = \lambda$.

Nel caso della grammatica delle espressioni matematiche, usando gli insiemi FIRST e FOLLOW precedentemente calcolati e le regole sopra descritte, otteniamo la tabella mostrata in precedenza. Infine, se la grammatica è di tipo LL(1) siamo sicuri che la tabella non conterrà elementi multipli.

Esercizi

Esercizio 7.1. Definire una grammatica libera da contesto che generi il seguente linguaggio

$$L = \{a^n b^m : n > m+2\}.$$

Esercizio 7.2. Definire una grammatica libera da contesto che generi il seguente linguaggio:

$$L = \{a^n b^n c^m : n > 0, m > 0\}.$$

Esercizio 7.3. Definire una grammatica libera da contesto che generi il linguaggio formato da sequenze palindrome, ovvero

$$L = \{xax^r : x \in (0|1)^* \wedge a \in (0|1|\epsilon)\}$$

dove x^r indica la sequenza ottenuta invertendo la sequenza x (ad esempio, $abc^r = cba$).
Esercizio 7.4. Dato un linguaggio L sull'alfabeto $\{0,1\}$, definiamo

$$\mathrm{Init}(L) = \{z \in \{0,1\}^* : (\exists w \in \{0,1\}^*)[zw \in L]\}.$$

Dimostrare che se L è libero da contesto allora anche $\mathrm{Init}(L)$ è libero da contesto.
Esercizio 7.5. Si consideri la grammatica

$$S \to \mathbf{a}S\mathbf{b}S \qquad S \to \mathbf{b}S\mathbf{a}S \qquad S \to \lambda$$

Quanti differenti alberi di derivazione esistono per la sequenza **abab**? Mostrare le derivazioni sinistre e destre.
Esercizio 7.6. Quali delle due seguenti grammatiche sono LL(1)? Giustificare la risposta.

1. $S \to ABBA \quad A \to \mathbf{a} \quad A \to \lambda \quad B \to \mathbf{b} \quad B \to \lambda$
2. $S \to \mathbf{a}S\mathbf{e} \quad S \to B \quad B \to \mathbf{b}B\mathbf{e} \quad B \to C \quad C \to \mathbf{c}C\mathbf{e} \quad B \to \mathbf{d}$

Esercizio 7.7. Dimostrare che ogni linguaggio riconosciuto da un automa a stati finiti può essere generato da una grammatica LL(1).
Esercizio 7.8. Dimostrare che l'eliminazione delle ricorsioni sinistre e la fattorizzazione non garantiscono che la grammatica ottenuta sia LL(1).
Esercizio 7.9. Progettare una tabella di parsing LL(1) per la grammatica

$$E \to -E \quad E \to (E) \quad E \to VT \quad T \to -E \quad T \to \lambda \quad V \to \mathbf{i}U \quad U \to (E) \quad U \to \lambda$$

Tracciare quindi la sequenza di parsing con input **i–i((i))** e con input **i-(()i)**.
Esercizio 7.10. Dimostrare che la seguente grammatica

$$S \to \mathbf{a}B \quad S \to \mathbf{a}C \quad S \to C \quad B \to \mathbf{b}B \quad B \to \mathbf{d} \quad C \to C\mathbf{c}B \quad C \to B\mathbf{b}B \quad C \to B$$

non è LL(1). Costruire una grammatica LL(1) equivalente alla precedente. Progettare quindi una tabella di parsing LL(1) per tale nuova grammatica e tracciare la sequenza di parsing con input **abdbd** e con input **abcbd**.

Capitolo 8

Automi a pila

SOMMARIO

In quest'ultimo capitolo di questa parte delle dispense, introdurremo un risultato, simile al pumping lemma per i linguaggi regolari, che consente di dimostrare che un linguaggio non è generabile da una grammatica libera da contesto. Infine, concluderemo introducendo il modello di calcolo equivalente alle grammatiche libere da contesto, ovvero gli automi a pila non deterministici, completando in tal modo la caratterizzazione della gerarchia di Chomsky in termini di modelli di calcolo.

8.1 Pumping lemma per linguaggi liberi da contesto

Come nel caso dei linguaggi generati da grammatiche regolari, è naturale chiedersi se esistano linguaggi che non sono di tipo 2 ma che siano di tipo 1. La risposta a tale domanda è affermativa e si basa sulla dimostrazione di un risultato simile al Lemma 6.1. Tale dimostrazione è leggermente più complicata di quella del Lemma 6.1, ma l'idea di base è la stessa: una stringa sufficientemente lunga di un linguaggio libero da contesto contiene al suo interno delle sotto-stringhe che possono essere ripetute in modo arbitrario fornendo un'altra stringa del linguaggio.

Lemma 8.1

Se L è un linguaggio infinito di tipo 2, allora esiste un numero intero $n_L > 0$, tale che, per ogni stringa $x \in L$ con $|x| > n_L$, x può essere decomposta nella concatenazione di cinque stringhe y, u, w, v e z per cui valgono le seguenti affermazioni: $|uv| > 0$, $|uwv| < n_L$ e, per ogni $i \geqslant 0$, $yu^i wv^i z \in L$.

Dimostrazione. Sia L è un linguaggio infinito libero da contesto. Poiché L è infinito, l'insieme dei simboli non terminali e quello dei simboli terminali sono finiti e la

parte destra di una regola di produzione è una stringa di lunghezza finita, deve allora esistere una stringa x sufficientemente lunga tale che un qualunque suo albero di derivazione abbia un altezza tale che, in almeno un cammino di tale albero, lo stesso simbolo non terminale A appaia almeno due volte (si veda la parte sinistra della Figura 8.1): come mostrato nella figura, x può essere decomposta nella concatenazione di cinque stringhe y, u, w, v e z. Per ottenere un albero di derivazione della stringa ywz è sufficiente sostituire al sotto-albero radicato nella prima occorrenza di A il sotto-albero radicato nella sua seconda occorrenza (si veda la parte centrale della figura). Al contrario, per ottenere un albero di derivazione della stringa yu^2wv^2z possiamo sostituire al sotto-albero radicato nella seconda occorrenza di A il sotto-albero radicato nella sua prima occorrenza (si veda la parte destra della figura): ripetendo tale sostituzione possiamo ottenere un albero di derivazione della stringa yu^iwv^iz, per ogni $i > 2$, e il lemma risulta essere dimostrato. ◇

Anche in questo caso come in quello dei linguaggi regolari, il risultato precedente viene comunemente utilizzato per dimostrare che un determinato linguaggio *non* è libero da contesto.

Esempio 8.1: un linguaggio non libero da contesto

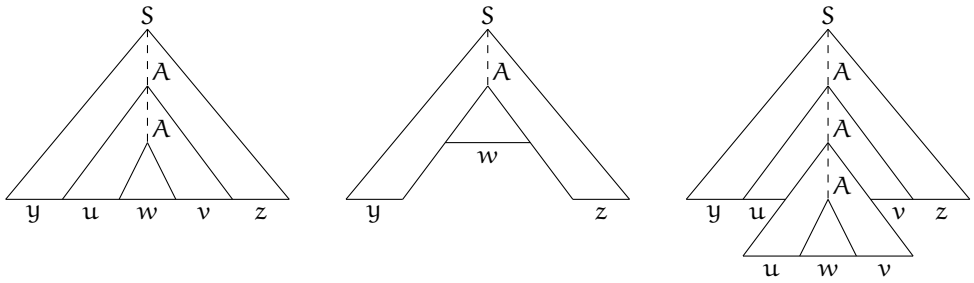
Consideriamo il linguaggio L costituito da tutte e sole le stringhe del tipo $0^n1^n2^n$, per $n > 0$. Chiaramente, L è infinito. Quindi, se L fosse libero da contesto, esisterebbe il numero intero $n_L > 0$ del Lemma 8.1. Consideriamo la stringa $x = 0^{n_L}1^{n_L}2^{n_L} \in L$: in base al lemma, abbiamo che $x = yuwvz$, con $|uv| > 0$, e che $yu^2wv^2z \in L$. Distinguiamo i seguenti due casi.

- Sia u che v contengono un solo tipo di simbolo. In tal caso, deve esistere un simbolo in $\{0, 1, 2\}$ che non appare in uv: pertanto, yu^2wv^2z non può contenere lo stesso numero di simboli `0`, `1` e `2` e, quindi, non può appartenere a L.
- u oppure v contiene almeno due tipi di simboli (supponiamo che ciò sia vero per u). Poiché $x \in L$, i simboli `0`, `1` e `2` devono apparire in u nell'ordine giusto, ovvero gli eventuali simboli `0` devono precedere gli eventuali simboli `1` i quali, a loro volta, devono precedere gli eventuali simboli `2`: pertanto, la stringa yu^2wv^2z può contenere lo stesso numero di simboli `0`, `1` e `2` ma certamente non li contiene nell'ordine giusto e, quindi, non può appartenere a L.

In entrambi i casi, abbiamo generato un assurdo e, quindi, il linguaggio L non può essere libero da contesto.

È facile verificare che il linguaggio L dell'esempio precedente è generato dalla grammatica contestuale contenente le seguenti regole di produzione: $A \to 0ABC$, $A \to 0BC$, $CB \to BC$, $1B \to 11$, $1C \to 12$, $2C \to 22$ e $0B \to 01$. Alternativamente, è altrettanto facile costruire una macchina di Turing lineare che decida L. Pertanto,

Figura 8.1: il pumping lemma per linguaggi liberi da contesto.



possiamo concludere che la classe dei linguaggi liberi da contesto è strettamente inclusa in quella dei linguaggi contestuali.

8.2 Forma normale di Chomsky

Nel capitolo precedente, al momento di descrivere la procedura per l'eliminazione delle ricorsioni sinistre immediate, abbiamo assunto che la grammatica non contenesse λ-produzioni. Mostriamo ora che tale assunzione non è restrittiva e che ogni linguaggio libero da contesto che non includa la stringa vuota ammette una grammatica G che lo genera in **forma normale di Chomsky**, ovvero tale che ogni produzione di G sia del tipo $A \to BC$ oppure del tipo $A \to a$.

Lemma 8.2

Per ogni grammatica G libera da contesto tale che $\lambda \notin L(G)$, ne esiste una equivalente le cui regole di produzione sono solo $A \to BC$ oppure del tipo $A \to a$, dove B e C sono diversi dal simbolo iniziale di G.

Dimostrazione. Dimostreremo qualcosa di leggermente diverso, ovvero che, per ogni grammatica G libera da contesto, esiste una grammatica G' equivalente a G (ovvero, tale che $L(G) = L(G')$) le cui regole di produzione sono solo del tipo $S' \to \lambda$, del tipo $A \to BC$ oppure del tipo $A \to a$, dove S' è il simbolo iniziale di G' e B e C sono diversi da S'. È chiaro che la produzione $S' \to \lambda$ può essere usata solo per generare la stringa vuota, per cui il lemma risulta essere una conseguenza di tale dimostrazione.

La dimostrazione consiste nell'inizializzare G' con le regole di G, per poi trasformarla eseguendo i seguenti quattro passi.

Nuovo simbolo iniziale Inseriamo in G' la regola $S' \to S$ (in questo modo siamo sicuri che S' non appare nella parte destra di alcuna regola).

Eliminazione delle regole con parte destra vuota Le regole la cui parte destra è vuota, ovvero del tipo $X \to \lambda$ con $X \neq S'$, sono eliminate applicandole direttamente (in tutti i modi possibili) a quelle regole che contengono almeno un'occorrenza di X nella loro parte destra. Se tale applicazione produce una regola con la parte destra vuota che sia stata già analizzata, allora essa viene scartata. Questo procedimento ha termine quando la grammatica G' non include alcuna regola con la parte destra vuota (a eccezione dell'eventuale regola $S' \to \epsilon$).

Eliminazione delle regole con parte destra unitaria Le regole la cui parte destra è costituita da un solo simbolo non terminale, ovvero del tipo $X \to Y$, sono eliminate applicando a esse eventuali regole la cui parte sinistra è uguale a Y. Se tale applicazione produce una regola con la parte destra unitaria che sia stata già analizzata, allora essa viene scartata. Questo procedimento ha termine quando la grammatica G' non include alcuna regola con la parte destra unitaria.

Riduzione della parte destra A questo punto, tutte le regole hanno la parte destra di lunghezza almeno pari a 2 (a eccezione dell'eventuale regola $S' \to \epsilon$ e delle regole la cui parte destra è costituita da un solo simbolo terminale). Quelle con la parte destra superiore a 2 sono sostituite con una sequenza di regole la cui parte destra è uguale a 2 nel modo seguente. Se $X \to \alpha_1\alpha_2\cdots\alpha_k$ è una regola con $k \geqslant 3$, sostituiamo tale regola con le regole $X \to \alpha_1 X_1$, $X_1 \to \alpha_2 X_2$, $\ldots$, $X_{k-3} \to \alpha_{k-2}X_{k-2}$ e $X_{k-2} \to \alpha_{k-1}\alpha_k$ (dove i simboli X_i sono nuovi simboli non terminali).

Normalizzazione finale Per ogni simbolo terminale a che appare nella parte destra di una regola del tipo $X \to \alpha_1\alpha_2$ creiamo un nuovo simbolo non terminale X_a, inseriamo una regola $X_a \to a$ e sostituiamo l'occorrenza di a con il simbolo X_a.

È facile verificare che la nuova grammatica è equivalente a quella originale: in effetti, quello che abbiamo fatto è stato semplicemente di anticipare l'applicazione di alcune produzioni e, successivamente, di scomporre le parti destre con più di due simboli in sottosequenze di esattamente due simboli, per poi, infine, far corrispondere a ogni simbolo terminale occorrente in una parte destra di due simboli un simbolo non terminale da usare come passaggio intermedio. ◇

La trasformazione di una grammatica libera da contesto in una in forma normale di Chomsky può essere, quindi, fatta in modo automatico: tuttavia, tale procedimento

può generare un numero abbastanza elevato di nuove regole e di nuovi simboli non terminali, come mostrato nel prossimo esempio.

Esempio 8.2: trasformazione in forma normale di Chomsky

Consideriamo la solita grammatica dell'espressioni aritmetiche ottenuta dopo avere eliminato le ambiguità e le ricorsioni sinistre. Per semplicità limitiamoci a considerare solo gli operatori di somma e moltiplicazione: pertanto, la grammatica è la seguente:

$$\mathsf{E} \to \mathsf{TQ} \quad \mathsf{Q} \to +\mathsf{TQ} \quad \mathsf{Q} \to \lambda \quad \mathsf{T} \to \mathsf{FR} \quad \mathsf{R} \to *\mathsf{FR} \quad \mathsf{R} \to \lambda \quad \mathsf{F} \to (\mathsf{E}) \quad \mathsf{F} \to \mathbf{id}$$

In base al primo passo della procedura descritta nella dimostrazione del teorema precedente, aggiungiamo alla nuova grammatica la produzione $\mathsf{E}' \to \mathsf{E}$. Il secondo passo, invece, ci porta ad aggiungere alla grammatica le regole $\mathsf{E} \to \mathsf{T}$, $\mathsf{Q} \to +\mathsf{T}$, $\mathsf{T} \to \mathsf{F}$ e $\mathsf{R} \to *\mathsf{F}$, eliminando così le due regole $\mathsf{Q} \to \lambda$ e $\mathsf{R} \to \lambda$. Passiamo ora a eliminare le regole con un solo simbolo non terminale nella parte destra, ovvero le tre regole appena create $\mathsf{E}' \to \mathsf{E}$, $\mathsf{E} \to \mathsf{T}$ e $\mathsf{T} \to \mathsf{F}$. La prima regola viene sostituita da $\mathsf{E}' \to \mathsf{TQ}$ e da $\mathsf{E}' \to \mathsf{T}$: la seconda di queste due nuove produzioni deve a sua volta essere sostituita con $\mathsf{E}' \to \mathsf{FR}$ e $\mathsf{E}' \to \mathsf{F}$, la quale a sua volta viene sostituita da $\mathsf{E}' \to (\mathsf{E})$ e $\mathsf{E}' \to \mathbf{id}$. Analogamente, la regola $\mathsf{E} \to \mathsf{T}$ viene sostituita da $\mathsf{E} \to \mathsf{FR}$ e da $\mathsf{E} \to \mathsf{F}$: la seconda di queste due nuove produzioni deve a sua volta essere sostituita con $\mathsf{E} \to (\mathsf{E})$ e $\mathsf{E} \to \mathbf{id}$. Infine, la regola $\mathsf{T} \to \mathsf{F}$ può essere sostituita dalla due regole $\mathsf{T} \to (\mathsf{E})$ e $\mathsf{T} \to \mathbf{id}$. Ci siamo così ricondotti a una grammatica con sole produzioni contenenti nella parte destra un simbolo terminale oppure almeno due simboli, ovvero la seguente grammatica (notiamo come il simbolo E sia sparito dalla grammatica):

$$\begin{array}{llll}
\mathsf{E}' \to \mathsf{TQ} & \mathsf{E}' \to \mathsf{FR} & \mathsf{E}' \to (\mathsf{E}) & \mathsf{E}' \to \mathbf{id} \\
\mathsf{Q} \to +\mathsf{TQ} & & & \\
\mathsf{T} \to \mathsf{FR} & \mathsf{T} \to (\mathsf{E}) & \mathsf{T} \to \mathbf{id} & \\
\mathsf{R} \to *\mathsf{FR} & & & \\
\mathsf{F} \to (\mathsf{E}) & \mathsf{F} \to \mathbf{id} & &
\end{array}$$

A questo punto la regola $\mathsf{E}' \to (\mathsf{E})$ viene sostituita dalle due regole $\mathsf{E}' \to (\mathsf{V}_1$ e $\mathsf{V}_1 \to \mathsf{E})$, la regola $\mathsf{Q} \to +\mathsf{TQ}$ con le due regole $\mathsf{Q} \to +W_1$ e $W_1 \to \mathsf{TQ}$, la regola $\mathsf{T} \to (\mathsf{E})$ con le due regole $\mathsf{T} \to (\mathsf{X}_1$ e $\mathsf{X}_1 \to \mathsf{E})$, la regola $\mathsf{R} \to *\mathsf{FR}$ con le due regole $\mathsf{R} \to *Y_1$ e $Y_1 \to \mathsf{FR}$ e la regola $\mathsf{F} \to (\mathsf{E})$ viene sostituita dalle due regole $\mathsf{F} \to (\mathsf{Z}_1$ e $\mathsf{Z}_1 \to \mathsf{E})$. Appllicando alle regole così ottenute l'ultimo passo della dimostrazione otteniamo la seguente grammatica in forma normale di Chomsky:

$$\begin{array}{lllllll}
\mathsf{E}' \to \mathsf{TQ} & \mathsf{E}' \to \mathsf{FR} & \mathsf{E}' \to \mathsf{V}_(\mathsf{V}_1 & \mathsf{V}_(\to (& \mathsf{V}_1 \to \mathsf{EV}_) & \mathsf{V}_) \to) & \mathsf{E}' \to \mathbf{id} \\
\mathsf{Q} \to W_+ W_1 & W_+ \to + & W_1 \to \mathsf{TQ} & & & & \\
\mathsf{T} \to \mathsf{FR} & \mathsf{T} \to \mathsf{X}_(\mathsf{X}_1 & \mathsf{X}_(\to (& \mathsf{X}_1 \to \mathsf{EX}_) & \mathsf{X}_) \to) & \mathsf{T} \to \mathbf{id} & \\
\mathsf{R} \to Y_* Y_1 & Y_* \to * & Y_1 \to \mathsf{FR} & & & & \\
\mathsf{F} \to \mathsf{Z}_(\mathsf{Z}_1 & \mathsf{Z}_(\to (& \mathsf{Z}_1 \to \mathsf{EZ}_) & \mathsf{Z}_) \to) & \mathsf{F} \to \mathbf{id} & &
\end{array}$$

Questa grammatica potrebbe ovviamente essere semplificata osservando, ad esempio, che i simboli non terminali $\mathsf{V}_(, \mathsf{X}_($ e $\mathsf{Z}_($ potrebbero essere fusi in un unico simbolo non terminale.

Osserviamo che esistono altre forme normali, di cui forse la più nota è quella di Greibach in cui ogni produzione è del tipo $A \to a\alpha$, dove a è un simbolo terminale e α è una sequenza (eventualmente vuota) di simboli terminali e non terminali.

8.3 Automi a pila non deterministici

L'Esempio 6.4 oltre a mostrare (combinato con l'Esempio 5.1) che la classe dei linguaggi di tipo 3 è strettamente contenuta nella classe dei linguaggi di tipo 2, suggerisce anche quale potrebbe essere il modello di calcolo corrispondente a quest'ultima classe. In effetti, per riconoscere il linguaggio costituito da tutte e sole le stringhe del tipo $a^n b^n$, per $n > 0$, sembra necessario avere a disposizione una memoria aggiuntiva potenzialmente infinita, che consenta all'automa di ricordare il numero di simboli a letti. In realtà, è sufficiente un ben noto tipo di memoria, ovvero una *pila*, a cui è possibile accedere da un solo estremo, detto *cima* della pila, in cui un nuovo elemento viene inserito e da cui un elemento viene estratto. In effetti, il linguaggio costituito da tutte e sole le stringhe del tipo $a^n b^n$, per $n > 0$, può essere deciso da un automa a stati finiti dotato di una pila aggiuntiva, il quale inizialmente legge i simboli a e li inserisce nella pila: quindi, una volta incontrato il primo simbolo b, l'automa inizia a rimuovere i simboli a dalla pila, uno per ogni simbolo b che incontra nella stringa di input. Quest'ultima sarà accettata se alla fine dell'input la pila risulterà essere vuota.

