Dispense del corso di Fondamenti di Informatica I

Luca Forlizzi, Giovanna Melideo, Enrico Nardelli, Guido Proietti

Premessa

Attraverso questa dispensa abbiamo cercato di fornire allo studente uno strumento didattico che lo aiutasse ad orientarsi tra i contenuti del corso di Fondamenti di Informatica I. Sebbene frutto dell'esperienza didattica maturata nei corsi di Fondamenti di Informatica: Calcolabilità e Complessità e Fondamenti di Informatica I tenuti all'Università dell'Aquila dal 1999 al 2003, la quantità di argomenti trattati e l'esigenza di rendere fruibile la materia a studenti con limitate conoscenze di matematica elementare, ci hanno sicuramente indotto ha commettere molti errori. Preghiamo gli studenti di perdonarceli, e di aiutarci a migliorare questo lavoro segnalandoceli. Lo studente è inoltre invitato, per approfondire o chiarire gli argomenti trattati, a consultare altri testi di Logica, Teoria della Calcolabilità e Teoria della complessità. A tale scopo, suggeriamo in bibliografia alcuni testi che riteniamo particolarmente interessanti e didatticamente utili.

Ringraziamenti

Il contributo portato da Caterina Mandolini alla scrittura di questa dispensa è stato fondamentale. Le sue idee, la sua prosa felice e il suo contagioso entusiasmo hanno elevato tanto la qualità del testo. Un ringraziamento particolare va a Domenico Cecchini che ci ha risolto tanti problemi tecnici con prontezza ed efficacia pari alla sua grande gentilezza. Ringraziamo anche Pasquale De Medio e Stefano Gentile che ci hanno offerto ulteriore supporto tecnico in più di un'occasione.

Indice

Capitolo 1

Logica

1.1 Logica proposizionale

1.1.1 Logica e linguaggio naturale

La logica nasce dall'opera che i filosofi hanno diretto verso lo studio dei meccanismi tipici del ragionamento, cioè della capacità di trarre conseguenze da un certo insieme di premesse. Fin dall'antichità si è osservato che certi schemi di ragionamento sono indipendenti dal senso espresso dalle singole parti del discorso.

Consideriamo il celebre sillogismo "ogni uomo è mortale, Socrate è un uomo, dunque Socrate è mortale". Questa proposizione è intuitivamente vera. Osserviamo che ciò non è diretta conseguenza della proprietà di essere mortale né dell'aver scelto un particolare individuo (in questo caso, Socrate). È immediato verificare che ciò accade per tutte le frasi che presentano la medesima struttura. Consideriamo, ad esempio, "tutti i cani hanno quattro zampe, Pluto è un cane, Pluto ha quattro zampe". In generale, se accade che una proprietà P è soddisfatta da tutti gli elementi di un determinato insieme, allora, preso un qualunque elemento a di quell'insieme, la proprietà P è necessariamente soddisfatta da a, indipendentemente da P.

La logica è un linguaggio che definisce formalmente la struttura delle sentenze e che, a differenza del linguaggio naturale, attribuisce loro un significato, che può essere un valore di verità o di falsità, indipendentemente dal contenuto informativo, basandosi solo sulla loro struttura, come accade per il sillogismo di Socrate. Le sentenze che esprimono un valore di verità o falsità vengono chiamate asserzioni. Nel seguito considereremo sinonimi i termini frase, sentenza, asserzione, proposizione, formula ben formata. Esempi di asserzioni sono: "4 è un numero dispari", "piove", "la mortadella è un vegetale". A tutte quante si può associare un valore di verità: falso per "4 è un numero dispari", vero o falso per "piove" a seconda delle condizioni metereologiche, vero o falso per "la mortadella è un vegetale" non appena si stabilisce in che accezione viene usato il termine vegetale (di origine vegetale oppure non animato).

Nel linguaggio naturale i connettivi sono le parti del discorso che combinano una o più sentenze per dare luogo ad una nuova sentenza. Ad esempio, la sentenza "ho sete quindi bevo" è composta dalle sentenze "ho sete" e "bevo" e dal connettivo *quindi*. La sentenza "ho sete, quindi bevo" esprime il nesso di causalità tra "ho sete" e "bevo" le quali prese

singolarmente hanno interpretazioni di verità non correlate. Entrambe possono essere vere oppure false, indipendentemente l'una dall'altra. Lo stesso grado di indipendenza hanno le sentenze "ho sete" e "mangio"; tuttavia la combinazione delle due in "ho sete, quindi mangio" esprime un nesso di causalità che non corrisponde al senso comune (avremmo probabilmente ritenuto più naturale associare le frasi "ho fame" e "mangio") e ci lascia un po' perplessi, nonostante il meccanismo di costruzione della frase sia lo stesso.

Nella logica le regole di interpretazione non fanno riferimento al significato delle frasi e non cedono spazio a tali perplessità: le sentenze vengono rappresentate tramite variabili proposizionali, cioè tramite simboli che denotano un qualunque asserto che abbia un valore di verità, vero o falso, e le regole in base alle quali i connettivi associano un valore di verità alle frasi composte non dipendono dal senso espresso dalla frase.

1.1.2 Connettivi della logica proposizionale

Prima di presentare la sintassi della logica proposizionale illustriamo le caratteristiche dei connettivi. Analogamente a quanto succede nel linguaggio naturale, i connettivi sono costruttori di frasi: prendono in pasto sentenze e restituiscono sentenze. Dal punto di vista della semantica i connettivi del linguaggio proposizionale consentono di determinare il valore delle frasi composte, a partire dal valore delle frasi componenti mediante una relazione di tipo funzionale. In altre parole, un connettivo con n argomenti è una funzione il cui dominio e codominio sono rispettivamente $\{0,1\}^n$ e $\{0,1\}$.

Specificheremo in seguito l'insieme dei connettivi della logica proposizionale. Limitiamoci adesso ad esaminare le classi delle funzioni da $\{0,1\}^n$ a $\{0,1\}$ con $n \in \{0,1,2\}$.

FUNZIONI NULLARIE

Sono le funzioni che non hanno argomenti e restituiscono costantemente un valore del codominio $\{0,1\}$. Esse sono:

n_0	n_1
0	1

Tabella 1.1: funzioni nullarie

FUNZIONI UNARIE

Le funzioni unarie hanno come argomento un elemento dell'insieme $\{0,1\}$ e come risultato un elemento dell'insieme $\{0,1\}$. Esistono quindi 4 diverse funzioni unarie riportate nella Tabella **??**.

FUNZIONI BINARIE

Hanno due argomenti, entrambi elementi di $\{0,1\}$, e il risultato è un valore di $\{0,1\}$. La

argomento	u_0	u_1	u_2	u_3
0	0	0	1	1
1	0	1	0	1

Tabella 1.2: funzioni unarie

tabella seguente mostra le funzioni binarie:

argomenti	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Tabella 1.3: funzioni binarie

1.1.3 Formule ben formate

Le *formule ben formate* (nel seguito fbf) sono le frasi sintatticamente corrette che appartengono al linguaggio della logica proposizionale.

Da un punto di vista formale una frase è una successione di simboli, detta stringa. Per stabilire se una frase è una fbf occorre conoscere dapprima qual è l'alfabeto dei simboli del linguaggio e poi quali sono le regole che stabiliscono come tali simboli possono succedersi nelle stringhe.

L'alfabeto del linguaggio della logica proposizionale è formato da:

- simboli atomici di proposizione: $A, B, \ldots$
- simboli di costante: $\top$, $\bot$
- connettivi: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
- simboli ausiliari: (e)

L'insieme FBF delle formule ben formate è definito come segue:

- $A, B, \ldots \in$ FBF per ogni simbolo atomico di proposizione
- $\top, \bot \in$ FBF
- se $P \in$ FBF, allora $(\neg P) \in$ FBF
- se $P, Q \in$ FBF allora $(P \wedge Q), (P \vee Q), (P \rightarrow Q), (P \leftrightarrow Q) \in$ FBF

Convenzionalmente, indicheremo con le prime lettere dell'alfabeto $(A, B, C, \ldots)$ i simboli atomici di proposizione e con le lettere $P, Q, R, \ldots$ le fbf.

ESERCIZIO 1. Le seguenti stringhe non sono formule ben formate. Perché?

- $((A \wedge B) \rightarrow \nabla)$
- $(\neg\bot \leftrightarrow A)$
- $((A) \vee (B))$
- $(A \wedge \rightarrow \bot) \vee B$
- $(((A \rightarrow B \wedge \top) \rightarrow \bot)$

REGOLE DI PRECEDENZA DEI CONNETTIVI

Introduciamo una regola che stabilisce una precedenza tra i connettivi. Utilizziamo $>$ per denotare che il connettivo a sinistra di tale simbolo ha precedenza di valutazione rispetto al connettivo che si trova a destra:

$$\neg \ > \ \wedge \ > \ \vee \ > \rightarrow \ > \leftrightarrow$$

In base a tale convenzione possiamo eliminare le parentesi in alcuni casi. Per esempio possiamo scrivere $A \wedge \neg B \vee C$ al posto di $((A \wedge \neg B) \vee C)$. Non è possibile, invece, eliminare le parentesi in $A \vee (B \rightarrow C)$ perché la fbf $A \vee B \rightarrow C$ sarebbe scorrettamente interpretata come $(A \vee B) \rightarrow C$.

ESERCIZIO 2. Eliminare le parentesi nelle seguenti fbf quando ciò risulta possibile.

- $(\bot \rightarrow C) \leftrightarrow (A \vee B)$
- $((A \wedge B) \wedge (C \vee D))$
- $(((A \wedge B) \wedge C) \leftrightarrow (B \rightarrow D))$
- $(\neg(\top \rightarrow \bot) \vee (A \wedge \top))$
- $(((\top \rightarrow A) \rightarrow B) \rightarrow \bot)$
- $(((A \leftrightarrow A) \wedge (B \rightarrow B)) \rightarrow ((B \wedge C) \vee D))$

1.1.4 Semantica dei connettivi

Diamo ora un significato ai simboli dei connettivi.