Per poter caratterizzare i linguaggi liberi da contesto, tuttavia, abbiamo bisogno anche della possibilità di definire transizioni non deterministiche: intuitivamente, il non determinismo consentirà all'automa a pila di provare in modo, appunto, non deterministico le diverse regole di produzione applicabili a una determinata forma sentenziale.

Definizione 8.1: automi a pila non deterministici

Un **automa a pila non deterministico** è una macchina di Turing non deterministica con due nastri semi-infiniti, con *alfabeto di input* Σ e con *alfabeto di pila* Γ (con $\square \notin \Sigma \cup \Gamma$), tale che ogni etichetta di un arco del grafo delle transizioni contiene triple del tipo (x, y, z), dove $x \in \Sigma^*$ è la sequenza di simboli letti sul primo nastro, $y \in \Gamma^*$ è la sequenza di simboli letti sul secondo nastro e $z \in \Gamma^*$ è la sequenza di simboli da scrivere sul secondo nastro al posto di y (eventualmente spostando a destra oppure a sinistra il contenuto alla destra di y). A ogni transizione, la testina del primo nastro si sposta di $|x|$ posizioni a destra e la testina del secondo nastro rimane sulla prima cella.

La computazione di un automa a pila non deterministico T ha inizio con la stringa di input x presente sul primo nastro e con il secondo nastro completamente vuoto:

un cammino di computazione è accettante se termina in uno stato finale con la testina del primo nastro posizionata sul primo simbolo □ alla destra di x. Quindi, se un cammino di computazione di T con input una stringa $x \in \Sigma^*$ termina in uno stato finale con la testina del primo nastro posizionata su un simbolo di x, allora tale cammino di computazione non è accettante. Osserviamo, inoltre, che un cammino di computazione accettante non necessariamente deve terminare con la pila vuota: non è difficile, comunque, modificare T in modo che ogni cammino di computazione accettante termini dopo aver svuotato completamente la pila.

Esempio 8.3: un automa a pila non deterministico

Consideriamo il linguaggio L costituito da tutte e sole le stringhe binarie del tipo $w_1 \cdots w_n w_n \cdots w_1$ con $w_i \in \{0,1\}$ e $n > 0$. Un automa a pila non deterministico T, che accetti tutte e sole le stringhe in L, può non deterministicamente scegliere un punto in cui spezzare la stringa x di input e successivamente, facendo uso della pila, verificare che le due parti di x così ottenute siano una l'inversa dell'altra (si veda la Figura 8.2). In particolare, l'automa T, dopo aver inserito nella pila un simbolo speciale (come, ad esempio, il simbolo #), non deterministicamente prosegue l'esecuzione inserendo nella pila il successivo simbolo di x oppure (se il simbolo letto e quello in cima alla pila coincidono) iniziando il confronto tra i simboli di x e quelli contenuti nella pila. In quest'ultimo caso, T termina il confronto ed entra nel suo unico stato finale nel momento in cui il simbolo in cima alla pila è il simbolo #: se la stringa x è stata interamente letta, allora T accetta x. Osserviamo che L è chiaramente un linguaggio libero da contesto generato dalla grammatica avente le seguenti regole di produzione: $A \to 0A0$, $A \to 1A1$, $A \to 00$ e $A \to 11$.

L'esempio precedente può essere generalizzato all'intera classe dei linguaggi generati da grammatiche libere da contesto, come mostrato dal seguente risultato.

Teorema 8.1

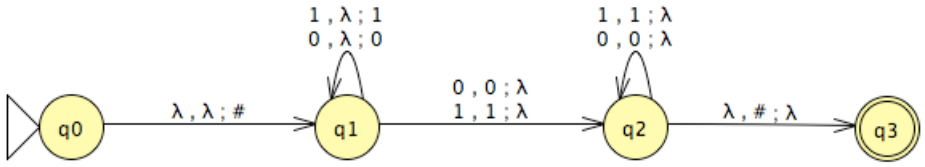
Un linguaggio L è di tipo 2 se e solo se esiste un automa a pila non deterministico T tale che $L = L(T)$.

Dimostrazione. Dato un linguaggio L libero da contesto, ovvero generato da una grammatica di tipo 2, definiamo un automa a pila non deterministico T che opera nel modo seguente (si veda l'automa mostrato nella Figura 8.3 che corrisponde alla grammatica di tipo 2 introdotta nell'Esempio 8.3).

1. Mette in pila un simbolo speciale (come, ad esempio, il simbolo #) e il simbolo iniziale della grammatica.

2. Se la cima della pila contiene un simbolo non terminale A, non deterministicamente seleziona una delle regole di produzione la cui parte sinistra è uguale

Figura 8.2: l'automa a pila non deterministico dell'Esempio 8.3.



a A e inserisce in pila (in ordine inverso) i simboli della parte destra della produzione.

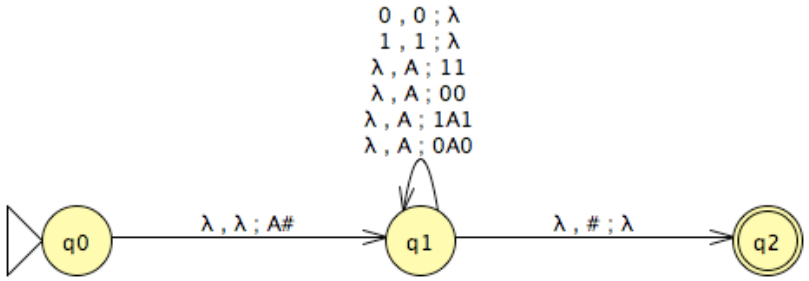
3. Se la cima della pila contiene un simbolo terminale a, lo confronta con il simbolo letto sul nastro di input: se sono uguali, toglie a dalla pila.

4. Se la cima della pila contiene il simbolo # termina in uno stato finale e, quindi, accetta se e solo se l'input è stato interamente letto.

Chiaramente T accetta tutte e sole le stringhe che possono essere generate dalla grammatica, ovvero tutte e sole le stringhe appartenenti a L. Osserviamo come la definizione dell'automa a pila non deterministico è molto simile a quella di un parser predittivo introdotto nel capitolo precedente: in quel caso, il non determinismo non era necessario in quanto avevamo assuntio che la grammatica fosse di tipo LL(1).

Viceversa, supponiamo che L sia un linguaggio accettato da un automa a pila non deterministico T. Assumiamo, senza perdita di generalità, che T abbia un unico stato finale q^*, che T svuoti la pila prima di terminare e che, a ogni transizione, T esegua un inserimento nella pila di un simbolo oppure un'estrazione dalla pila di un simbolo (in altre parole, a ogni transizione la pila aumenta o diminuisce la sua dimensione di un'unità e le triple contenute nell'etichetta di un arco sono del tipo (x, λ, σ) oppure del tipo (x, σ, λ), con $x \in \Sigma^*$ e $\sigma \in \Gamma$): per esemplificare la costruzione della grammatica, nel seguito faremo riferimento all'automa descritto nell'Esempio 8.3, il cui grafo delle transizioni è mostrato nella Figura 8.2 e che soddisfa le tre precedenti condizioni con $q^* = q2$. Per ogni coppia di stati p e q tali che esiste una transizione a partire da p che esegue un inserimento nella pila ed esiste una transizione che termina in q e che esegue un'estrazione dalla pila, la grammatica G include un simbolo non terminale A_{pq}: costruiremo G in modo che, a partire da A_{pq}, sia possibile generare solo le stringhe che portano T dallo stato p con la pila vuota allo stato q con la pila vuota (osserviamo che tali stringhe possono anche far passare T dallo stato p allo stato q, indipendentemente dal contenuto della pila nello stato p e

Figura 8.3: un automa a pila non deterministico corrispondente a una grammatica di tipo 2.



lasciando tale contenuto invariato al momento in cui lo stato q viene raggiunto). Il simbolo iniziale di A sarà dunque $A_{q_0q^*}$: osserviamo che, poichè la computazione ha inizio con la pila vuota, deve esistere una transizione a partire da q_0 che esegue un inserimento nella pila e che, poichè la computazione ha termine con la pila vuota, deve esistere una transizione che termina in q^* e che esegue un'estrazione dalla pila. Per consentire la produzione di sequenze contenenti solo simboli terminali, per ogni stato p di T, la grammatica G include anche il simbolo non terminale A_{pp}. Nel nostro esempio, quindi, abbiamo che la grammatica include i sette simboli non terminali A_{q0q2}, A_{q0q3}, A_{q1q2}, A_{q1q3}, A_{q0q0}, A_{q1q1} e A_{q2q2}. Sia (x,λ,a) una tripla contenuta nell'etichetta di un arco e uscente da uno stato p e sia (y,b,λ) una tripla contenuta nell'etichetta di un arco f entrante in uno stato q: distinguiamo allora i seguenti due casi.

- $a = b$. In tal caso, aggiungiamo a G la regola di produzione $A_{pq} \to xA_{rs}y$, dove r è lo stato in cui l'arco e entra, a partire da p, e s è lo stato da cui l'arco f arriva, per entrare in q. Nel nostro esempio, abbiamo che la grammatica include le seguenti produzioni.

$$A_{q0q3} \to A_{q1q2}$$
$$A_{q1q2} \to 0A_{q1q1}0 \quad A_{q1q2} \to 1A_{q1q1}1$$
$$A_{q1q2} \to 0A_{q1q2}0 \quad A_{q1q2} \to 1A_{q1q2}1$$

- $a \neq b$. In tal caso, per ogni stato r di T per il quale esiste una transizione a partire da r che esegue un inserimento nella pila ed esiste una transizione che termina in r e che esegue un'estrazione dalla pila, aggiungiamo a G la regola di produzione $A_{pq} \to A_{pr}A_{rq}$. Nel nostro esempio, non esiste alcuno

stato r con le suddette proprietà e, quindi, nessuna produzione va aggiunta alla grammatica.

Infine, aggiungiamo a G la regola di produzione $A_{pp} \to \lambda$, per ogni stato p. Nel nostro esempio, dunque, la definizione della grammatica si conclude aggiungendo le seguenti produzioni.

$$A_{q0q0} \to \lambda \quad A_{q1q1} \to \lambda \quad A_{q2q2} \to \lambda$$

Nel caso del nostro esempio, è immediato verificare che il linguaggio generato da G coincide con quello accettato dall'automa a pila non deterministico, in quanto a partire dal simbolo iniziale A_{q0q3} è possibile solo produrre il simbolo A_{q1q2} e, a partire da quest'ultimo, per ogni simbolo terminale prodotto a sinistra, viene prodotto lo stesso simbolo terminale a destra. In generale, se una stringa x è accettata da T, allora l'automa deve iniziare l'esecuzione nello stato q_0 con la pila vuota e un suo cammino di computazione deve terminare nello stato q^* con la pila vuota. D'altra parte, se un cammino di computazione passa da uno stato p con la pila contenente la stringa y a uno stato q con la pila contenente la stringa y, allora il contenuto della pila è uguale a y solo all'inizio e alla fine del passaggio da p a q oppure nel passaggio da p a q, il contenuto della pila deve essere stato uguale a y in corrispondenza di uno stato intermedio r: questi due casi corrispondono esattamente ai due casi precedentemente distinti. Lasciamo al lettore il compito di completare, in base alle suddette osservazioni, la dimostrazione formale che la grammatica G genera tutte e sole le stringhe accettate da T. ◇

Notiamo che è possibile mostrare come, nella dimostrazione del teorema precedente, il non determinismo sia necessario, ovvero che gli automi a pila deterministici non sono in grado di caratterizzare l'intero insieme dei linguaggi di tipo 2. Sebbene la dimostrazione di quest'affermazione sia piuttosto complicata, possiamo giustificarla in modo intuitivo considerando nuovamente il linguaggio dell'Esempio 8.3, ovvero il linguaggio L costituito da tutte e sole le stringhe binarie del tipo $w_1 \cdots w_n w_n \cdots w_1$ con $w_i \in \{0,1\}$ e $n > 0$, per il quale abbiamo già dimostrato l'esistenza di una automa a pila non deterministico. Per convincerci che non esiste un automa a pila deterministico in grado di accettare tutte e sole le stringhe di L, osserviamo che, leggendo la stringa 0^n110^n, un tale automa dovrebbe terminare con la pila vuota in quanto tale stringa appartiene a L. Se ciò che segue è la sequenza 0^n110^n, allora l'automa deve terminare in uno stato di accettazione, mentre se ciò che segue è 0^m110^m con $m \neq n$, allora l'automa deve terminare in uno stato di rifiuto. Tuttavia, poiché la pila è vuota, l'automa non è in grado di ricordare il valore di n in modo da confrontarlo con il valore di m: concludiamo che un tale automa non può esistere.

Osserviamo che, chiaramente, ogni linguaggio regolare può essere accettato da un automa a pila deterministico (il quale non fa uso della pila): la discussione pre-

cedente, combinata con l'Esempio 8.3, ci consente quindi di affermare che gli automi a pila deterministici accettano un insieme di linguaggi compreso tra quello dei linguaggi regolari e quello dei linguaggi liberi da contesto.

In conclusione, quanto esposto in questa parte delle dispense ci consente di riassumere nella seguente tabella, la classificazione dei linguaggi in base alla loro tipologia, al tipo di grammatiche che li generano e in base al modello di calcolo corrispondente (con riferimento poi alle proprietà di decidibilità introdotte nella prima parte delle dispense, possiamo concludere dicendo che la classe dei linguaggi di tipo 0 coincide con quella dei linguaggi semi-decidibili, mentre le altre tre classi di linguaggi sono interamente contenute nella classe dei linguaggi decidibili).

Tipo di linguaggio	**Tipo di produzioni**	**Modello di calcolo**
Tipo 0	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$ e $\beta \in (V \cup T)^*$	Macchina di Turing
Contestuale	$\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$, $\beta \in (V \cup T)(V \cup T)^*$ e $\lvert\beta\rvert \geqslant \lvert\alpha\rvert$	Macchina di Turing lineare
Libero da contesto	$A \to \beta$ con $A \in V$ e $\beta \in (V \cup T)(V \cup T)^*$	Automa a pila non deterministico
Regolare	$A \to aB$ e $A \to a$ con $A, B \in V$ e $a \in T$	Automa a stati finiti

Esercizi

Esercizio 8.1. Utilizzando il Lemma 8.1 dimostrare che il linguaggio costituito da tutte e sole le stringhe del tipo $w2x$, con $w, x \in \{0,1\}^*$ e w contenuta in x, non è libero da contesto.

Esercizio 8.2. Facendo uso del pumping lemma per i linguaggi liberi da contesto, dimostrare che il seguente linguaggio non è libero da contesto.

$$L = \{0^i 1^j 2^k : i \leqslant j \leqslant k\}$$

Esercizio 8.3. Dire se il seguente linguaggio è libero da contesto.

$$L = \{0^i 1^j 2^i 3^j : i \geqslant 1 \wedge j \geqslant 1\}$$

Giustificare la risposta.

Esercizio 8.4. Fornire un esempio di linguaggio non libero da contesto che soddisfi le condizioni del Lemma 8.1. Dimostrare che l'esempio è corretto.

Esercizio 8.5. Trasformare in forma normale di Chomsky la seguente grammatica: $S \to 0A0$, $S \to 1B1$, $S \to BB$, $A \to C$, $B \to S$, $B \to A$, $C \to S$, $C \to \lambda$.

Esercizio 8.6. Trasformare in forma normale di Chomsky la seguente grammatica: $S \to AAA$, $S \to B$, $A \to aA$, $A \to B$, $B \to \lambda$.

Esercizio 8.7. Trasformare in forma normale di Chomsky la seguente grammatica: $S \to aAa$, $S \to bBb$, $S \to \lambda$, $A \to C$, $A \to a$, $B \to C$, $B \to b$, $C \to CDE$, $C \to \lambda$, $D \to A$, $D \to B$, $D \to ab$.

Esercizio 8.8. Definire un automa a pila non deterministico per la seguente grammatica: $S \to 0S1$, $S \to A$, $A \to 1A0$, $A \to S$, $A \to \lambda$.

Esercizio 8.9. Definire un automa a pila non deterministico per la seguente grammatica: $S \to aAA$, $A \to aS$, $A \to bS$, $A \to a$.

Esercizio 8.10. Definire una grammatica libera da contesto che generi tutte e sole le stringhe accettate dall'automa a pila non deterministico che, per prima cosa, inserisce in pila il simbolo Z e passa nello stato p e successivamente esegue le seguenti istruzioni.

1. Una transizione dallo stato p allo stato p con etichetta $(0, Z, AZ)$.
2. Una transizione dallo stato p allo stato p con etichetta $(0, A, AA)$.
3. Una transizione dallo stato p allo stato q con etichetta (λ, Z, Z).
4. Una transizione dallo stato p allo stato q con etichetta (λ, A, A).
5. Una transizione dallo stato q allo stato q con etichetta $(1, A, \lambda)$.
6. Una transizione dallo stato q allo stato r con etichetta (λ, Z, λ).

Lo stato finale è lo stato r.

Parte III

Teoria della complessità

Capitolo 9

La classe P

SOMMARIO

Una volta stabilito che esistono problemi che non possono essere risolti in modo automatico, ovvero linguaggi non decidibili, ci chiediamo ora quanto costi invece risolvere quelli decidibili. In particolare, in questa parte delle dispense ci concentreremo sul tempo necessario affinché una macchina di Turing possa decidere se un determinato input appartiene o meno a uno specifico linguaggio. In base a tale misura di complessità, definiremo la classe P*, ovvero la classe dei problemi risolvibili efficientemente, e mostreremo alcuni esempi di linguaggi in tale classe, facendo anche uso del concetto di riducibilità polinomiale.*

9.1 La classe P

NEL PRIMO capitolo abbiamo già introdotto, più o meno esplicitamente, il concetto di **complessità temporale** di una macchina di Turing T con un singolo nastro (che d'ora in avanti chiameremo anche *algoritmo*): tale complessità è definita come la funzione $t_T : \mathbb{N} \to \mathbb{N}$ che, per ogni $n \geqslant 1$, indica il massimo numero di passi eseguiti da T con input una stringa di lunghezza n.

Notiamo come, in base a tale definizione, la complessità temporale di una macchina di Turing viene valutata esaminando il comportamento della macchina nel caso peggiore, ovvero nel caso relativo alla stringa di input che richiede il maggior numero di passi (questa è una prassi comune nella valutazione delle prestazioni degli algoritmi): così facendo, siamo sicuri che la macchina T, con input una *qualunque* stringa di lunghezza n, non eseguirà *mai* più di $t_T(n)$ passi (anche se per alcune stringhe di input ne potrebbe eseguire molti di meno).

Ricordiamo, inoltre, che, per facilitare il calcolo della complessità temporale di una macchina di Turing, di quest'ultima abbiamo fatto e sempre faremo un'**analisi asintotica**, utilizzando la notazione O, che consente di ignorare, durante il calcolo

della complessità temporale di una macchina di Turing, le costanti e tutti i termini di ordine inferiore (anche se ciò, nella pratica, può non essere del tutto irrilevante).

Una volta stabilito il concetto di complessità temporale di un algoritmo, osserviamo però che tale concetto risulta, sfortunatamente, dipendere dal modello di calcolo utilizzato. Se consideriamo, ad esempio, il linguaggio $L = \{0^n 1^n : n \geqslant 0\}$ (che non è regolare), abbiamo visto nel primo capitolo che tale linguaggio può essere deciso da una macchina di Turing con due nastri in tempo $O(n)$. Anche senza dimostrarlo, possiamo convincerci intuitivamente che, al contrario, non esiste una macchina di Turing con un solo nastro che decide L in tempo $O(n)$, in quanto altrimenti L sarebbe un linguaggio regolare (notiamo che nel primo capitolo abbiamo definito una macchina di Turing con un solo nastro che decide L in tempo $O(n^2)$ e che è solo leggermente più difficile definirne una che lo decide in tempo $O(n \log n)$).

La complessità temporale di un algoritmo può dipendere anche dalla codifica utilizzata. Ad esempio, se decidiamo di codificare i numeri interi utilizzando la codifica unaria, abbiamo che la lunghezza della rappresentazione di un numero intero aumenta in modo esponenziale rispetto a quella della rappresentazione basata sulla codifica binaria (in generale, sulla codifica in una qualunque base $b > 1$): infatti, codificare un numero intero n in base 2 richiede $\lceil \log n \rceil$ cifre binarie, mentre codificarlo in base unaria richiede $n + 1$ cifre unarie. Essendo l'input più lungo, la macchina ha più tempo a disposizione: avere una complessità temporale $O(n)$ rispetto alla codifica unaria, può voler dire che la complessità temporale rispetto alla codifica binaria sia $O(2^n)$.

Affinché la teoria della complessità computazionale (così come quella della calcolabilità) non sia dipendente dal modello di calcolo utilizzato, dobbiamo in qualche modo raggruppare in una stessa classe tutti quei linguaggi che hanno una complessità temporale non troppo diversa. La definizione di che cosa intendiamo per "non troppo diversa" è giustificata dalle seguenti due considerazioni.

In primo luogo, abbiamo già visto, nel secondo capitolo, che le diverse varianti di macchine di Turing deterministiche che sono state introdotte possono simularsi vicendevolmente con un sovraccarico computazionale polinomiale. È anche facile verificare che tutti i modelli di calcolo definiti nel quarto capitolo e che sono stati proposti in alternativa alle macchine di Turing, possono anch'essi simularsi vicendevolmente con un sovraccarico computazionale polinomiale. L'unico modello di calcolo che sfugge a questa regola è la macchina di Turing non deterministica, introdotta nel capitolo precedente: in effetti, in questo caso, non è difficile verificare che la simulazione di una macchina di Turing non deterministica mediante una deterministica richiede un numero di passi esponenziale rispetto a quello dei passi eseguiti dal più corto cammino di computazione accettante. Tuttavia, sembra difficile affermare che la macchina di Turing non deterministica sia veramente un modello di calcolo

"ragionevole".

In secondo luogo, anche se ogni definizione formale di una codifica "ragionevole" non sarebbe soddisfacente, alcune regole standard possono essere stabilite. Per esempio, possiamo fare uso delle seguenti assunzioni. I numeri sono rappresentati in una base $b > 1$, gli insiemi sono rappresentati come sequenze di elementi separati da un simbolo speciale e, quando possibile, sono specificate le regole che generano un insieme, piuttosto che l'insieme stesso. L'uso di tali codifiche standard ci permetterà di assumere che tutte le rappresentazioni ragionevoli di un problema siano tra di loro scambiabili, ovvero, data una di esse, sia possibile passare a una qualunque altra rappresentazione in un tempo polinomiale rispetto alla rappresentazione originale. Tali considerazioni ci portano, abbastanza naturalmente, a fornire la seguente definizione.

Definizione 9.1: la classe P

La classe P è l'insieme dei linguaggi L per i quali esiste una macchina di Turing T con un solo nastro che decide L e per cui $t_T(n) \in O(n^k)$ per qualche $k \geqslant 1$.

La definizione della classe P risulta, quindi, essere *robusta* rispetto sia a modifiche del modello di calcolo utilizzato che rispetto a modifiche della codifica utilizzata, purché questi siano entrambi ragionevoli. Ma l'importanza di tale classe risiede, più probabilmente, nel fatto che essa viene comunemente identificata con l'insieme dei linguaggi *trattabili*, ovvero decidibili in tempi ragionevoli. Per convincerci di tale affermazione, supponiamo di avere cinque algoritmi $A_1, \ldots, A_5$ che decidono lo stesso linguaggio ma con diverse complessità temporali, come mostrato nella seguente tabella (in cui assumiamo che 10^{-9} secondi siano richiesti per eseguire un singolo passo di ciascun algoritmo).

Dimensione istanza n	Algoritmo/complessità				
	A_1/n^2	A_2/n^3	A_3/n^5	$A_4/2^n$	$A_5/3^n$
10	0.1 μs	1 μs	0.01 ms	1 μs	59 μs
30	0.9 μs	27 μs	24.3 ms	1 s	2.4 giorni
40	1.6 μs	64 μs	0.1 s	18 minuti	386 anni
50	2.5 μs	0.125 ms	0.31 s	13 giorni	2.3×10^5 secoli

È evidente dalla tabella che i primi tre algoritmi hanno una complessità decisamente ragionevole, che il quarto algoritmo è poco utilizzabile per istanze di lunghezza superiore a 50 e che il quinto algoritmo risulta del tutto inutilizzabile per istanze di lunghezza superiore a 40. Scegliendo una complessità temporale più favorevole di quella di A_4 e di A_5 ma non polinomiale (ad esempio, una complessità sub-esponenziale del tipo $O(k^{\log^h(n)})$ con k e h costanti), le differenze sarebbero meno significative per

piccoli valori di n, ma riapparirebbero comunque al crescere di n. Possiamo, quindi, concludere che se il nostro obiettivo è quello di usare il calcolatore per risolvere problemi la cui descrizione includa un numero relativamente alto di elementi, allora è necessario che l'algoritmo di risoluzione abbia una complessità temporale polinomiale: per questo motivo, la classe P è generalmente vista come l'insieme dei problemi risolvibili in modo efficiente.

9.2 Esempi di linguaggi in P

Data l'equivalenza tra le macchine di Turing e i linguaggi di programmazione di alto livello, che abbiamo stabilito nel quarto capitolo, nel resto di questo capitolo ci limiteremo a descrivere gli algoritmi facendo riferimento a questi ultimi: in realtà, quasi sempre saremo ancora più "pigri" e faremo uso del linguaggio naturale, lasciando al lettore interessato il compito di tradurre tali descrizioni degli algoritmi in un programma scritto in un qualunque linguaggio di programmazione.

9.2.1 Numeri relativamente primi

Consideriamo il problema di decidere se due numeri interi a e b sono relativamente primi, ovvero se il loro massimo comun divisore (indicato con $\gcd(a,b)$) è uguale a 1. Un algoritmo con complessità temporale polinomiale per la risoluzione di tale problema, noto sotto il nome di **algoritmo di Euclide**, si basa sulla seguente relazione (la cui dimostrazione è lasciata come esercizio).

$$\gcd(a,b) = \begin{cases} a & \text{se } b = 0, \\ \gcd(b, a \bmod b) & \text{altrimenti} \end{cases}$$

L'algoritmo è dunque il seguente.

```
gcd(a, b):
  if b = 0 then
    return a
  else
    return gcd(b, a mod b);
```

Per valutare la complessità temporale di tale algoritmo, mostriamo che sono necessarie $O(\log b)$ chiamate ricorsive. Siano (a_{k-1}, b_{k-1}), (a_k, b_k) e (a_{k+1}, b_{k+1}) tre coppie successive di valori su cui viene invocata la funzione `gcd`: dimostriamo che $b_{k-1} \geqslant b_k + b_{k+1}$. In effetti, abbiamo che $a_k = q b_k + b_{k+1}$ per cui $a_k \geqslant b_k + b_{k+1}$.

Poiché $b_{k-1} = a_k$, segue che $b_{k-1} \geqslant b_k + b_{k+1}$. Da questa diseguaglianza deriva che $b_{k-1} \geqslant 2b_{k+1}$ e, quindi, che $b = b_0 \geqslant 2^{k/2} b_k$, per ogni $k \geqslant 2$. Pertanto, il numero di invocazioni ricorsive è $O(\log b)$ e la complessità temporale dell'algoritmo di Euclide è logaritmica rispetto al *valore* dei due numeri interi a e b, ovvero è *lineare* rispetto alla loro codifica. In conclusione, il problema di decidere se due numeri interi sono relativamente primi appartiene alla classe P.

9.2.2 Cammino minimo in un grafo

Consideriamo il problema di decidere se, dato un grafo G, due nodi u e v di G e un intero k, esiste un cammino da u a v di lunghezza al più k. Un algoritmo basato sul paradigma della programmazione dinamica per calcolare la lunghezza del cammino minimo tra una qualunque coppia di nodi (e, quindi, anche tra u e v) opera nel modo seguente. Sia n il numero di nodi di G (indicati con i numeri $1, \ldots, n$): definiamo una matrice A a tre dimensioni tale che, per ogni i, j e h con $1 \leqslant i, j \leqslant n$ e $0 \leqslant h \leqslant n$, $A[h][i][j]$ è uguale alla lunghezza del cammino minimo dal nodo i al nodo j che passi solo per nodi di indice minore oppure uguale a h. Chiaramente, per ogni i e j con $1 \leqslant i, j \leqslant n$, $A[0][i][j] = 1$ se e solo se esiste l'arco che connette i a j. Inoltre, è facile verificare che vale la seguente relazione.

$$A[h+1][i][j] = \min\{A[h][i][h+1] + A[h][h+1][j], A[h][i][j]\}$$

Intuitivamente, tale relazione afferma che un cammino minimo da i a j, che non passi per nodi di indici maggiori di $h+1$, passa per il nodo $h+1$ oppure coincide con il cammino minimo da i a j che non passa per nodi di indici maggiori di h. Pertanto, la matrice tridimensionale A può essere facilmente costruita partendo dal "piano" corrispondente a $h = 0$ e calcolando il "piano" corrispondente a $h+1$ facendo uso solamente del "piano" precedente. La complessità temporale di tale costruzione è dunque $O(n^3)$. Poichè decidere se esiste un cammino da u a v di lunghezza al più k equivale a verificare se $A[n][u][v] \leqslant k$, abbiamo che tale problema appartiene anch'esso alla classe P.

9.2.3 Soddisfacibilità di clausole con due letterali

Sia $X = \{x_0, x_1, \ldots, x_{n-1}\}$ un insieme di n variabili booleane. Una **formula booleana in forma normale congiuntiva** su X è un insieme $C = \{c_0, c_1, \ldots, c_{m-1}\}$ di m **clausole**, dove ciascuna clausola c_i, per $0 \leqslant i < m$, è a sua volta un insieme di **letterali**, ovvero un insieme di variabili in X e/o di loro negazioni (indicate con $\neg x_i$). Un'**assegnazione di valori** per X è una funzione $\tau : X \to \{\texttt{true}, \texttt{false}\}$ che assegna a ogni variabile un valore di verità. Un letterale l è soddisfatto da τ se $l = x_j$

e $\tau(x_j) = \texttt{true}$ oppure se $l = \neg x_j$ e $\tau(x_j) = \texttt{false}$, per qualche $0 \leqslant j < n$. Una clausola è soddisfatta da τ se *almeno* un suo letterale lo è e una formula è soddisfatta da τ se *tutte* le sue clausole lo sono. Il **problema della soddisfacibilità** (indicato con SAT) consiste nel decidere se una formula booleana in forma normale congiuntiva è soddisfacibile. In particolare, il problema 2-SAT è la restrizione di SAT al caso in cui le clausole contengano esattamente due letterali. Mostriamo ora che 2-SAT appartiene alla classe P descrivendo un algoritmo la cui complessità temporale è polinomiale: l'idea alla base dell'algoritmo consiste nel tentare di assegnare un valore booleano a una variabile arbitraria e nel dedurre tutte le conseguenze derivanti da quest'assegnazione per le altre variabili. Inizialmente, tutte le clausole sono dichiarate come non soddisfatte. Selezioniamo in modo arbitrario una variabile iniziale x e assegniamo il valore `true` a x: estendiamo quindi l'assegnazione a quante piú variabili possibile applicando ripetutamente il seguente passo.

> Sia c una clausola non soddisfatta. Se uno dei due letterali di c è soddisfatto, allora dichiariamo c soddisfatta. Altrimenti, se uno dei due letterali di c non è soddisfatto, allora assegniamo alla variabile corrispondente all'altro letterale il valore di verità per cui esso risulti soddisfatto e dichiariamo c soddisfatta.

I seguenti tre casi (esclusivi) possono verificarsi.

- Durante l'esecuzione di un passo, ha luogo un conflitto nel momento in cui l'algoritmo cerca di assegnare il valore `true` a una variabile a cui è già stato assegnato il valore `false` (o viceversa). Questo significa che il tentativo iniziale di assegnazione di un valore alla variabile x di partenza era sbagliato. In tal caso, tutti i passi eseguiti a partire dall'assegnazione del valore `true` a x sono annullati e, questa volta, a x viene assegnato il valore `false`: la procedura di estensione dell'assegnazione viene quindi riavviata. Se si verifica un secondo conflitto, allora l'algoritmo termina dichiarando la formula non soddisfacibile.

- Tutte le variabili hanno ricevuto un valore booleano: in tal caso, l'algoritmo termina dichiarando la formula soddisfacibile.

- Alcune variabili non hanno ancora assegnato un valore booleano. Ciò può accadere solo quando, per ogni clausola non ancora soddisfatta, alle variabili che appaiono nella clausola non è stato assegnato alcun valore. In tal caso, possiamo ignorare le clausole già soddisfatte e proseguire l'esecuzione dell'algoritmo sulla formula ridotta contenente le sole clausole non ancora soddisfatte: chiaramente, la formula ridotta è soddisfacibile se e solo se la formula di partenza

è soddisfacibile. Un tentativo di assegnazione è allora fatto per una variabile arbitraria a cui non sia ancora stato assegnato un valore di verità e la procedura di estensione dell'assegnazione viene riavviata.

È facile verificare che l'algoritmo appena descritto decide il problema 2-SAT e ha complessità temporale $O(nm)$, per cui 2-SAT appartiene alla classe P. Osserviamo, però, che tale algoritmo non può essere esteso al caso in cui ogni clausola contenga esattamente tre letterali (indicato con 3-SAT), in quanto il fatto che un letterale sia falso all'interno di una clausola non ha una diretta conseguenza sugli altri due letterali. Osserviamo, infine, che se invece di considerare formule in forma normale congiuntiva, considerassimo formule in forma normale disgiuntiva, allora il problema della soddisfacibilità sarebbe banalmente risolvibile in tempo polinomiale, indipendentemente dal numero di letterali inclusi in una clausola (si veda l'Esercizio 9.2).

9.3 Riducibilità polinomiale

Analogamente a quanto fatto nel caso della teoria della calcolabilità, possiamo mostrare che un linguaggio appartiene alla classe P mostrando che tale linguaggio non è più difficile di un altro linguaggio, la cui appartenenza alla classe P è già stata stabilita. Per tale motivo, introduciamo ora il concetto di riducibilità polinomiale.

Definizione 9.2: linguaggi polinomialmente riducibili

Un linguaggio L_1 è **polinomialmente riducibile** a un linguaggio L_2 se esiste una riduzione f da L_1 a L_2 che sia calcolabile da un algoritmo con complessità temporale polinomiale.

In modo analogo a quanto fatto nel caso del Lemma 3.4, possiamo dimostrare che se L_1 è polinomialmente riducibile a L_2 e L_2 appartiene alla classe P, allora L_1 appartiene alla classe P (si veda l'Esercizio 9.6). Pertanto la riducibilità polinomiale può essere vista come uno strumento alternativo (e, in realtà, molto utilizzato nel campo della progettazione e dell'analisi di algoritmi) per dimostrare la trattabilità di un problema. Consideriamo, ad esempio, il problema 2-COLORABILITY, che consiste nel decidere se, dato un grafo $G = (N, E)$, esiste una funzione $f : N \to \{1, 2\}$ tale che $f(u) \neq f(v)$ se $(u, v) \in E$ o se $(v, u) \in E$. Dimostriamo che 2-COLORABILITY appartiene alla classe P mostrando che 2-COLORABILITY è polinomialmente riducibile a 2-SAT. La riduzione consiste semplicemente nell'associare una variabile booleana u_i a ogni nodo i del grafo e nell'associare le due clausole $\{u_i, u_j\}$ e $\{\neg u_i, \neg u_j\}$ a ogni arco (i, j) di G. Chiaramente, se queste due clausole sono soddisfatte, allora le

due variabili u_i e u_j devono avere assegnati due diversi valori e, quindi, i due nodi corrispondenti devono essere colorati con due colori diversi. Quindi, esiste una colorazione di G che usa due colori se e solo se la formula ottenuta dalla riduzione è soddisfacibile: pertanto, 2-COLORABILITY è polinomialmente riducibile a 2-SAT.

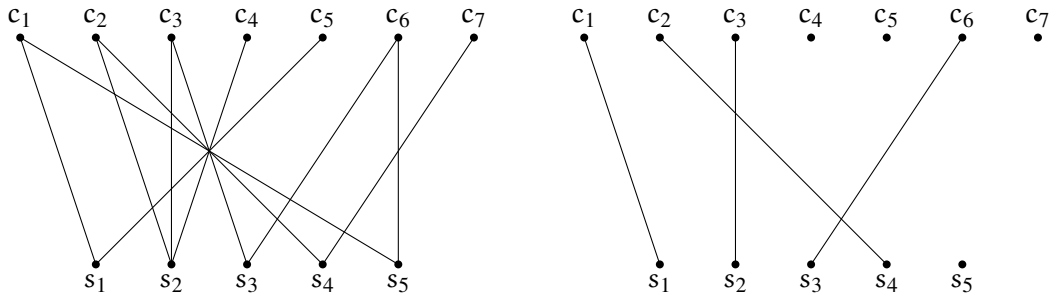
Come vedremo più avanti nel prossimo capitolo, la riducibilità polinomiale viene principalmente usata, nel campo della teoria della complessità computazionale, per dimostrare che un problema è presumibilmente non trattabile, allo stesso modo con cui abbiamo usato la riducibilità nel quarto capitolo per dimostrare che un dato linguaggio non è decidibile. Nel resto di questo paragrafo, mostreremo però un'altra applicazione "positiva" della riducibilità polinomiale.

9.3.1 Il problema del massimo accoppiamento bipartito

Supponiamo di avere un insieme di n compiti che devono essere svolti da m studenti: ogni compito può essere svolto solo da un sottoinsieme degli studenti e ogni studente può svolgere un solo compito. Il problema che ci proponiamo di risolvere consiste nel trovare un'assegnazione dei compiti agli studenti che massimizzi il numero totale dei compiti svolti. Questa situazione può essere modellata mediante un **grafo bipartito**, ovvero un grafo $G = (C \cup S, E)$ in cui C e S sono due insiemi di vertici disgiunti ed E è l'insieme degli archi che possono connettere solo vertici in C con vertici in S, ovvero $E \subseteq C \times S$: ad esempio, nella parte destra della Figura 9.1 è mostrato un grafo bipartito con 7 nodi nell'insieme C, 5 nodi nell'insieme S e 11 archi. Il **problema del massimo accoppiamento bipartito** consiste nel decidere se, dato un grafo bipartito G e dato un intero k, esiste un sottoinsieme M di E (detto *accoppiamento*) tale che $|M| \geqslant k$ e, per ogni coppia di archi (u, v) e (x, y) in M, si abbia $u \neq x$ e $v \neq y$ (in altre parole, ogni vertice di G è incidente ad al più un arco in M). Ad esempio, nella parte destra della Figura 9.1 è mostrato un accoppiamento M con 4 archi, per cui la risposta al problema del massimo accoppiamento bipartito con input il grafo mostrato nella parte sinistra e $k = 4$ è affermativa: la risposta sarebbe ancora affermativa se k fosse uguale a 5? Ovvero, quello che ci domandiamo è quale sia la cardinalità del più grande accoppiamento possibile: osserviamo che, chiaramente, tale cardinalità non potrà mai essere superiore al minimo tra $|C|$ e $|S|$ (che, nel nostro esempio, è 5).

Per rispondere a tale domanda, possiamo pensare a una strategia che calcoli il massimo accoppiamento mediante miglioramenti successivi. Il punto è, dato un accoppiamento M che non sia massimo, come sia possibile migliorarlo. Consideriamo, ad esempio, l'accoppiamento M mostrato nella parte destra della Figura 9.1: a prima vista, potrebbe sembrare che non ci sia nulla da fare per migliorare quest'accoppiamento, in quanto il nodo s_5 potrebbe solo accoppiarsi con i nodi c_1 e c_6, i quali tuttavia sono già accoppiati con i nodi s_1 e s_3 rispettivamente. Pertanto, se vogliamo migliorare tale accoppiamento dobbiamo in qualche modo riarrangiare gli accoppia-

Figura 9.1: un esempio di grafo bipartito e un possibile accoppiamento.



menti già esistenti, in modo da aumentare il numero degli accoppiamenti totali. A tale scopo, consideriamo un nodo dell'insieme C non ancora accoppiato, come, ad esempio, il nodo c_4: per poter accoppiare tale nodo, dobbiamo farlo necessariamente con il nodo s_2, il quale però è già accoppiato con il nodo c_3. Proviamo allora a separare s_2 da c_3 e ad accoppiare c_3 con qualche altro nodo, che necessariamente deve essere s_3. Anche in questo caso, s_3 è già accoppiato con c_6: per accoppiare c_3 e s_3, separiamo s_3 da c_6 e proviamo ad accoppiare c_6 a qualche altro nodo: finalmente, c_6 può accoppiarsi con s_5 che è al momento libero. Abbiamo così trovato un accoppiamento migliore (si veda la parte sinistra della Figura 9.2), in cui tutti i nodi di S sono incidenti a un arco dell'accoppiamento: per questo motivo, tale accoppiamento è anche quello massimo.

Nell'esempio precedente, il nuovo accoppiamento è stato trovato seguendo un cammino che partiva da un nodo dell'insieme C non ancora accoppiato, che terminava in un nodo dell'insieme S non ancora accoppiato e che alternava il percorrere un arco non incluso in M con il percorrere uno incluso in M: tale cammino è mostrato nella parte destra della Figura 9.2, in cui gli archi percorsi dal cammino non inclusi in M sono disegnati tratteggiati. Un cammino che alterna archi non in M con archi in M è detto essere un **cammino alternante** rispetto a M: un cammino alternante in cui il primo e l'ultimo nodo non sono accoppiati è detto essere un **cammino aumentante**. Il motivo di questa definizione è che un cammino aumentante implica sempre che l'accoppiamento M può essere migliorato togliendo da M tutti gli archi del cammino che ne facevano parte e aggiungendo a M tutti gli archi del cammino che non ne facevano parte: poiché il cammino inizia da un nodo dell'insieme C e termina in un nodo dell'insieme S, il numero di archi che vengono aggiunti è sempre maggiore di quello degli archi che vengono tolti, per cui il nuovo accoppiamento ha un arco in più.

Quest'osservazione suggerisce un algoritmo per cercare un accoppiamento di cardinalità massima: tale algoritmo procede nel modo seguente.

1. Inizializza M ponendolo uguale all'insieme vuoto.
2. Cerca un cammino p aumentante rispetto a M: se tale cammino non esiste, allora restituisce M come accoppiamento finale e termina.
3. Altrimenti, toglie da M tutti gli archi di p che non appartenevano a M, aggiunge a M tutti gli archi di p che non appartenevano a M e ritorna al passo precedente.

Dimostriamo anzitutto che l'accoppiamento restituito da tale algoritmo è il massimo possibile: a tale scopo, dobbiamo dimostrare che se non esiste un cammino aumentante rispetto a M, allora M è massimo. Quest'affermazione discende dal prossimo risultato.

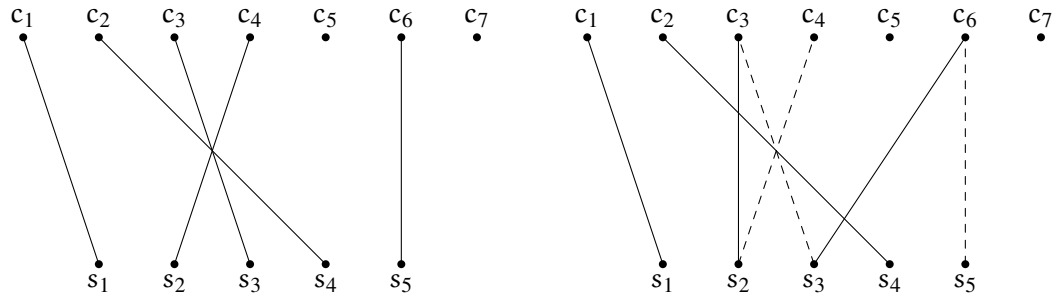
Lemma 9.1

Se $G = (C \cup S, E)$ un grafo bipartito e sia M un accoppiamento per G. Inoltre, sia k il numero di nodi in C non accoppiati. Le seguenti tre affermazioni sono equivalenti.

1. M è un accoppiamento di cardinalità massima.
2. G non ammette un cammino aumentante rispetto a M.
3. Esiste un sottoinsieme X di C per cui $|\mathtt{adj}(X)| = |X| - k$, dove $\mathtt{adj}(X)$ denota l'insieme dei nodi di S adiacenti ad almeno un nodo in S.

Dimostrazione. Chiaramente, (1) implica (2), in quanto se esistesse un cammino aumentante rispetto a M, allora M potrebbe essere migliorato e, quindi, non sarebbe massimo. Per dimostrare che (2) implica (3), sia X il sottoinsieme dei nodi di C raggiungibili da nodi di C non accoppiati mediante cammini alternanti. Tale insieme X include k nodi non accoppiati e j nodi accoppiati con $j \geqslant 0$. Quindi, $|X| = k + j$. Poiché G non ammette un cammino aumentante rispetto a M, tutti i vertici in $\mathtt{adj}(X)$ devono essere accoppiati. Inoltre, tali nodi devono essere accoppiati a un nodo in X, in quanto X include tutti i nodi di C raggiungibili da nodi di C non accoppiati mediante cammini alternanti. Pertanto, $|\mathtt{adj}(X)| = j = |X| - k$. Infine, (3) implica (1) in quanto l'esistenza di un sottoinsieme X di C per cui $|\mathtt{adj}(X)| = |X| - k$ implica che ogni accoppiamento deve necessariamente lasciare non accoppiati almeno k nodi in C. Poiché M è un accoppiamento che lascia non accoppiati esattamente k nodi di C, abbiamo che M deve essere un accoppiamento di cardinalità massima. ◇

Figura 9.2: un esempio di cammino aumentante.

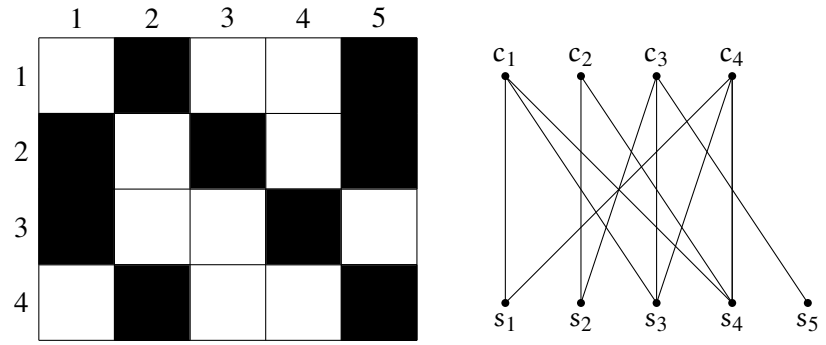


Osserviamo che l'algoritmo precedentemente descritto può eseguire al più $O(|E|)$ volte i passi 2 e 3: infatti, ogni volta che tali due passi vengono eseguiti, la cardinalità dell'accoppiamento aumenta di un'unità e, chiaramente, un accoppiamento non può contenere più di $|E|$ archi. Per dimostrare, quindi, che il problema dell'accoppiamento massimo appartiene alla classe P, rimane da far vedere che il secondo passo dell'algoritmo può essere eseguito in tempo polinomiale (il terzo passo, infatti, può essere banalmente eseguito in tempo polinomiale). A tale scopo, possiamo utilizzare la seguente modifica di una visita in ampiezza del grafo.

1. Inseriamo in coda tutti i nodi C non accoppiati.
2. Fintanto che la coda non è vuota, eseguiamo i seguenti passi.
   (a) Estraiamo un nodo x dalla coda e marchiamolo come visitato.
   (b) Se $x \in C$ e ha un nodo adiacente non accoppiato, abbiamo trovato un cammino aumentante (tale cammino può essere facilmente ricostruito se, ogni qualvolta che inseriamo un nodo in coda per la prima volta, memorizziamo il nodo che lo precede).
   (c) Altrimenti, se $x \in C$ e tutti i suoi nodi adiacenti sono accoppiati, inseriamo in coda tutti questi nodi.
   (d) Altrimenti (ovvero $x \in S$), inseriamo in coda il nodo a cui x è accoppiato (che deve esistere in quanto i nodi di S che sono inseriti in coda sono tutti accoppiati).

È facile verificare che l'esecuzione di tale algoritmo richiede tempo $O(|C||E|)$, in quanto nel peggiore dei casi siamo costretti a eseguire una visita in ampiezza (che

Figura 9.3: una scacchiera e il corrispondente grafo bipartito.



richiede tempo $O(|E|)$ per ogni nodo di C. In conclusione, abbiamo dimostrato che il problema dell'accoppiamento massimo appartiene alla classe P.

9.3.2 Il problema delle torri

Per mostrare un ulteriore esempio di applicazione del concetto di riducibilità polinomiale, consideriamo il seguente problema. Data una scacchiera di n righe e m colonne (con $n \leqslant m$), sulla quale alcune celle sono bianche e altre sono nere (si veda la parte sinistra della Figura 9.3, ci domandiamo se è possibile posizionare sulle celle bianche di tale scacchiera k *torri* senza che queste si possano "mangiare" vicendevolmente (ricordiamo che una torre può mangiare un'altra torre se le due torri si trovano sulla stessa riga o sulla stessa colonna). Questo problema può essere facilmente ridotto al problema del massimo accoppiamento nel modo seguente. Per ogni riga i della scacchiera, creiamo un nodo c_i in C e, per ogni colonna j della scacchiera, creiamo un nodo s_j in S. Infine, creiamo un arco tra il nodo c_i e il nodo s_j se e solo se la cella in riga i e colonna j è bianca (si veda la parte destra della Figura 9.3). Un accoppiamento per tale grafo contenente k archi corrisponde in modo univoco a un posizionamento di k torri sulla scacchiera, in quanto ogni arco dell'accoppiamento corrisponde a una cella bianca della scacchiera e due archi dell'accoppiamento non possono condividere alcun nodo. Viceversa, un posizionamento di k torri sulla scacchiera corrisponde in modo univoco a un accoppiamento per il grafo bipartito contenente k archi, in quanto le torri possono essere posizionate solo su celle bianche e due torri non possono condividere né la riga né la colonna. Chiaramente tale trasformazione può essere calcolata in tempo polinomiale: abbiamo quindi dimostrato che il problema del posizionamento delle torri è polinomialmente riducibile al problema

del massimo accoppiamento. Poiché quest'ultimo problema appartiene alla classe P, abbiamo che anche il problema delle torri appartiene a tale classe.

Esercizi

Esercizio 9.1. Dimostrare che se a e b sono due numeri interi maggiori di zero tali che $a > b$, allora $\gcd(a, b) = \gcd(b, a \bmod b)$.

Esercizio 9.2. Una formula booleana è detta essere *in forma normale disgiuntiva* se una sua clausola è soddisfatta da un'assegnazione τ se *tutti* i suoi letterali lo sono e se la formula è soddisfatta da τ se *almeno* una delle sue clausole lo è. Dimostrare che il problema di decidere se una formula booleana in forma normale disgiuntiva è soddisfacibile appartiene alla classe P.

Esercizio 9.3. Una formula booleana in forma normale congiuntiva è detta essere una **tautologia** se è soddisfatta da una qualunque assegnazione di verità. Dimostrare che il problema di decidere se una formula booleana in forma normale congiuntiva è una tautologia appartiene alla classe P.

Esercizio 9.4. Una formula booleana in forma normale congiuntiva è detta essere *Horn* se ogni clausola contiene al più una variabile non negata. Dimostrare che il problema di decidere se una formula Horn è soddisfacibile appartiene alla classe P, descrivendo un algoritmo che inizialmente assegna il valore `false` a tutte le variabili e, quindi, prosegue modificando tale assegnazione in base alle sue conseguenze.

Esercizio 9.5. Facendo uso del paradigma della programmazione dinamica, dimostrare che il problema di decidere se, dato un insieme A di n numeri interi la cui somma è uguale a $2b \leqslant n^2$, A ammette un sottoinsieme B la somma dei cui elementi è uguale a b, appartiene alla classe P.

Esercizio 9.6. Dimostrare che se $L_1 \leqslant_p L_2$ e L_2 appartiene alla classe P, allora L_1 appartiene alla classe P.

Esercizio 9.7. Dimostrare che una formula booleana ψ può essere trasformata in tempo polinomiale in una formula booleana ψ' in forma normale congiuntiva, tale che ψ è soddisfacibile se e solo se ψ' è soddisfacibile: a tale scopo, conviene osservare che una qualunque formula del tipo $f_1 \vee f_2$ può essere trasformata in $(f_1 \vee y) \wedge (f_2 \vee \neg y)$ dove y è una nuova variabile.

Capitolo 10

La classe NP

SOMMARIO

In questo capitolo considereremo la classe dei problemi verificabili efficientemente, ovvero la classe NP*, e formuleremo la congettura* $P \neq NP$ *che è considerato uno tra i problemi aperti più significativi nel campo dell'informatica teorica. Introdurremo, poi, il concetto di problema* NP*-completo, mostrando diversi esempi di linguaggi* NP*-completi: intuitivamente, tali problemi sono i potenziali candidati a risolvere la questione del rapporto esistente tra* P *e* NP*. Concluderemo il capitolo definendo due ulteriori classi di complessità, ovvero* EXP *e* PSPACE*, e analizzando il loro rapporto con le classi* P *e* NP*.*

10.1 La classe NP

Migliaia di problemi interessanti da un punto di vista applicativo appartengono alla classe P. Tuttavia, è anche vero che migliaia di altri problemi, altrettanto interessanti, sfuggono a una loro risoluzione efficiente. Molti di questi, però, hanno una caratteristica in comune che giustificherà l'introduzione di una nuova classe di linguaggi, più estesa della classe P: per capire tale caratteristica, consideriamo i seguenti quattro esempi di problemi computazionali.

Il problema SAT, come detto, consiste nel decidere se una data formula booleana in forma normale congiuntiva è soddisfacibile. Non si conoscono algoritmi polinomiali che risolvono questo problema, ma possiamo osservare che se una formula φ è soddisfacibile, allora esiste un'assegnazione τ che soddisfa φ: tale τ può essere codificata con una stringa di lunghezza polinomiale nella lunghezza della codifica di φ. Inoltre, esiste un semplice algoritmo polinomiale che verifica se, effettivamente, τ soddisfa φ: tale algoritmo deve semplicemente sostituire alle variabili i valori assegnati da τ e verificare che ogni clausola sia soddisfatta.

Il problema 3-COLORABILITY consiste nel decidere se, dato un grafo $G = (N, E)$, esiste una funzione $f : N \to \{1,2,3\}$ tale che $f(u) \neq f(v)$ se $(u,v) \in E$ o se $(v,u) \in E$.

Non si conoscono algoritmi polinomiali che risolvono questo problema, ma possiamo osservare che se un grafo G è 3-colorabile, allora deve esistere una funzione f che (in modo corretto) colora con tre colori tutti i nodi: tale funzione può essere chiaramente codificata con una stringa di lunghezza polinomiale nella lunghezza della codifica di G. Inoltre, esiste un semplice algoritmo polinomiale che verifica se, effettivamente, f colora con tre colori tutti i nodi (in modo corretto): tale algoritmo deve semplicemente verificare che, per ogni arco (u, v), $f(u) \neq f(v)$.

Il problema HAMILTONIAN PATH consiste nel decidere se, dato un grafo $G = (N, E)$, esiste un cammino che passi per tutti i nodi una e una sola volta. Non si conoscono algoritmi polinomiali che risolvono questo problema, ma possiamo osservare che se un grafo ammette un cammino hamiltoniano, allora esiste una sequenza s di $|N|$ nodi distinti $u_1, \ldots, u_{|N|}$ tale che u_{i-1} è adiacente a u_i, per ogni i con $2 \leqslant i \leqslant |N|$: tale sequenza può essere codificata con una stringa di lunghezza polinomiale nella lunghezza della codifica di G. Inoltre, esiste un semplice algoritmo polinomiale che verifica se, effettivamente, s è un cammino hamiltoniano: tale algoritmo deve semplicemente verificare che i nodi della sequenza siano distinti e che, per ogni i con $2 \leqslant i \leqslant n$, $(u_{i-1}, u_i) \in E$.

Il problema PARTITION consiste nel decidere se, dato un insieme A di n numeri interi $a_1, \ldots, a_n$, esiste una partizione di A in due sottoinsiemi A_1 e A_2 tale che la somma dei numeri in A_1 sia uguale a quella dei numeri in A_2. Non si conoscono algoritmi polinomiali che risolvono questo problema, ma possiamo osservare che se un insieme A ammette una partizione perfetta, allora esiste un sottoinsieme I dell'insieme $\{1, \ldots, n\}$ tale che $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$: tale insieme può essere codificato con una stringa di lunghezza polinomiale nella lunghezza della codifica di A. Inoltre, esiste un semplice algoritmo polinomiale che verifica se, effettivamente, I è una partizione perfetta: tale algoritmo deve semplicemente verificare che $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.

Questi quattro esempi (descritti in modo volutamente ripetitivo) suggeriscono il fatto che esistono problemi per i quali non conosciamo algoritmi efficienti in grado di decidere se esiste una soluzione, ma per i quali siamo sicuri che se tale soluzione esiste essa può essere descritta in modo relativamente succinto e può essere verificata in modo efficiente. Quest'osservazione conduce alla seguente definizione.

Definizione 10.1: la classe NP

La classe NP è l'insieme di tutti i linguaggi L per cui esiste un polinomio p e una macchina di Turing V con complessità temporale polinomiale tali che, per ogni x, $x \in L$ se e solo se esiste y per cui $|y| \leqslant p(|x|)$ e $V(x, y)$ termina in uno stato finale.

La stringa y che appare nella precedente definizione è detta essere un **certificato** dell'appartenenza di x a L. Da quanto sopra esposto, abbiamo che i quattro problemi

SAT, 3-COLORABILITY, HAMILTONIAN PATH e PARTITION appartengono alla classe NP: in realtà, migliaia di altri problemi godono della proprietà di ammettere certificati "brevi" e efficientemente verificabili. In particolare, ciò è vero per tutti i problemi inclusi nella classe P, come mostrato dal seguente risultato.

Teorema 10.1

La classe P è contenuta nella classe NP.

Dimostrazione. Sia $L \in P$ e sia T una macchina di Turing con complessità temporale polinomiale che decide L. Pertanto, per ogni stringa x, $T(x)$ termina in uno stato finale se e solo se $x \in L$ e $T(x)$ termina dopo aver eseguito al più un numero polinomiale di passi $p(|x|)$, per qualche polinomio p. Quindi, $x \in L$ se e solo se esiste una computazione di T con input x che è costituita da al più $p(|x|)$ configurazioni e che termina in uno stato finale. Poiché, a ogni passo, al più una nuova cella del nastro viene esaminata, ogni configurazione può essere codificata con una stringa di lunghezza polinomiale nella lunghezza di x: quindi, la computazione di T con input x può essere codificata con una stringa di lunghezza polinomiale nella lunghezza di x. Sia V una macchina di Turing che, date in input due stringhe x e y, termina in uno stato finale se e solo se y è la codifica della computazione $T(x)$ che termina in uno stato finale: è facile verificare che la complessità temporale di V è polinomiale. Allora, per ogni stringa x, $x \in L$ se e solo se esiste una stringa y, con $|y| \leqslant q(|x|)$, per cui $V(x, y)$ termina in uno stato finale, dove $q(n)$ è il polinomio che limita la lunghezza della codifica di una computazione di T con input una stringa di lunghezza n. $\diamond$

Una domanda che, a questo punto, sorge in modo naturale è se P è diversa da NP. Tale domanda, che fu posta in modo esplicito nel 1975 (ma che sembra risalire a una lettera che Gödel scrisse a von Neumann negli anni cinquanta), risulta tuttora essere senza risposta ed è considerata uno dei principali problemi matematici del secolo scorso, al punto che una "taglia" di un milione di dollari è stata posta sulla sua risoluzione. Informalmente, tale problema è equivalente a chiedersi se trovare una soluzione di un problema è, in generale, più difficile che verificare se una soluzione proposta è corretta. Chiunque si sia posto una simile domanda sa che la risposta più naturale è quella affermativa: per questo motivo, il problema è stato quasi sempre presentato come quello di trovare la dimostrazione della cosiddetta **congettura** $P \neq NP$ la quale, appunto, afferma che la classe P è diversa dalla classe NP.

10.2 Linguaggi NP-completi

Intuitivamente, un linguaggio NP-completo è tra i più difficili della classe NP, nel senso che se appartenesse alla classe P, allora l'intera classe NP sarebbe inclusa nella classe P. Abbiamo già introdotto uno strumento adatto a comparare la difficoltà di

due linguaggi relativamente al tempo di calcolo necessario per la loro risoluzione: si tratta, infatti, della riducibilità polinomiale. Questo ci porta a fornire la seguente definizione.

Definizione 10.2: linguaggi NP-completi

Un linguaggio L è NP**-completo** se L appartiene a NP e se ogni altro linguaggio in NP è polinomialmente riducibile a L.

È naturale a questo punto chiedersi se esistono linguaggi NP-completi (anche se il lettore avrà già intuito la risposta a tale domanda). Ma prima di discutere l'esistenza di un linguaggio NP-completo, osserviamo che una volta dimostratane l'esistenza, possiamo sfruttare la proprietà di transitività della riducibilità polinomiale per estendere l'insieme dei linguaggi siffatti. La definizione di riducibilità soddisfa infatti la seguente proprietà: se L_0 è polinomialmente riducibile a L_1 e L_1 è polinomialmente riducibile a L_2, allora L_0 è polinomialmente riducibile a L_2. A questo punto, volendo dimostrare che un certo linguaggio L è NP-completo, possiamo procedere in tre passi: prima dimostriamo che L appartiene a NP mostrando l'esistenza del suo certificato polinomiale; poi individuiamo un altro linguaggio L', che già sappiamo essere NP-completo; infine, riduciamo polinomialmente L' a L.

10.2.1 Teorema di Cook-Levin

Per applicare la strategia sopra esposta dobbiamo necessariamente trovare un primo linguaggio NP-completo: il **teorema di Cook-Levin** afferma che SAT è NP-completo. Abbiamo già visto che SAT appartiene a NP: la parte difficile del teorema di Cook-Levin consiste nel mostrare che ogni linguaggio in NP è polinomialmente riducibile a SAT. Prima di dare la dimostrazione del suddetto teorema, forniamo una breve descrizione dell'approccio utilizzato. Dato un linguaggio $L \in$ NP, sappiamo che esiste un algoritmo con complessità temporale polinomiale V e un polinomio p tali che, per ogni stringa x, se $x \in L$, allora esiste una stringa y di lunghezza non superiore a $p(|x|)$ tale che V con x e y in ingresso termina in uno stato finale, mentre se $x \notin L$, allora, per ogni sequenza y, V con x e y in ingresso termina in uno stato non finale. L'idea della dimostrazione consiste nel costruire, per ogni x, in tempo polinomiale una formula booleana φ_x le cui uniche variabili *libere* sono $p(|x|)$ variabili $y_0, y_1, \ldots, y_{p(|x|)-1}$, intendendo con ciò che la soddisfacibilità della formula dipende solo dai valori assegnati a tali variabili: intuitivamente, la variabile y_i corrisponde al valore del bit in posizione i all'interno della stringa y (senza perdita di generalità, possiamo assumere che la lunghezza di y sia esattamente uguale a $p(|x|)$). La formula φ_x in un certo senso *simula* il comportamento di V con x e y in ingresso ed è soddisfacibile solo se tale

computazione termina in uno stato finale (ovvero, se y è un certificato che testimonia che x appartiene a L). Il fatto che possiamo costruire una tale formula non dovrebbe sorprenderci più di tanto, se consideriamo che, in fin dei conti, l'esecuzione di un algoritmo all'interno di un calcolatore avviene attraverso circuiti logici le cui componenti di base sono porte logiche che realizzano la disgiunzione, la congiunzione e la negazione.

Teorema 10.2

SAT è NP-completo.

Dimostrazione. Sia L un linguaggio in NP e siano p e V come nella definizione di NP, ovvero tali che, per ogni stringa x, valga la seguente affermazione.

$$x \in L \text{ se e solo se } \exists y[|y| = p(|x|) \wedge V(x,y) \text{ termina in uno stato finale}]$$

(osserviamo che non è restrittivo assumere che il certificato abbia esattamente lunghezza pari a $p(|x|)$). Sia poi q il polinomio che limita il tempo di calcolo di V ovvero, con input x e y, $V(x,y)$ esegue al più $q(|x|)$ passi. Per semplicità, assumiamo che l'alfabeto di lavoro di V sia $\{\sigma_0 = \square, \sigma_1 = 0, \sigma_2 = 1\}$, che gli stati di V siano $q_0, q_1, \ldots, q_k$, che q_0 sia lo stato iniziale e che q_1 sia l'unico stato finale. Assumiamo inoltre che il nastro di V sia semi-infinito e che la testina possa solo spostarsi a destra oppure a sinistra. Come già detto in precedenza, l'idea della dimostrazione è quella di simulare, per ogni x, attraverso le assegnazioni di verità a una formula booleana (non necessariamente in forma normale congiuntiva) la computazione $V(x,y)$ con y da determinare. A tale scopo, faremo uso delle seguenti variabili (nel seguito, n indica la lunghezza di x).

- $P^i_{s,t}$: tale variabile ha il valore `true` se e solo se la cella s contiene il simbolo σ_i al tempo t ($0 \leqslant i \leqslant 2, 0 \leqslant s \leqslant q(n), 0 \leqslant t \leqslant q(n)$).
- Q^i_t: tale variabile ha il valore `true` se e solo se V è nello stato q_i al tempo t ($0 \leqslant i \leqslant k, 0 \leqslant t \leqslant q(n)$)
- $S_{s,t}$: tale variabile ha il valore `true` se e solo se la testina è posizionata sulla cella s al tempo t ($0 \leqslant s \leqslant q(n), 0 \leqslant t \leqslant q(n)$)

La formula booleana globale è costituita dalla congiunzione delle seguenti formule booleane, il cui scopo è quello di verificare specifiche caratteristiche della computazione di V con input x e y (da determinare).

Posizione della testina Se soddisfatta, la formula A afferma che, in ogni istante t, la testina è posizionata esattamente su una cella. In particolare,

$$A = A_0 \wedge A_1 \wedge \ldots \wedge A_{q(n)}$$

dove

$$\begin{aligned} A_t &= (S_{0,t} \vee S_{1,t} \vee \ldots \vee S_{q(n),t}) \\ &\wedge \left(\bigwedge_{(i,j)} : 0 \leqslant i < q(n), i < j \leqslant q(n) \right) [S_{i,t} \to \neg S_{j,t}] \end{aligned}$$

Contenuto di una cella Se soddisfatta, la formula B afferma che, in ogni istante t, ogni cella contiene esattamente un simbolo. In particolare,

$$B = B_{0,0} \wedge \ldots \wedge B_{q(n),0} \wedge B_{0,1} \wedge \ldots \wedge B_{q(n),1} \wedge \ldots \wedge B_{0,q(n)} \wedge \ldots \wedge B_{q(n),q(n)}$$

dove

$$\begin{aligned} B_{s,t} &= (P^0_{s,t} \vee P^1_{s,t} \vee P^2_{s,t}) \\ &\wedge \left(\bigwedge_{(i,j)} : 0 \leqslant i < 2, i < j \leqslant 2 \right) [P^i_{s,t} \to \neg P^j_{s,t}] \end{aligned}$$

Stato Se soddisfatta, la formula C afferma che, in ogni istante t, V si trova in un solo stato. In particolare,

$$C = C_1 \wedge C_2 \wedge \ldots \wedge C_{q(n)}$$

dove

$$\begin{aligned} C_t &= (Q^0_t \vee \ldots \vee Q^k_t) \\ &\wedge \left(\bigwedge_{(i,j)} : 0 \leqslant i < k, i < j \leqslant k \right) [Q^i_t \to \neg Q^j_t] \end{aligned}$$

Input Se soddisfatta, la formula D_x afferma che, all'istante 0, le prime n celle contengono la stringa $x = \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_n}$ e che la cella successiva contiene $\square$. In particolare,

$$D_x = P^{i_1}_{0,0} \wedge \ldots \wedge P^{i_n}_{n-1,0} \wedge P^0_{n,0}$$

Certificato Se soddisfatta, la formula D^y afferma che, all'istante 0, le successive $m = p(n)$ celle contengono una stringa binaria. In particolare,

$$D^y = D^y_{n+1} \wedge D^y_{n+2} \wedge \ldots \wedge D^y_{n+p(n)}$$

dove

$$D^y_s = (P^1_{s,0} \vee P^2_{s,0}) \wedge (\neg P^1_{s,0} \vee \neg P^2_{s,0})$$

Blank Se soddisfatta, la formula $D^{\square}$ afferma che, all'istante 0, le successive celle contengono $\square$. In particolare,

$$D^{\square} = P^0_{n+m+1,0} \wedge P^0_{n+m+2,0} \wedge \cdots \wedge P^0_{q(n),0}$$

Stato iniziale Se soddisfatta, la formula E afferma che, all'istante 0, lo stato è quello iniziale e che la testina è posizionata sulla prima cella. In particolare,

$$E = Q^0_0 \wedge S_{0,0}$$

Stato finale Se soddisfatta, la formula F afferma che, a un certo istante, V termina nello stato finale. In particolare,

$$F = Q^1_0 \vee Q^1_1 \vee \ldots \vee Q^1_{q(n)}$$

Transizioni Se soddisfatta, G afferma che, a ogni istante, V esegue una transizione legittima. Chiaramente G dipende dal grafo delle transizioni di V. Supponiamo, ad esempio, che, leggendo σ_j, V va dallo stato q_i allo stato q_{i_1}, scrive σ_{j_1} e esegue il movimento a destra. Allora, per ogni istante t e per ogni cella s, G include la formula

$$(Q^i_t \wedge P^j_{s,t} \wedge S_{s,t}) \rightarrow (Q^{i_1}_{t+1} \wedge P^{j_1}_{s,t+1} \wedge S_{s+1,t+1})$$

e, poiché dobbiamo anche assicurarci che le altre celle non cambino contenuto, la formula

$$P^j_{s,t} \wedge \neg S_{s,t} \rightarrow P^j_{s,t+1}$$

Dalla costruzione segue immediatamente che se esiste una stringa y, con $|y| = p(n)$, tale che $V(x,y)$ termina in q_1, allora a partire da y e dalla computazione $V(x,y)$ possiamo derivare un'assegnazione che soddisfa la formula. Viceversa, da un'assegnazione che soddisfa la formula possiamo ricavare dal valore delle variabili che appaiono in D^y un certificato y tale che $V(x,y)$ termina in q_1. La costruzione della formula può chiaramente essere fatta in tempo polinomiale. Osserviamo, però, che la formula non è in forma normale congiuntiva: tuttavia, possiamo facilmente dimostrare che una formula booleana ψ può essere trasformata in tempo polinomiale in una formula booleana ψ' in forma normale congiuntiva, tale che ψ è soddisfacibile se e solo se ψ' è soddisfacibile (si veda l'Esercizio 9.7). Pertanto, abbiamo dimostrato che L è polinomialmente riducibile a SAT: poiché L era un generico linguaggio in NP, il teorema di Cook-Levin risulta essere dimostrato. $\diamond$

Una volta dimostrata l'esistenza di un primo linguaggio NP-completo, possiamo ora mostrare la NP-completezza di altri linguaggi partendo da SAT. Questo è quanto faremo nel resto di questo capitolo, riesaminando i quattro problemi descritti precedentemente.

10.2.2 Soddisfacibilità di clausole con tre letterali

Questo problema è la restrizione di SAT, in cui ogni clausola contiene esattamente tre letterali: dimostriamo che SAT è polinomialmente riducibile a 3-SAT. Sia $C = \{c_0, \ldots, c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di variabili booleane $\{x_0, \ldots, x_{n-1}\}$. Vogliamo costruire, in tempo polinomiale, un nuovo insieme D di clausole, ciascuna di cardinalità 3, costruite a partire da un insieme Z di variabili booleane e tali che C è soddisfacibile se e solo se D è soddisfacibile. A tale scopo usiamo una tecnica di trasformazione detta di **sostituzione locale**, in base alla quale costruiremo D e Z sostituendo ogni clausola $c \in C$ con un sottoinsieme D_c di D in modo indipendente dalle altre clausole di C: l'insieme D è quindi uguale a $\cup_{c \in C} D_c$ e Z è l'unione di tutte le variabili booleane che appaiono in D. Data una clausola $c = \{l_0, \ldots, l_{k-1}\}$ dell'insieme C, definiamo D_c distinguendo i seguenti quattro casi.

- $k = 1$: in questo caso,

$$D_c = \{\{l_0, y_0^c, y_1^c\}, \{l_0, y_0^c, \neg y_1^c\}, \{l_0, \neg y_0^c, y_1^c\}, \{l_0, \neg y_0^c, \neg y_1^c\}\}$$

  Osserviamo che le quattro clausole in D_c sono soddisfatte se e solo se l_0 è soddisfatto.

- $k = 2$: in questo caso, $D_c = \{\{l_0, l_1, y_0^c\}, \{l_0, l_1, \neg y_0^c\}$. Osserviamo che le due clausole in D_c sono soddisfatte se e solo se l_0 oppure l_1 è soddisfatto.

- $k = 3$: in questo caso, D_c è formato dalla sola clausola c.

- $k > 3$: in questo caso,

$$D_c = \{\{l_0, l_1, y_0^c\}, \{\neg y_0^c, l_2, y_1^c\}, \{\neg y_1^c, l_3, y_2^c\}, \ldots, \{\neg y_{k-4}^c, l_{k-2}, l_{k-1}\}\}$$

Dalla definizione di D_c abbiamo che

$$Z = X \cup \bigcup_{c \in C \wedge |c|=1} \{y_0^c, y_1^c\} \cup \bigcup_{c \in C \wedge |c|=2} \{y_0^c\} \cup \bigcup_{c \in C \wedge |c|>3} \{y_0^c, \ldots, y_{|c|-4}^c\}$$

Inoltre, è chiaro che la costruzione dell'istanza di 3-SAT può essere eseguita in tempo polinomiale. Supponiamo che esista un'assegnazione τ di verità alle variabili di X che soddisfa C. Quindi, τ soddisfa c per ogni clausola $c \in C$: mostriamo che tale assegnazione può essere estesa alle nuove variabili di tipo y^c introdotte nel definire D_c, in modo che tutte le clausole in esso contenute siano soddisfatte (da quanto detto sopra, possiamo supporre che $|c| > 3$). Poiché c è soddisfatta da τ, deve esistere h tale che τ soddisfa l_h con $0 \leqslant h \leqslant |c| - 1$: estendiamo τ assegnando il valore `true`

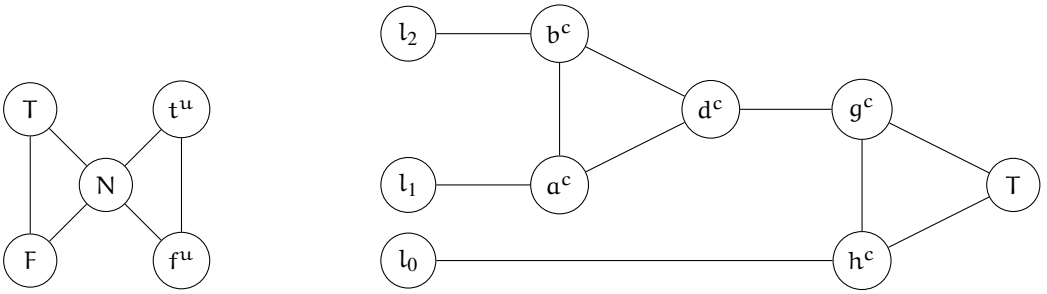
a tutte le variabili y_i^c con $0 \leqslant i \leqslant h-2$ e il valore `false` alle rimanenti variabili di tipo y^c. In questo modo, siamo sicuri che la clausola di D_c contenente l_h è soddisfatta (da l_h stesso), le clausole che la precedono sono soddisfatte grazie al loro terzo letterale e quelle che la seguono lo sono grazie al loro primo letterale. Viceversa, supponiamo che esista un'assegnazione τ di verità alle variabili di Z che soddisfi tutte le clausole in D e, per assurdo, che tale assegnazione ristretta alle sole variabili di X non soddisfi almeno una clausola $c \in C$, ovvero che tutti i letterali contenuti in c non siano soddisfatti (di nuovo, ipotizziamo che $|c| > 3$). Ciò implica che tutte le variabili di tipo y^c devono essere vere, perché altrimenti una delle prime $|c|-3$ clausole in D_c non è soddisfatta, contraddicendo l'ipotesi che τ soddisfa tutte le clausole in D. Quindi, $\tau(y^c_{|c|-4}) = \texttt{true}$, ovvero $\tau(\neg y^c_{|c|-4}) = \texttt{false}$: poiché abbiamo supposto che anche $l_{|c|-2}$ e $l_{|c|-1}$ non sono soddisfatti, l'ultima clausola in D_c non è soddisfatta, contraddicendo nuovamente l'ipotesi che τ soddisfa tutte le clausole in D. In conclusione, abbiamo dimostrato che C è soddisfacibile se e solo se D lo è e, quindi, che il linguaggio SAT è riducibile in tempo polinomiale al linguaggio 3-SAT: quindi, quest'ultimo è NP-completo. Notiamo che, in modo simile a quanto fatto per 3-SAT, possiamo mostrare la NP-completezza del problema della soddisfacibilità nel caso in cui le clausole contengano esattamente k letterali, per ogni $k \geqslant 3$: tale affermazione non si estende però al caso in cui $k = 2$, in quanto, come abbiamo visto in precedenza, in questo caso il problema diviene risolvibile in tempo polinomiale e, quindi, difficilmente esso è anche NP-completo.

La NP-completezza di 3-SAT ha un duplice valore: da un lato esso mostra che la difficoltà computazionale del problema della soddisfacibilità non dipende dalla lunghezza delle clausole (fintanto che queste contengono almeno tre letterali), dall'altro ci consente nel seguito di usare 3-SAT come linguaggio di partenza, il quale, avendo istanze più regolari, è più facile da utilizzare per sviluppare riduzioni volte a dimostrare risultati di NP-completezza.

10.2.3 Colorabilità di un grafo con tre colori

Dimostriamo che 3-SAT è polinomialmente riducibile a 3-COLORABILITY, facendo uso di una tecnica di riduzione più sofisticata di quella vista nel paragrafo precedente, che viene generalmente indicata con il nome di **progettazione di componenti** e che opera definendo, per ogni variabile, una componente (*gadget*) del grafo il cui scopo è quello di modellare l'assegnazione di verità alla variabile e, per ogni clausola, una componente il cui scopo è quello di modellare la soddisfacibilità della clausola. I due insiemi di componenti sono poi collegati tra di loro per garantire che l'assegnazione alle variabili soddisfi tutte le clausole. In particolare, sia $C = \{c_0, \ldots, c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di n variabili booleane

Figura 10.1: riduzione da 3-SAT a 3-COLORABILITY.



$\{x_0, \ldots, x_{n-1}\}$. Per ogni variabile u definiamo il gadget mostrato nella parte sinistra della Figura 10.1 (in cui i nodi N, F e T sono gli stessi per tutte le variabili), mentre per ogni clausola $c = \{l_0, l_1, l_2\}$ definiamo il gadget mostrato nella parte destra della figura (in cui il nodo T e i nodi l_0, l_1 e l_2 sono gli stessi di quelli inclusi nei gadget corrispondenti alle variabili).

È chiaro che la costruzione dell'istanza di 3-COLORABILITY può essere eseguita in tempo polinomiale. Supponiamo che esista un'assegnazione τ di verità alle variabili di X che soddisfa C. Quindi, τ soddisfa $c = \{l_0, l_1, l_2\}$ per ogni clausola $c \in C$: mostriamo come, a partire da tale assegnazione, sia possibile costruire una colorazione f del grafo con tre colori. Anzitutto, poniamo $f(T) = 1$, $f(F) = 2$ e $f(N) = 3$. Quindi, per ogni variabile u per cui $\tau(u) = \texttt{true}$, poniamo $f(t^u) = 1$ e $f(f^u) = 2$ e, per ogni variabile u per cui $\tau(u) = \texttt{false}$, poniamo $f(t^u) = 2$ e $f(f^u) = 1$. Poiché c è soddisfatta da τ, deve esistere h tale che τ soddisfa l_h con $0 \leqslant h \leqslant 2$: ciò implica che il nodo corrispondente a l_h è colorato con il colore 1. Possiamo estendere la colorazione dei rimanenti nodi contenuti nel gadget corrispondente a c nel modo indicato nella seguente tabella.

l_0	l_1	l_2	a^c	b^c	d^c	g^c	h^c
1	2	2	1	3	2	3	2
2	1	2	2	1	3	2	3
1	1	2	2	1	3	2	3
2	2	1	1	2	3	2	3
1	2	1	1	2	3	2	3
2	1	1	2	3	1	2	3
1	1	1	2	3	1	2	3

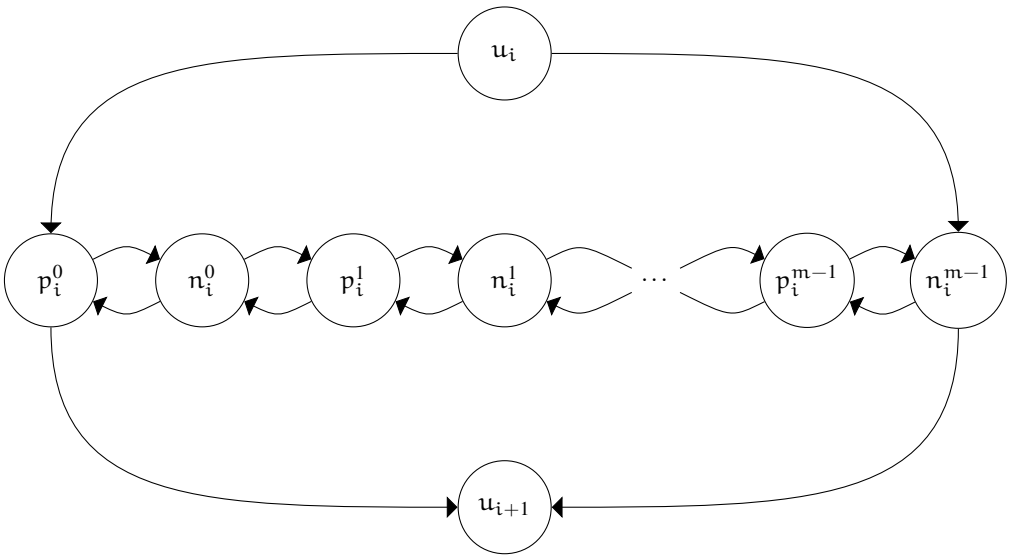
Viceversa, supponiamo che esista una colorazione f del grafo con tre colori e definiamo un'assegnazione τ di verità alle variabili di X nel modo seguente: per ogni

variabile $u \in X$, se $f(t^u) = 1$, allora $\tau(u) = \texttt{true}$, altrimenti $\tau(u) = \texttt{false}$. Supponiamo, per assurdo, che tale assegnazione non soddisfi almeno una clausola $c \in C$, ovvero che tutti i letterali contenuti in c non siano soddisfatti. Ciò implica che tutti i nodi corrispondenti a tali letterali e contenuti nel gadget corrispondente a c sono colorati con il colore 2. Quindi, $f(h^c) = 3$ e $f(g^c) = 2$: ciò implica che $f(d^c) \neq 2$ e, quindi, che $f(a^c) = 2$ oppure $f(b^c) = 2$, contraddicendo l'ipotesi che f sia una colorazione corretta del grafo. In conclusione, abbiamo dimostrato che C è soddisfacibile se e solo se il grafo è colorabile con tre colori e, quindi, che il linguaggio 3-SAT è riducibile in tempo polinomiale al linguaggio 3-COLORABILITY: quindi, quest'ultimo è NP-completo.

10.2.4 Cammino hamiltoniano

Dimostriamo che 3-SAT è polinomialmente riducibile a HAMILTONIAN PATH, facendo uso della tecnica di riduzione per progettazione di componenti. A tale scopo, sia $C = \{c_0, \ldots, c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di n variabili booleane $\{x_0, \ldots, x_{n-1}\}$. Vogliamo costruire, in tempo polinomiale, un grafo G tale che C è soddisfacibile se e solo se G include un cammino hamiltoniano. Per ogni variabile x_i con $0 \leqslant i < n$, definiamo il gadget mostrato nella Figura 10.2, in cui il numero di nodi inclusi nella diagonale del rombo è pari a due volte il numero delle clausole. In particolare, per ogni variabile x_i, introduciamo in G un nodo u_i e, per ogni variabile x_i e per ogni clausola c_j, introduciamo in G due nodi p_i^j e n_i^j. Inoltre, per ogni variabile x_i, aggiungiamo a G gli archi (u_i, p_i^0), (u_i, n_i^{m-1}), (p_i^0, u_{i+1}) e (n_i^{m-1}, u_{i+1}), dove il nodo u_n è un nuovo nodo di G. Infine, per ogni variabile x_i e per ogni clausola c_j, aggiungiamo a G gli archi (p_i^j, n_i^j), (n_i^j, p_i^j) e (n_i^j, p_i^{j+1}) (ad eccezione, ovviamente, dell'arco (n_i^{m-1}, p_i^m) che non può essere aggiunto in quanto il nodo p_i^m non esiste). Ciò completa la costruzione dei gadget che modellano l'assegnazione di verità alle variabili. Per ogni clausola c_j, definiamo poi il gadget che modella la soddisfacibilità della clausola come costituito da un solo nodo v_j. I due insiemi di gadget sono collegati tra di loro nel modo seguente (si veda la Figura 10.3). Se una variabile x_i appare affermata nella clausola c_j allora aggiungiamo un arco dal nodo p_i^j al nodo v_j e uno dal nodo v_j al nodo n_i^j (come mostrato nella parte sinistra della figura). Viceversa, se una variabile x_i appare negata nella clausola c_j allora aggiungiamo un arco dal nodo n_i^j al nodo v_j e uno dal nodo v_j al nodo p_i^j (come mostrato nella parte destra della figura). Intuitivamente, ogni cammino hamiltoniano dovrà decidere se visitare la diagonale del rombo di una variabile x_i da sinistra verso destra oppure da destra verso sinistra: il primo caso corrisponde all'aver assegnato il valore $\texttt{true}$ a x_i e, quindi, alla possibilità di soddisfare tutte le clausole c_j in cui x_i appare affermata, mentre il secondo caso corrisponde ad aver assegnato

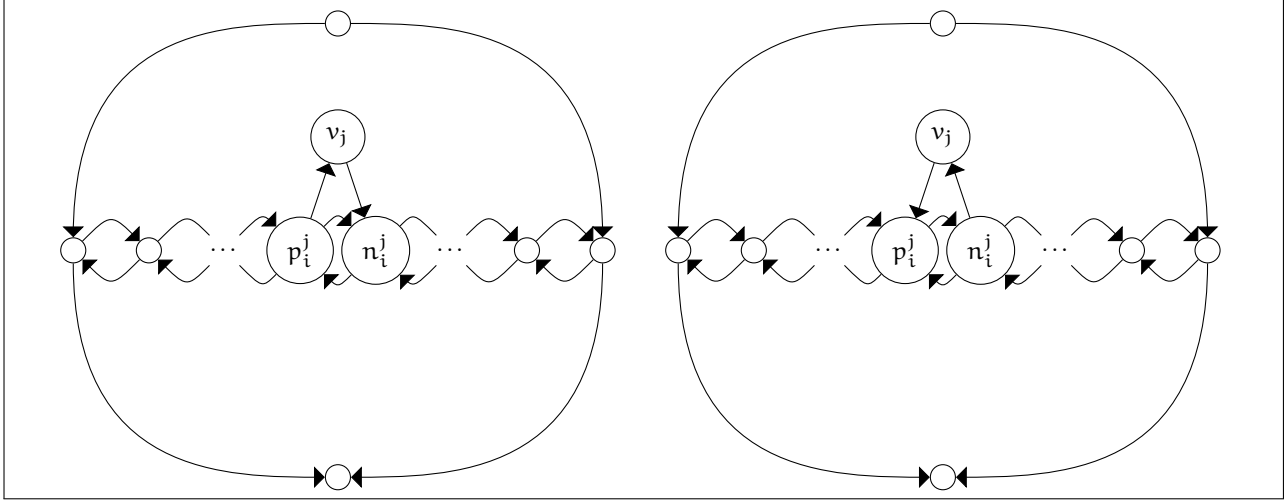
Figura 10.2: il gadget associato alla variabile x_i nella riduzione da 3-SAT a HAMILTONIAN PATH.



il valore `false` a x_i e, quindi, alla possibilità di soddisfare tutte le clausole c_j in cui x_i appare negata. Ancora una volta, è chiaro che la costruzione dell'istanza di HAMILTONIAN PATH può essere eseguita in tempo polinomiale. Dimostriamo ora formalmente che la formula booleana è soddisfacibile se e solo se il grafo risultante dalla riduzione ammette un cammino hamiltoniano. Supponiamo che esista un'assegnazione τ di verità alle variabili di X che soddisfa C. Quindi, τ soddisfa c_j per ogni clausola $c_j \in C$: sia l_j il primo dei letterali in c_j a cui τ assegna il valore `true` e sia $x_{i(j)}$ la variabile corrispondente a l_j. Possiamo costruire un cammino hamiltoniano in G nel modo seguente. Per ogni nodo u_i con $0 \leqslant i \leqslant n-1$, distinguiamo i seguenti due casi.

1. $\tau(x_i) = \texttt{true}$. In questo caso, includiamo nel cammino hamiltoniano l'arco (u_i, p_i^0) e l'arco (n_i^{m-1}, u_{i+1}). Inoltre, per ogni j con $0 \leqslant j \leqslant m-1$ tale che $i \neq i(j)$, includiamo nel cammino gli archi (p_i^j, n_i^j) e, per ogni j con $0 \leqslant j < m-1$, includiamo nel cammino gli archi (n_i^j, p_i^{j+1}). Infine, per ogni j con $0 \leqslant j \leqslant m-1$ tale che $i = i(j)$, includiamo nel cammino gli archi (p_i^j, v_j) e (v_j, n_i^j). In altre parole, alle variabili il cui valore di verità è `true` facciamo visitare la diagonale del corrispondente rombo da sinistra verso destra e, ogni qualvolta incontriamo una clausola il cui valore di verità è determinato da tale

Figura 10.3: connessione tra gadget associati a variabili e gadget associati a clausole.



variabile, eseguiamo una "deviazione" passando per il nodo corrispondente alla clausola.

2. $\tau(x_i) = \texttt{false}$. In tal caso, includiamo nel cammino l'arco (u_i, n_i^{m-1}) e l'arco (p_i^0, u_{i+1}). Inoltre, per ogni j con $0 \leqslant j \leqslant m-1$ tale che $i \neq i(j)$, includiamo nel cammino gli archi (n_i^j, p_i^j) e, per ogni j con $0 < j \leqslant m-1$, includiamo nel cammino gli archi (p_i^j, n_i^{j-1}). Infine, per ogni j con $0 \leqslant j \leqslant m-1$ tale che $i = i(j)$, includiamo nel cammino gli archi (n_i^j, v_j) e (v_j, p_i^j). In altre parole, alle variabili il cui valore di verità è `false` facciamo visitare la diagonale del corrispondente rombo da destra verso sinistra e, ogni qualvolta incontriamo una clausola il cui valore di verità è determinato da tale variabile, eseguiamo una "deviazione" passando per il nodo corrispondente alla clausola.

Chiaramente, il cammino così costruito visita tutti i nodi inclusi nei gadget corrispondenti alle variabili una e una sola volta. Inoltre, poiché ogni clausola è soddisfatta da τ, anche i nodi inclusi nei gadget corrispondenti alle clausole sono visitati dal cammino. Avendo, infine, selezionato un solo letterale vero all'interno di ogni clausola, abbiamo che questi nodi sono anch'essi visitati una sola volta.

Viceversa, supponiamo che il grafo ammetta un cammino hamiltoniano e definiamo un'assegnazione τ di verità alle variabili di X nel modo seguente: per ogni variabile $u \in X$, se l'arco (u_i, p_i^0) è incluso nel cammino, allora $\tau(u) = \texttt{true}$, altrimenti

$\tau(u) = \texttt{false}$ (osserviamo che, poiché in un cammino hamiltoniano un nodo viene visitato una e una sola volta, allora τ è effettivamente un'assegnazione di verità). Chiaramente, il nodo u_0 deve essere il primo nodo visitato dal cammino hamiltoniano (non avendo alcun arco entrante) e il nodo u_n deve essere l'ultimo nodo visitato dal cammino (non avendo alcun nodo uscente): dimostriamo, ora, che, per ogni i con $0 \leqslant i < n-1$ e per ogni j con $0 \leqslant j \leqslant m-1$, i nodi p_i^j e n_i^j sono visitati dal cammino prima dei nodi u_{i+k}, p_{i+k}^j e n_{i+k}^j, per ogni k con $1 \leqslant k \leqslant n-i-1$. In effetti, se ciò non accade per un certo i e per un certo j, allora il cammino deve entrare nel nodo v_j provenendo dalla diagonale del rombo della variabile x_i e ne deve uscire entrando nella diagonale del rombo della variabile x_{i+k}. Supponiamo che l'arco (p_i^j, v_j) faccia parte del cammino hamiltoniano: allora, l'unico modo che il cammino ha di visitare il nodo n_i^j consiste nell'arrivarci da destra (non potendoci più arrivare dal nodo v_j che è già stato visitato). In questo caso, però, una volta entrato nel nodo n_i^j, il cammino non ha alcun modo di uscirne (essendo entrambi i nodi a esso collegati già stati visitati) e, d'altra parte, n_i^j non può essere l'ultimo nodo del cammino hamiltoniano. In altre parole, abbiamo dimostrato che il cammino hamiltoniano visita le diagonali dei rombi delle n variabili (da sinistra verso destra oppure da destra verso sinistra) una dopo l'altra. Per poter visitare anche i nodi corrispondenti alle clausole, il cammino deve eseguire delle "deviazioni" entrando in ciascuno di tali nodi a partire da un nodo corrispondente all'occorrenza nella clausola di una variabile a cui è stato assegnato il valore `true` oppure a partire da un nodo corrispondente all'occorrenza nella clausola della negazione di una variabile a cui è stato assegnato il valore `false`: in entrambi i casi, l'assegnazione τ soddisfa la clausola e, quindi, l'intera formula.

In conclusione, abbiamo dimostrato che C è soddisfacibile se e solo se il grafo ammette un cammino hamiltoniano e, quindi, che il linguaggio 3-SAT è riducibile in tempo polinomiale al linguaggio HAMILTONIAN PATH: quindi, quest'ultimo è NP-completo.

10.2.5 Partizione

Per dimostrare la NP-completezza di PARTITION, introduciamo prima un altro problema, detto SUBSET SUM, che consiste nel decidere se, dato un insieme A di numeri interi e un numero intero s, esiste un sottoinsieme di A tale che la somma dei suoi numeri sia uguale a s. Dimostriamo che 3-SAT è polinomialmente riducibile a SUBSET SUM, facendo uso della tecnica di riduzione per progettazione di componenti. In particolare, sia $C = \{c_0, \ldots, c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di n variabili booleane $\{x_0, \ldots, x_{n-1}\}$. Per ogni variabile x_i con $0 \leqslant i < n$, definiamo i seguenti due numeri p_i e n_i di $n+m$ cifre decimali: p_i ha un 1 in posizione i (da sinistra) e in ogni posizione $n+j$ (da sinistra) per cui x_i appare

nella clausola c_j, con $0 \leqslant j < m$, mentre n_i ha un 1 in posizione i (da sinistra) e in ogni posizione $n+j$ (da sinistra) per cui $\neg u_i$ appare nella clausola c_j, con $0 \leqslant j < m$. Per ogni clausola c_j con $0 \leqslant j < m$, invece, definiamo due numeri uguali x_j e y_j di $n+m$ cifre decimali, tali che x_j e y_j hanno un 1 in posizione $n+j$ (da sinistra). Il numero s, infine, è un numero di $n+m$ cifre decimali che ha un 1 in posizione i (da sinistra), per $0 \leqslant i < n$, e ha un 3 in posizione $n+j$ (da sinistra), per $0 \leqslant j < m$. Anche in questo caso, è chiaro che la costruzione dell'istanza di SUBSET SUM può essere eseguita in tempo polinomiale. Ad esempio, se la formula è costituita dalle clausole $\{x_1, x_2, x_3\}$,$\{\neg x_1, \neg x_2, x_3\}$,$\{\neg x_1, x_2, \neg x_3\}$, allora l'istanza di SUBSET SUM è costituita dai seguenti numeri.

Numero	1	2	3	1	2	3
p_1	1	0	0	1	0	0
n_1	1	0	0	0	1	1
p_2	0	1	0	1	0	1
n_2	0	1	0	0	1	0
p_3	0	0	1	1	1	0
n_3	0	0	1	0	0	1
x_1	0	0	0	1	0	0
y_1	0	0	0	1	0	0
x_2	0	0	0	0	1	0
y_2	0	0	0	0	1	0
x_3	0	0	0	0	0	1
y_3	0	0	0	0	0	1
s	1	1	1	3	3	3

Non è difficile dimostrare che la formula booleana è soddisfacibile se e solo se esiste un sottoinsieme dei numeri di tipo p, n, x e y la cui somma è uguale a s (si veda l'Esercizio 10.2). Quindi, il linguaggio 3-SAT è riducibile in tempo polinomiale al linguaggio SUBSET SUM: pertanto, quest'ultimo è NP-completo. A questo punto, facendo uso della tecnica di riduzione **per similitudine**, è facile dimostrare la NP-completezza di PARTITION (che in effetti è un problema simile a SUBSET SUM). In particolare, dato A e s, sia S la somma degli elementi di A: definiamo $A' = A \cup \{a_{n+1} = 3S - s, a_{n+2} = 2S + s\}$. Se esiste $B \subseteq A$ la somma dei cui elementi è uguale a s, allora definiamo $A'_1 = B \cup \{a_{n+1}\}$ (che ha somma $s + (3S - s) = 3S$) e $A'_2 = A - B \cup \{a_{n+2}\}$ (che ha somma $(S - s) + (2S + s) = 3S$): quindi, A'_1 e A'_2 è una soluzione per PARTITION. Viceversa, se esistono due sottoinsiemi A'_1 e A'_2 di A' di uguale somma, allora la somma dei loro elementi è uguale a $\frac{S+(3S-s)+(2S+s)}{2} = 3S$. Chiaramente, a_{n+1} deve appartenere a A'_1 oppure a A'_2: supponiamo che appartenga ad A'_1. Quindi, $A'_1 - \{a_{n+1}\}$ è un sottoinsieme di A la somma dei cui elementi è s. In conclusione,

abbiamo dimostrato che il linguaggio SUBSET SUM è riducibile in tempo polinomiale al linguaggio PARTITION: quindi, quest'ultimo è NP-completo.

Per concludere questa breve rassegna di tecniche di riduzione, consideriamo il problema KNAPSACK, che consiste nel decidere se, dati due insiemi di n numeri interi $V = \{v_1, \ldots, v_n\}$ e $P = \{p_1, \ldots, p_n\}$ e dati due numeri interi l e u, esiste un sottoinsieme J di $\{1, \ldots, n\}$ tale che

$$\sum_{j \in J} v_j \geqslant l \qquad \text{e} \qquad \sum_{j \in J} p_j \leqslant u$$

Possiamo mostrare che tale problema è NP-completo mediante la tecnica di riduzione **per restrizione**. In effetti, è facile verificare che PARTITION non è altro che un caso particolare di KNAPSACK, in cui, per ogni i con $1 \leqslant i \leqslant n$, $v_i = p_i$ e in cui $l = u = \frac{\sum_{i=1}^{n} v_i}{2}$.

10.3 Oltre NP

In quest'ultimo paragrafo, introdurremo due classi di complessità che includono la classe NP e che molto probabilmente sono strettamente più grandi di quest'ultima. La prima classe è la classe EXP, che include tutti i linguaggi decidibili in tempo esponenziale: mostreremo, mediante diagonalizzazione, come questa classe sia strettamente più grande della classe P. La seconda classe è la classe PSPACE, che include tutti i linguaggi decidibili usando un numero polinomiale di celle di memoria: in tal caso, mostreremo come, a differenza della complessità temporale, il non determinismo non aumenti il potere computazionale di macchine di Turing che operano con spazio polinomialmente limitato.

10.3.1 La classe EXP

Le classi di complessità P e NP ci hanno consentito di eseguire una prima approssimativa distinzione tra i linguaggi computazionalmente "trattabili" e quelli "non trattabili". Come il titolo di questo paragrafo suggerisce, introduciamo ora nuove classi di linguaggi che non sembrano appartenere alla classe NP: in effetti, anche se la maggior parte dei problemi combinatoriali che sorgono nella pratica rientrano nella classe NP, esiste una varietà di interessanti tipi di problemi che non sembrano essere inclusi in questa classe, giustificando in tal modo la definizione di classi di complessità aggiuntive.

In questo paragrafo, in particolare, introduciamo una classe di complessità temporale basata su funzioni esponenziali: in effetti, questa nuova classe è la più grande comunemente utilizzata nel classificare la complessità di problemi computazionali,

in quanto la maggior parte di questi ha una complessità temporale meno che esponenziale e, solo nel caso di pochi problemi interessanti, è noto che non è possibile avere una complessità temporale sub-esponenziale.

Da un punto di vista leggermente diverso, possiamo dire che la classe di complessità temporale esponenziale è la classe più ricca investigata nell'ambito della teoria della complessità computazionale, mentre quelle di complessità temporale più che esponenziale sono principalmente investigate nell'ambito della teoria della calcolabilità.

Definizione 10.3: la classe EXP

La classe EXP è l'insieme dei linguaggi L per i quali esiste una macchina di Turing T con un solo nastro che decide L e per cui $t_T(n) \in O(2^{n^k})$ per qualche $k \geqslant 1$.

Chiaramente, la classe P è contenuta nella classe EXP, in quanto ogni polinomio è minore di 2^n per n sufficientemente grande. Il prossimo risultato mostra che questa affermazione vale anche per la classe NP.

Teorema 10.3

La classe NP è contenuta nella classe EXP.

Dimostrazione. Se un linguaggio L appartiene alla classe NP, allora esiste un polinomio n^k e una macchina di Turing V che opera in tempo polinomiale tali che, per ogni stringa binaria x di lunghezza n, $x \in L$ se e solo se esiste una stringa binaria y di lunghezza n^k per cui $V(x,y)$ termina in uno stato finale. Definiamo, allora, una macchina di Turing T che, con input una stringa x di lunghezza n, genera una dopo l'altra tutte le stringhe di lunghezza n^k e, per ciascuna di esse, simula $V(x,y)$: non appena una di tali simulazioni termina in uno stato finale, T termina nel suo unico stato finale, mentre se nessuna di tali simulazioni termina in uno stato finale, allora T termina in uno stato non finale. Chiaramente, T decide L. Inoltre, poiché il numero di stringhe binarie di lunghezza n^k è pari a 2^{n^k} e poiché V opera in tempo polinomiale, è facile verificare che $t_T(n) \in O(2^{n^h})$ per qualche $h \geqslant k$. Quindi, L appartiene alla classe EXP e il teorema risulta essere dimostrato. ◇

Il prossimo risultato è una delle poche certezze della teoria della complessità computazionale e afferma che le macchine di Turing con complessità temporale esponenziale sono strettamente più potenti di quelle con complessità temporale polinomiale.

Teorema 10.4

Esiste un linguaggio L che appartiene alla classe EXP ma non appartiene alla classe P.

Dimostrazione. La dimostrazione si basa sull'utilizzo della tecnica di diagonalizzazione ed è simile a quella con cui abbiamo mostrato che il linguaggio L_{stop} non è decidibile (si veda il Teorema 3.5). Consideriamo, in particolare, il seguente linguaggio.

$$L_{acc-exp} = \{\langle C_T, x\rangle : T(x) \text{ termina in uno stato finale dopo al più } 2^{2|x|} \text{ passi}\}$$

$L_{acc-exp}$ appartiene alla classe EXP. In effetti, una macchina che decida tale linguaggio deve semplicemente inizializzare un contatore con il valore $2^{2|x|}$, simulare $T(x)$ e, per ogni passo della simulazione, diminuire il contatore di un'unità: se la simulazione termina in uno stato finale prima che il contatore divenga zero, allora la macchina termina in uno stato finale, altrimenti la macchina termina in uno stato non finale. È facile verificare che una macchina siffatta opera in tempo $O(2^{n^k})$ per qualche $k \geqslant 1$: quindi, $L_{acc-exp} \in$ EXP. Supponiamo, per assurdo, che $L_{acc-exp}$ sia decidibile da una macchina di Turing $T_{acc-exp}$ che opera in tempo 2^n. Pertanto, per ogni coppia di stringhe C_T e x, $T_{acc-exp}(C_T, x)$ termina in uno stato finale se $T(x)$ termina in uno stato finale entro $2^{2|x|}$ passi, altrimenti termina in uno stato non finale. Definiamo, ora, una nuova macchina di Turing $T_{diag-exp}$ che usa $T_{acc-exp}$ come sotto-macchina nel modo seguente: per ogni stringa C_T, $T_{diag-exp}(C_T)$ termina in uno stato finale se $T_{acc-exp}(C_T, C_T)$ termina in uno stato non finale, altrimenti termina in uno stato non finale. Osserviamo che il tempo richiesto da $T_{diag-exp}$ con input C_T è uguale al tempo richiesto da $T_{acc-exp}$ con input due copie di C_T: poiché $T_{acc-exp}$ opera in tempo 2^n, abbiamo che $T_{diag-exp}$ opera in tempo 2^{2n}. Consideriamo il comportamento di $T_{diag-exp}$ con input la sua codifica, ovvero $C_{T_{diag-exp}}$. Dalla definizione di $T_{acc-exp}$ e di $T_{diag-exp}$, abbiamo che se $T_{diag-exp}(C_{T_{diag-exp}})$ termina in uno stadio finale allora $T_{acc-exp}(C_{T_{diag-exp}}, C_{T_{diag-exp}})$ termina in uno stato non finale, ovvero $T_{diag-exp}(C_{T_{diag-exp}})$ non termina in uno stato finale entro $2^{2|C_{T_{diag-exp}}|}$ passi: poiché $T_{diag-exp}$ opera in tempo 2^{2n}, questo implica che $T_{diag-exp}(C_{T_{diag-exp}})$ deve terminare in uno stato non finale, generando una contraddizione. In modo analogo, possiamo mostrare che se $T_{diag-exp}(C_{T_{diag-exp}})$ termina in uno stato non finale, allora $T_{diag-exp}(C_{T_{diag-exp}})$ termina in uno stato finale e che, quindi, anche in questo caso si ha una contraddizione. Pertanto, abbiamo dimostrato che non può esistere la macchina di Turing $T_{acc-exp}$ e che $L_{acc-exp}$ non è decidibile in tempo 2^n. Poiché abbiamo già osservato che ogni linguaggio in P è decidibile in tempo 2^n, abbiamo quindi dimostrato che $L_{acc-exp}$ non appartiene alla classe P. Il teorema risulta così essere dimostrato. ◇

Concludiamo questo paragrafo osservando che non è invece noto se la classe EXP includa almeno un linguaggio non appartenente alla classe NP: in effetti, il rapporto tra NP e EXP è una delle tante questioni aperte che fanno compagnia a quella più famosa (ovvero, la congettura $P \neq NP$).

10.3.2 La classe PSPACE

Fino a ora abbiamo considerato solamente classi di complessità relative al tempo impiegato da una macchina di Turing per terminare la sua computazione. In questo paragrafo conclusivo, definiremo invece una classe di complessità basata sullo spazio utilizzato dalla computazione di una macchina di Turing e mostreremo i rapporti esistenti tra questa classe di complessità e le tre classi P, NP e EXP. Infine, dimostreremo un'interessante risultato relativo al rapporto esistente tra macchine di Turing deterministiche e non deterministiche limitate nello spazio a disposizione.

Data una macchina di Turing T con un singolo nastro che termina per ogni input x, la **complessità spaziale** di T è una funzione $s_T : \mathbb{N} \to \mathbb{N}$ tale che, per ogni $n \geqslant 1$, $s_T(n)$ indica il massimo numero di celle utilizzate da T con input una stringa di lunghezza n. È ovvio che la complessità spaziale di una macchina di Turing non può essere maggiore di quella temporale, in quanto nel caso peggiore a ogni passo la macchina usa una nuova cella. Il prossimo risultato, invece, mostra una relazione inversa tra la complessità temporale e quella spaziale di una macchina di Turing.

Teorema 10.5

Sia T una macchina di Turing con complessità spaziale s_T che termina per ogni input. Allora $t_T(n) \in O(2^{k s_T(n)})$, per qualche costante $k > 0$.

Dimostrazione. Sia $f_T(n) = |\Sigma|^{s_T(n)} s_T(n) |Q|$, dove Σ e Q indicano, rispettivamente, l'alfabeto di lavoro e l'insieme degli stati di T. Per ognuna delle $|\Sigma|^{s_T(n)}$ stringhe sull'alfabeto Σ di lunghezza $s_T(n)$, la testina può essere posizionata su una qualunque delle $s_T(n)$ celle e lo stato di T può essere uno qualunque degli stati in Q: quindi, $f_T(n)$ è un limite superiore al numero di possibili configurazioni distinte di T con input x di lunghezza n e, dopo $f_T(n)$ passi, $T(x)$ deve aver l'esecuzione. Quindi, $t_T(n) \leqslant f_T(n) \in O(2^{k s_T(n)})$ e il teorema risulta essere dimostrato. ◇

Definizione 10.4: la classe PSPACE

La classe PSPACE è l'insieme dei linguaggi L per i quali esiste una macchina di Turing T con un solo nastro che decide L e per cui $s_T(n) \in O(n^k)$ per qualche $k \geqslant 1$.

Da quanto detto nel paragrafo immediatamente precedente al Teorema 10.5, segue che la classe P è inclusa nella classe PSPACE (anche se non sappiamo se tale inclusione è stretta). Inoltre, dal Teorema 10.5 segue che la classe PSPACE è inclusa nella classe EXP (anche se non sappiamo se tale inclusione è stretta). Infine, dalla dimostrazione del Teorema 10.3 e dal fatto che lo spazio (a differenza del tempo) è riutilizzabile, segue anche che la classe NP è inclusa nella classe PSPACE (anche

se non sappiamo se tale inclusione è stretta). Il **teorema di Savitch**, invece, mostra come, nel caso di macchine di Turing con complessità spaziale polinomiale, il non determinismo non aumenti più di tanto il loro potere computazionale.

Teorema 10.6

Sia T una macchina di Turing non deterministica con un solo nastro che, per ogni cammino di computazione, opera in spazio polinomiale. Allora esiste una macchina di Turing deterministica con un solo nastro la cui complessità spaziale è polinomiale e che decide lo stesso linguaggio deciso da T.

Dimostrazione. Sia $s(n)$ il numero di celle usate da un qualunque cammino di computazione di T con input una stringa x di lunghezza n. Dal Teorema 10.5 segue che il numero di configurazioni di un qualunque cammino di computazione è limitato da $2^{cs(n)}$ dove c è una constante. Assumiamo, senza perdita di generalità, che, con input x, T abbia un'unica configurazione iniziale C_x e un'unica configurazione finale C^* e consideriamo il predicato $\texttt{reachable}(C, C', j)$ che è vero se la configurazione C' è raggiungibile a partire dalla configurazione C in al più 2^j passi. Da quanto detto in precedenza, abbiamo che T accetta la stringa x se e solo se $\texttt{reachable}(C_x, C^*, 2^{cs(|x|)})$ è vero. La dimostrazione consiste ora nello specificare un algoritmo deterministico in grado di calcolare il predicato `reachable` facendo uso di spazio polinomiale. A tale scopo, notiamo che invece di chiederci direttamente se è possibile raggiungere una configurazione C' a partire da una configurazione C in al più 2^j passi, possiamo usare la tecnica del *divide et impera* e dividere il problema in due sotto-problemi in qualche modo più facili, chiedendoci se esiste una configurazione C'' per cui è possibile raggiungere C'' a partire da C in al più 2^{j-1} passi ed è possibile raggiungere C' a partire da C'' in al più 2^{j-1} passi. Questo processo di scomposizione di un problema in sotto-problemi può essere iterato per valori sempre più piccoli di j fino a dividere il problema relativo a $j = 1$ in due problemi con $j = 0$: questi ultimi due problemi sono facilmente risolvibili in quanto è sufficiente verificare se la configurazione di partenza può produrre quella di arrivo. Per analizzare la complessità spaziale di tale algoritmo, osserviamo che il livello di annidamento delle chiamate ricorsive è $O(s(n))$ e che, ad ogni livello, è necessario una quantità di spazio $O(s(n))$ da usare per memorizzare i due parametri C e C' della chiamata ricorsiva. Pertanto, l'algoritmo ha complessità spaziale $O(s^2(n))$: poiché s è un polinomio, il teorema risulta essere dimostrato. ◇

Esercizi

Esercizio 10.1. Dimostrare che la classe NP coincide con l'insieme di tutti i linguaggi L per i quali esiste una macchina di Turing non deterministica che decide L in tempo polinomiale.

Esercizio 10.2. Facendo riferimento alla dimostrazione della NP-completezza di SUBSET SUM, dimostrare che la formula booleana di partenza è soddisfacibile se e solo se esiste un sottoinsieme dei numeri di tipo p, n, x e y la cui somma è uguale a s.

Esercizio 10.3. Il linguaggio VERTEX COVER consiste nel decidere se, dato un grafo non orientato G e un intero k, esiste un sottoinsieme C dei vertici di G tale che $|C| \leqslant k$ e, per ogni arco (u,v) di G, si abbia $\{u,v\} \cap C \neq \emptyset$ (ovvero, ogni arco è "coperto" da almeno un nodo di C). Dimostrare che il linguaggio VERTEX COVER è NP-completo, riducendo a esso 3-SAT mediante la tecnica di riduzione per progettazione di componenti.

Esercizio 10.4. Il linguaggio INDEPENDENT SET consiste nel decidere se, dato un grafo non orientato G e un intero k, esiste un sottoinsieme I dei vertici di G tale che $|I| \geqslant k$ e, per ogni arco (u,v) di G, si abbia $u \notin I \vee v \notin I$ (ovvero, i nodi di I sono indipendenti). Dimostrare che il linguaggio INDEPENDENT SET è NP-completo, riducendo a esso VERTEX COVER mediante la tecnica di riduzione per similitudine.

Esercizio 10.5. Il linguaggio CLIQUE consiste nel decidere se, dato un grafo non orientato G e un intero k, esiste un sottoinsieme C dei vertici di G tale che $|C| \geqslant k$ e, per ogni coppia di nodi $u, v \in C$, si abbia che (u,v) è un arco di G (ovvero, i nodi di C sono connessi a due a due). Dimostrare che il linguaggio CLIQUE è NP-completo, riducendo a esso INDEPENDENT SET mediante la tecnica di riduzione per similitudine.

Esercizio 10.6. Il linguaggio 3-DIMENSIONAL MATCHING consiste nel decidere se, dato un insieme $M \subseteq X \times Y \times Z$ dove X, Y e Z sono tre insiemi disgiunti di cardinalità pari a k, M ammette un accoppiamento tridimensionale perfetto, ovvero un sottoinsieme M' tale che ogni elemento di $X \cup Y \cup Z$ appare in una e una sola tripla di M'. Dimostrare che 3-DIMENSIONAL MATCHING è NP-completo, riducendo a esso 3-SAT mediante la tecnica di riduzione per progettazione di componenti.

Capitolo 11

Algoritmi di approssimazione

SOMMARIO

La maggior parte dei linguaggi NP*-completi apparsi in letteratura sono, in realtà, versioni decisionali di problemi di ottimizzazione, per i quali è richiesto non solo di decidere se esiste una soluzione, ma anche di calcolare eventualmente una soluzione ottima. In questo capitolo, vedremo che se la versione decisionale di un problema di ottimizzazione è* NP*-completa, allora il problema di ottimizzazione non può essere risolto in tempo polinomiale. Questo ci condurrà al concetto di algoritmo di approssimazione, ovvero di algoritmo che produca soluzioni non troppo lontane da una ottima. Mostreremo quindi come rispetto a tale nozione, i problemi di ottimizzazione possono avere comportamenti drasticamente diversi.*

11.1 Problemi di ottimizzazione

UN'IMPORTANTE caratteristica dei linguaggi è che consentono di modellare solamente problemi per i quali tutte le soluzioni sono considerate ugualmente accettabili e la cui risoluzione consiste semplicemente nel determinare se almeno una soluzione esiste. In molte applicazioni, tuttavia, quest'assunzione non è ragionevole ed è necessario operare un ordinamento delle soluzioni in base a qualche criterio. Ciò viene generalmente realizzato associando una misura a ogni soluzione: a seconda dell'applicazione, la soluzione migliore è quella con misura massima oppure con misura minima. Problemi di questo tipo sono detti problemi di ottimizzazione: alcuni di essi, come, ad esempio, problemi di schedulazione, di instradamento e di controllo del flusso, sono stati attentamente analizzati e in molti casi per essi sono state progettate euristiche che consentano di calcolare buone soluzioni (anche se non necessariamente ottime).

Oltre a questi sviluppi più sperimentali, due domande principali sorgono quando si ha a che fare con problemi di ottimizzazione: la prima consiste nel decidere se

esiste un algoritmo polinomiale che produca sempre una soluzione ottima, mentre la seconda, in caso ciò non sia possibile, consiste nel decidere se esistono algoritmi polinomiali in grado di calcolare soluzioni non ottime ma il cui grado di accuratezza possa essere tenuto sotto controllo. Per poter formalizzare tali domande introduciamo anzitutto il concetto di problema di ottimizzazione.

Definizione 11.1: Problemi di ottimizzazione

Un **problema di ottimizzazione** è una quadrupla $\langle I, S, m, \text{GOAL}\rangle$ tale che:

- I è un insieme di stringhe che codificano le **istanze** del problema;
- S è una funzione che associa a ogni istanza $x \in I$ un insieme finito e non vuoto di stringhe che codificano le **soluzioni** di x;
- m è una funzione che associa a ogni $x \in I$ e a ogni $y \in S(x)$ un numero intero positivo $m(x,y)$ che indica la **misura** della soluzione y rispetto all'istanza x;
- $\text{GOAL} = \max$ oppure $\text{GOAL} = \min$.

Risolvere un problema di ottimizzazione consiste nel trovare, data una stringa $x \in I$, una stringa $y \in S(x)$ tale che

$$m(x,y) = \text{OPT}(x) = \text{GOAL}\{m(x,z) : z \in S(x)\}.$$

I prossimi esempi di problemi di ottimizzazione sono collegati ad alcuni dei linguaggi NP-completi che abbiamo analizzato nel capitolo precedente. Come vedremo, tali esempi mostreranno come i problemi di ottimizzazione possono comportarsi in modo diverso relativamente alla loro soluzione approssimata.

Esempio 11.1: maximum Sat

Il problema MAXIMUM SAT consiste nel trovare, data una formula booleana in forma normale congiuntiva, un'assegnazione di verità che soddisfi il massimo numero di clausole. Quindi, nel caso di MAXIMUM SAT, abbiamo che:

- I è l'insieme delle stringhe che codificano formule booleane in forma normale congiuntiva;
- S associa a una formula booleana in forma normale congiuntiva l'insieme delle sue possibili assegnazioni di verità;
- per ogni formula booleana in forma normale congiuntiva x e per ogni sua possibile assegnazione di verità y, $m(x,y)$ è uguale al numero di clausole di x soddisfatte da y;
- $\text{GOAL} = \max$.

Esempio 11.2: problema del commesso viaggiatore

Il problema TRAVELING SALESPERSON consiste nel trovare, dato un grafo etichettato completo $G = (N, E)$ la cui funzione di etichetta associa a ogni arco un numero intero positivo, un ciclo di $|N|$ nodi distinti la somma dei cui archi sia minima (osserviamo che l'etichetta dell'arco (u, v) non necessariamente deve essere uguale all'etichetta dell'arco (v, u)). Quindi, nel caso di TRAVELING SALESPERSON, abbiamo che:

- I è l'insieme delle stringhe che codificano un grafo completo pesato;
- S associa a un grafo completo pesato l'insieme di tutte le possibili permutazioni dei suoi nodi;
- per ogni grafo completo pesato x e per ogni possibile permutazione dei suoi nodi y, $m(x, y)$ è uguale alla somma dei pesi degli archi inclusi nel ciclo specificato da x;
- GOAL $=$ min.

Esempio 11.3: minima partizione

Il problema MINIMUM PARTITION consiste nel trovare, dato un insieme A di numeri interi, una partizione di A in due sottoinsiemi A_1 e A_2 tale che sia minimo il valore massimo tra la somma degli elementi di A_1 e quella degli elementi di A_2. Quindi, nel caso di MINIMUM PARTITION, abbiamo che:

- I è l'insieme delle stringhe che codificano un insieme finito di numeri interi;
- S associa a un insieme finito di numeri interi l'insieme dei suoi sottoinsiemi;
- per ogni insieme finito x di numeri interi e per ogni suo sottoinsieme y, $m(x, y)$ è uguale al valore massimo tra la somma degli elementi di $x \cap y$ e quella degli elementi di $x - y$;
- GOAL $=$ min.

11.1.1 Linguaggi soggiacenti

Dato un problema di ottimizzazione, possiamo derivare, a partire da esso, un linguaggio, comunemente detto **linguaggio soggiacente**, nel modo seguente. Invece di cercare una soluzione ottima, ci domandiamo se esiste una soluzione la cui misura è almeno pari a k (nel caso in cui GOAL $=$ max) oppure è al più pari a k (nel caso in cui GOAL $=$ min): il linguaggio soggiacente sarà costituito da tutte le coppie $\langle x, k \rangle$ per le quali ciò sia vero. In altre parole, il linguaggio consiste di tutte le coppie $\langle x, k \rangle$ con $x \in I$ e $k > 0$, tali che $\text{OPT}(x) \geqslant k$ se GOAL $=$ max, $\text{OPT}(x) \leqslant k$ altrimenti.

Ad esempio, il linguaggio soggiacente di MAXIMUM SAT consiste di tutte le coppie $\langle x, k \rangle$ tali che x è una formula booleana in forma normale congiuntiva per la quale

esiste un'assegnazione di verità che soddisfa almeno k clausole. Il linguaggio soggiacente di TRAVELING SALESPERSON, invece, consiste di tutte le coppie $\langle x, k\rangle$ tali che x è un grafo completo pesato di n nodi per il quale esiste un ciclo di n nodi distinti la cui somma degli archi è minore oppure uguale a k. Infine, il linguaggio soggiacente MINIMUM PARTITION consiste di tutte le coppie $\langle x, k\rangle$ tali che x è un insieme di numeri interi che ammette un sottoinsieme y per cui il valore massimo tra la somma degli elementi di y e quella degli elementi non appartenenti a y sia minore oppure uguale a k.

Teorema 11.1

Sia Π un problema di ottimizzazione la cui funzione m sia calcolabile in tempo polinomiale. Se il linguaggio soggiacente di Π non appartiene alla classe P, allora Π non ammette un algoritmo di risoluzione che abbia complessità temporale polinomiale.

Dimostrazione. Supponiamo, per assurdo, che T sia una macchina di Turing con complessità temporale polinomiale che risolve Π, ovvero tale che, per ogni $x \in I$, $T(x)$ calcola una stringa $y \in S(x)$ per cui $m(x, y) = \text{OPT}(x)$. Allora, potremmo decidere il linguaggio soggiacente di Π operando nel modo seguente. Per ogni coppia $\langle x, k\rangle$, eseguiamo T con input x e verifichiamo se $m(x, y) \geqslant k$ nel caso in cui GOAL $=$ max, se $m(x, y) \leqslant k$ altrimenti. In base all'ipotesi fatta sulla funzione m, tale algoritmo ha una complessità temporale polinomiale, contraddicendo il fatto che il linguaggio soggiacente di Π non appartiene alla classe P. ◇

Osserviamo che, sotto determinate condizioni, è possibile dimostrare il viceversa del Teorema 11.1 (si veda l'Esercizio 11.1). Osserviamo, inoltre, che un'immediata conseguenza del risultato precedente è che se il linguaggio soggiacente di un problema di ottimizzazione è NP-completo, allora il problema di ottimizzazione stesso non ammette un algoritmo di risoluzione che abbia complessità temporale polinomiale (a meno che P $=$ NP).

Teorema 11.2

I linguaggi soggiacenti di MAXIMUM SAT, TRAVELING SALESPERSON e MINIMUM PARTITION sono NP-completi.

Dimostrazione. Dimostriamo che il linguaggio soggiacente di MAXIMUM SAT è NP-completo, riducendo a esso il problema SAT mediante la tecnica di restrizione. In effetti, SAT è un caso particolare del linguaggio soggiacente di MAXIMUM SAT, ristretto alle sole coppie $\langle x, k\rangle$ per cui k è esattamente uguale al numero delle clausole di x. In modo del tutto analogo, possiamo mostrare che il linguaggio soggiacente di MINIMUM PARTITION è NP-completo, riducendo a esso il problema PARTITION: infatti, PARTITION non è altro che un caso particolare del linguaggio soggiacente di

MINIMUM PARTITION, ristretto alle sole coppie $\langle x, k \rangle$ per cui k è esattamente uguale alla somma di tutti gli elementi di x divisa per due. Per dimostrare, invece, che il linguaggio soggiacente di TRAVELING SALESPERSON è NP-completo, introduciamo la seguente variante di HAMILTONIAN PATH. Il problema HAMILTONIAN CIRCUIT consiste nel decidere se un grafo di n nodi ammette un circuito di n nodi distinti: mostriamo che HAMILTONIAN CIRCUIT è NP-completo, riducendo a esso il problema NP-completo HAMILTONIAN PATH mediante la tecnica di similitudine. Dato un grafo G, creiamo un nuovo grafo G' ottenuto a partire da G aggiungendo due nodi s e t, aggiungendo un arco da s a ogni nodo di G, aggiungendo un arco da ogni nodo di G a t e aggiungendo un arco da t a s. È facile verificare che G ammette un cammino hamiltoniano se e solo se G' ammette un circuito hamiltoniano: quindi, HAMILTONIAN CIRCUIT è NP-completo. Possiamo ora dimostrare che il linguaggio soggiacente di TRAVELING SALESPERSON è NP-completo, riducendo a esso HAMILTONIAN CIRCUIT mediante la tecnica di similitudine. Dato un grafo G, costruiamo un nuovo grafo G' completo e pesato ottenuto a partire da G associando a ogni arco di G un peso pari a 1 e aggiungendo a G' tutti gli archi che non sono presenti in G assegnando a ciascuno di essi un peso pari a 2. Se esiste un circuito hamiltoniano in G, allora in G' esiste un circuito di n nodi distinti la somma dei cui archi è uguale a n. Viceversa, se non esiste un circuito hamiltoniano in G, allora ogni circuito di n nodi distinti in G' deve usare almeno un arco non presente in G e, quindi, la somma suoi archi deve essere almeno pari a $(n-1)+2 = n+1$. Quindi G ammette un circuito hamiltoniano se e solo se $\langle G, n \rangle$ appartiene al linguaggio soggiacente di TRAVELING SALESPERSON: pertanto, tale linguaggio è NP-completo. ◇

Corollario 11.1

I problemi di ottimizzazione MAXIMUM SAT, TRAVELING SALESPERSON e MINIMUM PARTITION non ammettono algoritmi di risoluzione con complessità temporale polinomiale, a meno che P = NP.

Dimostrazione. Il corollario segue immediatamente dai Teoremi 11.1 e 11.2 e dall'osservazione fatta tra i due teoremi. ◇

11.2 Algoritmi di approssimazione

A differenza dei linguaggi, nel caso dei problemi di ottimizzazione che non sono risolvibili in tempo polinomiale, ha senso pensare a algoritmi efficienti che producano soluzioni non ottime, ma non troppo "lontane" (rispetto alla funzione di misura) da una ottima. A tale scopo, introduciamo anzitutto il concetto di rapporto di prestazione, il cui scopo è quello di misurare la "bontà" di una soluzione (notiamo che in let-

teratura diverse nozioni di rapporto di prestazione sono state introdotte: quella da noi adottata ci consentirà di trattare in modo uniforme sia i problemi di massimizzazione, per i quali GOAL = max, che quelli di minimizzazione, per i quali GOAL = min).

Definizione 11.2: rapporto di prestazione

Sia Π un problema di ottimizzazione. Per ogni istanza $x \in I$ e per ogni soluzione $y \in S(x)$, il **rapporto di prestazione** di y rispetto a x è pari a

$$R(x,y) = \max\left\{\frac{OPT(x)}{m(x,y)}, \frac{m(x,y)}{OPT(x)}\right\}$$

Osserviamo che, dalla precedente definizione, segue immediatamente che il rapporto di prestazione è una quantità sempre maggiore oppure uguale a 1: più il rapporto di prestazione è vicino a 1, migliore è la soluzione y.

Definizione 11.3: algoritmo di approssimazione

Sia Π un problema di ottimizzazione e sia r una costante maggiore oppure uguale a 1. Un **algoritmo di r-approssimazione** per Π è una macchina di Turing con complessità temporale polinomiale che, per ogni istanza $x \in I$, produce una stringa $y \in S(x)$ tale che $R(x,y) \leqslant r$.

Come già detto, i tre problemi MAXIMUM SAT, TRAVELING SALESPERSON e MINIMUM PARTITION rappresentano tre tipologie diverse di problemi di ottimizzazione rispetto al fatto di ammettere o meno un algoritmo di approssimazione. Il prossimo risultato mostra come MAXIMUM SAT sia effettivamente un problema approssimabile.

Teorema 11.3

Esiste un algoritmo di 2-approssimazione per MAXIMUM SAT.

Dimostrazione. Data una formula booleana x in forma normale congiuntiva, consideriamo un algoritmo per la risoluzione di MAXIMUM SAT che opera nel modo seguente.

1. Seleziona il letterale che appare nel maggior numero di clausole. Sia C_l l'insieme di clausole che contengono l e sia $C_{\neg l}$ l'insieme di clausole che contengono $\neg l$.

2. Assegna a l il valore `true`, cancella le clausole in C_l e, per ogni clausola in $C_{\neg l}$, elimina da essa l'occorrenza di $\neg l$ (se la clausola diviene vuota, allora la cancella).

3. Se vi sono ancora variabili a cui non è stato assegnato un valore, allora torna al passo 1. Altrimenti termina.

Chiaramente, questo algoritmo ha complessità temporale polinomiale. È anche facile definire un insieme infinito di formule booleane in forma normale congiuntiva per le quali l'algoritmo non calcola la soluzione ottima (si veda l'Esercizio 11.2). Dimostriamo ora che l'algoritmo sicuramente soddisfa almeno la metà di tutte le clausole di x: poiché una soluzione ottima, al massimo, può soddisfare tutte le clausole di x, ciò implica che questo algoritmo è un algoritmo di 2-approssimazione per MAXIMUM SAT. La dimostrazione procede per induzione sul numero n di variabili che appaiono in x. Se $n = 1$, è immediato verificare che l'algoritmo soddisfa almeno la metà delle clausole di x esaminando tutti i possibili casi. Supponiamo, quindi, che, per ogni formula booleana in forma normale congiuntiva con al più n variabili, l'algoritmo soddisfa almeno la metà delle clausole della formula e sia x una formula con $n+1$ variabili. Sia c il numero delle clausole di x e indichiamo con c_l e $c_{\neg l}$ la cardinalità degli insiemi C_l e $C_{\neg l}$ definiti la prima volta che viene eseguito il passo 1 dell'algoritmo: per definizione di l, abbiamo che $c_l \geqslant c_{\neg l}$. Dopo aver eseguito il passo 2, il numero di clausole ancora presenti nella formula sarà almeno pari a $c - c_l - c_{\neg l}$: inoltre, in esse appaiono al più n variabili. Per ipotesi induttiva, l'algoritmo soddisfa almeno $(c - c_l - c_{\neg l})/2$ clausole: quindi, il numero totale di clausole soddisfatte dall'algoritmo è maggiore oppure uguale a

$$c_l + \frac{c - c_l - c_{\neg l}}{2} = \frac{c + c_l - c_{\neg l}}{2} \geqslant \frac{c}{2}$$

dove la diseguaglianza segue dal fatto che $c_l \geqslant c_{\neg l}$. Il teorema risulta, dunque, essere dimostrato. ◇

Il problema TRAVELING SALESPERSON, invece, è un esempio importante di problema di ottimizzazione che non può essere r-approssimato, per nessuna costante $r \geqslant 1$ (a meno che P = NP): in altre parole, il prossimo risultato ci consente di affermare che TRAVELING SALESPERSON è, molto probabilmente, un problema più difficile di MAXIMUM SAT.

Teorema 11.4

Se esiste un algoritmo di r-approssimazione per TRAVELING SALESPERSON, per una una qualunque costante $r \geqslant 1$, allora P = NP.

Dimostrazione. Supponiamo che vi sia una costante $r \geqslant 1$ per la quale esista un algoritmo T di r-approssimazione per TRAVELING SALESPERSON. Mostriamo come ciò implichi che il problema HAMILTONIAN CIRCUIT apparterebbe a P: poiché HAMILTONIAN CIRCUIT è un linguaggio NP-completo, avremmo che le due classi P

e NP coinciderebbero. Data un'istanza di HAMILTONIAN CIRCUIT, ovvero un grafo $G = (N, E)$, sia $n = |N|$. Costruiamo un'istanza di TRAVELING SALESPERSON, in modo che, esaminando la soluzione calcolata da T su tale istanza, sia possibile in tempo polinomiale decidere se G ammette o meno un circuito hamiltoniano. L'istanza di TRAVELING SALESPERSON è costruita a partire da G associando a ogni arco di G un peso pari a 1 e aggiungendo a G' tutti gli archi che non sono presenti in G assegnando a ciascuno di essi un peso pari a $\lceil 1 + nr \rceil$. Se esiste un circuito hamiltoniano in G, allora in G' esiste un circuito di n nodi distinti la somma dei cui archi è uguale a n: poichè T è un algoritmo di r-approssimazione, ciò implica che la misura della soluzione calcolata da T deve essere minore oppure uguale a nr. Viceversa, se non esiste un circuito hamiltoniano in G, allora ogni circuito di n nodi distinti in G' (in particolare, quello calcolato da T) deve usare almeno un arco non presente in G e, quindi, la somma suoi archi deve essere almeno pari a $(n-1)+1+nr = n(r+1)$. Quindi G ammette un circuito hamiltoniano se e solo se la misura della soluzione y calcolata da T è minore oppure uguale a nr: possiamo quindi decidere se G ammette un circuito hamiltoniano semplicemente verificando se $m(x, y) \leqslant nr$, ovvero il problema HAMILTONIAN CIRCUIT appartiene alla classe P. ◇

11.3 Schemi di approssimazione

Per molte applicazioni, sorge la necessità di accostarsi a una soluzione ottima in modo più forte di quanto consenta un algoritmo di r-approssimazione: chiaramente, se il problema non è risolvibile in tempo polinomiale, dovremo sempre accontentarci di soluzioni approssimate, ma possiamo possiamo pretendere di sviluppare algoritmi di approssimazione sempre migliori che ci consentano di avvicinarci il più possibile a una soluzione ottima. Per ottenere algoritmi di r-approssimazione con prestazioni migliori, siamo anche disposti a pagare un prezzo in termini di complessità temporale, la quale potrebbe crescere al diminuire del rapporto di prestazione. Queste considerazioni ci conducono alla seguente definizione.

Definizione 11.4: schemi di approssimazione

Sia Π un problema di ottimizzazione. Uno **schema di approssimazione** per Π è una macchina di Turing T tale che, per ogni istanza $x \in I$ e per ogni valore r maggiore di 1, $T(x, r)$ produce una stringa $y \in S(x)$ per cui $R(x, y) \leqslant r$: inoltre, $T(x, r)$ opera con complessità temporale polinomiale in $|x|$.

Sebbene uno schema di approssimazione deve terminare dopo un numero di passi polinomiale nella lunghezza dell'istanza x, la sua complessità temporale può dipendere anche da $\frac{1}{r-1}$: migliore è l'approssimazione, maggiore è il tempo di calcolo

richiesto dalla computazione di T. In molti casi, possiamo effettivamente avvicinarci tanto quanto desideriamo a una soluzione ottima, ma al prezzo di un aumento drammatico della complessità temporale. Questo è quanto accade nel caso di MINIMUM PARTITION, come mostrato dal seguente risultato.

Teorema 11.5

Esiste uno schema di approssimazione per MINIMUM PARTITION.

Dimostrazione. Dato un insieme x di n numeri interi e dato un valore $r > 1$, consideriamo un algoritmo per la risoluzione di MINIMUM PARTITION che opera nel modo seguente.

1. Ordina i numeri in ordine non crescente: sia $x_1, \ldots, x_n$ la sequenza ordinata.
2. Risolve in modo ottimo l'istanza formata dai primi k numeri, dove $k = \lceil \frac{2-r}{r-1} \rceil$: siano y_1 e y_2 la partizione così ottenuta.
3. Inserisci, uno dopo l'altro, i rimanenti $n-k$ numeri nell'insieme y_1 oppure nell'insieme y_2 a seconda di quale dei due insiemi ha la somma dei suoi elementi più piccola, al momento dell'inserimento.

Dimostriamo anzitutto che la soluzione prodotta da tale algoritmo ha un rapporto di prestazione minore oppure uguale a r. Se $r \geqslant 2$, allora

$$0 \geqslant \left\lceil \frac{2-r}{r-1} \right\rceil \geqslant \frac{2-r}{r-1} > -1$$

per cui $k = 0$: in questo caso, quindi, l'algoritmo costruisce una soluzione inserendo, uno dopo l'altro, tutti gli n numeri in uno di due insiemi y_1 e y_2, inizialmente vuoti, a seconda di quale dei due insiemi ha la somma dei suoi elementi più piccola, al momento dell'inserimento. Poiché ogni soluzione ha una misura almeno pari alla metà di $X = \sum_{i=1}^{n} x_i$ e non superiore a X, allora ogni soluzione (in particolare, quella calcolata dall'algoritmo) ha un rapporto di prestazione non superiore a 2. Assumiamo ora che $r < 2$ e indichiamo con Y_i la somma degli elementi inclusi in y_i, per $i = 1, 2$, al termine dell'esecuzione dell'algoritmo: senza perdita di generalità possiamo supporre che $Y_1 \geqslant Y_2$ (in quanto l'altro caso può essere analizzato in modo del tutto simile). Sia x_h l'ultimo elemento inserito in y_1 dall'algoritmo: quindi, $Y_1 - x_h \leqslant Y_2$. Sommando Y_1 a entrambi i membri di questa diseguaglianza e dividendo per 2, abbiamo che

$$Y_1 \leqslant \frac{X + x_h}{2}$$

Se x_h è stato inserito in y_1 durante il passo 2 dell'algoritmo, allora il passo 3 non può che aver diminuito la differenza tra Y_1 e Y_2: poiché la soluzione calcolata al passo 1

è ottima, allora anche la soluzione prodotta al termine dell'algoritmo è ottima. Se, invece, x_h è stato inserito in y_1 durante il passo 3 dell'algoritmo, allora abbiamo che, per ogni i con $1 \leqslant i \leqslant k$, $x_i \geqslant x_h$ (in quanto gli elementi sono stati ordinati al passo 1 in ordine non crescente) e, quindi, che

$$X \geqslant x_h + \sum_{i=1}^{n} x_i \geqslant x_h + \sum_{i=1}^{n} x_h = x_h(k+1)$$

Poiché $Y_1 \geqslant \frac{X}{2} \geqslant Y_2$ e poiché $\text{OPT}(x) \geqslant \frac{X}{2}$, abbiamo che il rapporto di prestazione della soluzione calcolata dall'algoritmo è uguale a

$$\frac{Y_1}{\text{OPT}(x)} \leqslant \frac{2Y_1}{X} \leqslant \frac{X+x_h}{X} = 1+\frac{x_h}{X} \leqslant 1+\frac{x_h}{x_h(k+1)} = 1+\frac{1}{k+1} \leqslant 1+\frac{1}{\frac{2-r}{r-1}+1} = r$$

Rimane, ora, da dimostrare che l'algoritmo opera in tempo polinomiale nella lunghezza di x: in particolare, mostriamo che la complessità temporale dell'algoritmo è $O(n\log n + \frac{1}{r-1}2^{\frac{1}{r-1}})$. Chiaramente, il passo 1 richiede tempo $O(n\log n)$, facendo uso di uno qualunque degli algoritmi ottimi di ordinamento, mentre il passo 3 richiede tempo $O(n)$. Il passo 2 può essere realizzato esaminando tutti i possibili sottoinsiemi dei primi k elementi di x e selezionando quello che minimizza il massimo tra la somma dei suoi elementi e quella degli altri elementi: tale operazione richiede tempo $O(k2^k) = O(\frac{1}{r-1}2^{\frac{1}{r-1}})$, in quanto $k \leqslant \frac{2-r}{r-1}+1 = \frac{1}{r-1}$. Il teorema risulta, quindi, essere dimostrato. ◇

11.3.1 Algoritmi di approssimazione e schemi di approssimazione

Concludiamo questa breve trattazione della teoria degli algoritmi di approssimazione mostrando come esistano problemi di ottimizzazione che ammettono un algoritmo di r-approssimazione, per una fissata costante $r \geqslant 1$, ma che non ammettono uno schema di approssimazione.

Esempio 11.4: minimum bin packing

Il problema MINIMUM BIN PACKING consiste nel trovare, dato un insieme A di numeri interi e dato un numero intero b, una partizione di A in k sottoinsiemi $A_1, A_2, \ldots, A_k$ (detti *bin*) tale che la somma degli elementi di A_i non sia superiore a b e tale che k sia il minimo possibile. Informalmente, il problema consiste, quindi, nell'impacchettare un insieme di oggetti facendo uso del minor numero possibile di scatole della stessa dimensione.

Mostriamo anzitutto che il linguaggio soggiacente di MINIMUM BIN PACKING è NP-completo, riducendo a esso PARTITION mediante la tecnica di riduzione per

restrizione. In effetti, PARTITION non è altro che un caso particolare del linguaggio soggiacente di MINIMUM BIN PACKING, ristretto alle sole coppie $\langle x, k \rangle$ per cui b è uguale alla metà della somma di tutti gli elementi di A e k è uguale a 2. Pertanto, MINIMUM BIN PACKING non ammette una algoritmo di risoluzione polinomiale, a meno che P = NP.

Il prossimo risultato mostra, invece, che MINIMUM BIN PACKING può essere risolto con un rapporto di prestazione limitato da una costante.

Teorema 11.6

Esiste un algoritmo di 2-approssimazione per MINIMUM BIN PACKING.

Dimostrazione. Data un insieme A di numeri interi $a_1, a_2, \ldots, a_n$ e dato un numero intero b, consideriamo un algoritmo per la risoluzione di MINIMUM BIN PACKING che opera nel modo seguente (nel seguito, assumiamo senza perdità di generalità, che $a_i \leqslant b$ per ogni i con $1 \leqslant i \leqslant n$).

1. Inserisce il primo numero a_1 nel primo bin A_1.
2. Dovendo inserire il numero a_i, con $2 \leqslant i \leqslant n$, sia A_j il bin in cui è stato inserito a_{i-1}: se a_i può entrare in A_j senza superare il limite b, allora inserisce a_i in A_j. Altrimenti, crea un nuovo bin A_{j+1} e inserisce a_i in A_{j+1}.
3. Ripete il passo 2 fino a quando tutti i numeri sono stati inseriti in un bin.

Chiaramente, questo algoritmo ha complessità temporale polinomiale. È anche facile definire un insieme infinito di istanze per le quali l'algoritmo non calcola la soluzione ottima (si veda l'Esercizio 11.3). Dimostriamo ora che la soluzione calcolata dall'algoritmo ha un rapporto di prestazione non superiore a 2. A tale scopo, osserviamo che, per ogni coppia di bin consecutivi, la somma degli elementi inclusi in questi due bin deve essere maggiore di b (altrimenti, l'algoritmo avrebbe inserito tutti questi elementi nel primo dei due bin). Quindi, il numero di bin usati dalla soluzione calcolata dall'algoritmo non può essere maggiore di $2\frac{S}{b}$, dove S indica la somma di tutti i numeri in A. D'altra parte, ogni soluzione (inclusa quella ottima) deve usare almeno $\frac{S}{b}$ bin: pertanto, il rapporto tra la misura della soluzione calcolata dall'algoritmo e la misura ottima è minore oppure uguale a 2. ◇

A questo punto è naturale chiedersi se MINIMUM BIN PACKING ammette uno schema di approssimazione, ovvero se è possibile calcolare una soluzione arbitrariamente vicina a una ottima (al prezzo di una complessità temporale più elevata). L'ultimo risultato di questo capitolo mostra che la risposta a tale domanda è, molto probabilmente, negativa: in effetti, se un tale schema di approssimazione esistesse, allora le classi P e NP coinciderebbero.

Teorema 11.7

Se esiste uno schema di approssimazione per MINIMUM BIN PACKING, allora P = NP.

Dimostrazione. Mostriamo che se esiste un algoritmo T di r-approssimazione per MINIMUM BIN PACKING, per una qualunque costante $r < \frac{3}{2}$, allora PARTITION appartiene alla classe P: poiché PARTITION è un linguaggio NP-completo, questo implica che P = NP. Dato un insieme di numeri interi A, consideriamo l'istanza x di MINIMUM BIN PACKING costituita da A e da b pari alla metà della somma di tutti gli elementi in A. Se A ammette una partizione perfetta, allora $\text{OPT}(x) = 2$: poiché T è un algoritmo di r-approssimazione con $r < \frac{3}{2}$, abbiamo che la misura della soluzione calcolata da T deve essere minore di 3, ovvero deve essere uguale a 2. Se, al contrario, A non ammette una partizione perfetta, allora ogni soluzione di x (e, in particolare, quella calcolata da T) deve avere una misura non inferiore a 3. Quindi, possiamo decidere se A ammette una partizione perfetta semplicemente verificando se la misura della soluzione calcolata da T è minore di 3: poichè T ha una complessità temporale polinomiale (essendo r fissata), ciò implica che PARTITION appartiene alla classe P. Il teorema risulta dunque essere dimostrato. ◇

In conclusione, abbiamo mostrato che, relativamente all'esistenza di algoritmi di approssimazione, i problemi di ottimizzazione possono avere comportamenti ben diversi tra di loro. In particolare, assumendo che P ≠ NP, valgono le seguenti affermazioni.

- Esistono problemi che non ammettono alcun algoritmo di r-approssimazione, per nessuna costante $r \geqslant 1$ (ad esempio, TRAVELING SALESPERSON).
- Esistono problemi che ammettono un algoritmo di r-approssimazione, per una fissata costante r, ma che non ammettono schemi di approssimazione (ad esempio, MINIMUM BIN PACKING e, come è possibile mostrare con strumenti particolarmente sofisticati, MAXIMUM SAT).
- Esistono problemi di ottimizzazione che ammettono schemi di approssimazione (ad esempio, MINIMUM PARTITION).

Migliaia di problemi di ottimizzazione sono stati classificati sulla base delle loro proprietà di approssimabilità e, per molti di essi, la principale questione ancora da risolvere consiste nel determinare fino a che punto possano essere approssimati, ovvero nel determinare esattamente la soglia del rapporto di prestazione ottenibile in tempo polinomiale. Fino all'inizio degli anni novanta, questa questione sembrava ancora molto difficile da risolvere e pochi erano i risultati che dimostravano limiti inferiori sul rapporto di prestazione ottenibile per un determinato problema di ottimizzazione. In seguito a uno dei teoremi forse più interessanti della teoria della complessità

computazionale, ovvero il teorema PCP, combinato con tecniche di riduzione simili a quelle viste nel capitolo precedente, è stato, invece, possibile incominciare a ottenere risultati di questo tipo (anche se, per la maggior parte dei problemi di ottimizzazione, la determinazione dell'esatta soglia di approssimabilità risulta, tuttora, sconosciuta).

Esercizi

Esercizio 11.1. Sia Π un problema di ottimizzazione per il quale (1) il linguaggio I appartiene alla classe P, (2) esiste un polinomio p tale che, per ogni $x \in I$, se $y \in S(x)$, allora $y \leqslant p(|x|)$, (3) il linguaggio $\{\langle x, y\rangle : y \leqslant p(|x|) \wedge y \in S(x)\}$ appartiene alla classe P e (4) la funzione m è calcolabile in tempo polinomiale. Facendo uso della tecnica di ricerca binaria, dimostrare che se Π non ammette un algoritmo di risoluzione che abbia complessità temporale polinomiale, allora il linguaggio soggiacente di Π non appartiene alla classe P.

Esercizio 11.2. Definire un insieme infinito di formule booleane in forma normale congiuntiva per le quali l'algoritmo di 2-approssimazione per MAXIMUM SAT non calcola una soluzione ottima.

Esercizio 11.3. Definire un insieme infinito di istanze di MINIMUM BIN PACKING per le quali l'algoritmo di 2-approssimazione non calcola una soluzione ottima.

Esercizio 11.4. Il problema MINIMUM VERTEX COVER consiste nel trovare, dato un grafo non orientato G, un sottoinsieme C dei suoi vertici di cardinalità minima tale che, per ogni arco (u, v) di G, si abbia $\{u, v\} \cap C \neq \emptyset$ (ovvero, ogni arco è "coperto" da almeno un nodo di C). Dimostrare che il linguaggio soggiacente di MINIMUM VERTEX COVER è NP-completo, riducendo a esso 3-SAT mediante la tecnica di riduzione per progettazione di componenti.

Esercizio 11.5. Si consideri il seguente algoritmo per la risoluzione del problema dell'esercizio precedente, ovvero MINIMUM VERTEX COVER.

1. Marca tutti gli archi di G come "non coperti" e pone $C = \emptyset$.
2. Sia (u, v) un arco non coperto. Se $\{u, v\} \cap C = \emptyset$, allora includi u e v in C. In ogni caso, marca (u, v) come "coperto".
3. Ripete il passo precedente fino a quando non vi sono più archi non coperti.

Dimostrare che tale algoritmo produce sempre una soluzione il cui rapporto di prestazione non è maggiore di 2.