VERO E FALSO

Le costanti $\top$ (*vero, top*) e $\bot$ (*falso, bottom*) hanno rispettivamente valore 1 e 0. Essi corrispondono alle funzioni nullarie n_1 e n_0 della tabella 1.1.

NEGAZIONE

Il connettivo $\neg$ (*not, non*) corrisponde alla funzione u_2 della tabella 1.2. Quando P è falsa, $\neg P$ è vera e viceversa.

CONGIUNZIONE

Il connettivo $\wedge$ (*and, e*) corrisponde alla funzione b_1 della tabella 1.3. $P \wedge Q$ è vera quando P è vera e Q è vera; falsa altrimenti.

DISGIUNZIONE

Il connettivo $\vee$ (*or, o*) corrisponde alla funzione b_7 della tabella 1.3. $P \vee Q$ è vera quando P è vera oppure Q è vera oppure quando P e Q sono entrambe vere; falsa altrimenti.

Il connettivo $\vee$ viene detto anche *or inclusivo* per distinguerlo dal connettivo *or esclusivo* (detto anche xor o eor) che a differenza del primo restituisce 0 anche nel caso in cui P e Q sono entrambe vere. L'*or esclusivo* corrisponde alla funzione b_6 della tabella 1.3: esso tuttavia non fa parte del linguaggio proposizionale così come lo abbiamo definito.

IMPLICAZIONE

Il connettivo $\rightarrow$ (*...implica..., se...allora, ...solo se..., ...se...*) corrisponde alla funzione b_{13} della tabella 1.3. $P \rightarrow Q$ è falsa quando P è vera e Q è falsa; vera altrimenti.

La parte a sinistra del connettivo $\rightarrow$ è detta premessa o ipotesi; la parte a destra è detta conclusione o tesi. Nelle diciture "P implica Q" e "se P allora Q" la premessa è P e la conclusione è Q: ovvero entrambe vengono denotate, in simboli, dalla fbf $P \rightarrow Q$. Diverso è il caso del "se": infatti, in "P se Q", Q è l'ipotesi e P è la tesi: in simboli abbiamo $Q \rightarrow P$. Il "solo se", al contrario, rispetta il verso dell'implicazione: "P solo se Q" vuol dire $P \rightarrow Q$.

Consideriamo, ad esempio, le sentenze "vincerò alla lotteria solo se comprerò almeno un biglietto" e "vincerò alla lotteria se comprerò almeno un biglietto". Nel primo caso la vincita alla lotteria costituisce la premessa: la frase composta risulta vera perché, supposta vera la premessa, è immediato pervenire alla conclusione (se vinco alla lotteria vuol dire che ho comprato almeno un biglietto) e, per la regola dell'implicazione, se P è vera e Q è vera allora $P \rightarrow Q$ è vera. Nel secondo caso la vincita alla lotteria è la conclusione: la frase risulta palesemente falsa (altrimenti tutti gli acquirenti sarebbero vincitori).

Tornando alla semantica del connettivo, l'implicazione è falsa quando la premessa è vera e la conclusione è falsa. Risulta vera in tutti gli altri casi, in particolare quando la premessa è falsa. In tal caso, il nesso di causalità tra premessa e conclusione, propria del linguaggio naturale, viene a mancare completamente per cui da una premessa falsa si può concludere qualsiasi tesi. Ad esempio "se gli asini volano, allora il Papa è una donna" e "se gli asini volano, allora il Papa è un uomo" sono entrambe asserzioni vere.

EQUIVALENZA

Il connettivo $\leftrightarrow$ (*... equivale a...*, *... se e solo se...*) corrisponde alla funzione b_9 della tabella 1.3. $P \leftrightarrow Q$ è vera quando P e Q hanno lo stesso valore di verità, falsa altrimenti. La parte sinistra e la parte destra del connettivo $\leftrightarrow$ sono premessa e conclusione di due diverse implicazioni. In "P se e solo se Q" ($P \leftrightarrow Q$) distinguiamo una parte "se" e una parte "solo se": "P se Q" è denotata da $Q \rightarrow P$ e "P solo se Q" è denotata da $P \rightarrow Q$.

Una equivalenza risulta vera se sono vere entrambe le implicazioni, quella della parte "se" e quella della parte "solo se", oppure entrambe false. Ad esempio "farò tredici se e solo se azzeccherò tutti i risultati della schedina" è una frase vera. Infatti, supponiamo vera "farò tredici": questo vuol dire che il vincitore del montepremi "azzeccherà tutti i risultati della schedina". Viceversa, supponiamo che "azzeccherò tutti i risultati della schedina" risulti vera: allora "farò tredici" è anch'essa vera. Entrambe le frasi risultano vere; pertanto l'equivalenza è vera.

Consideriamo un altro esempio: "15 è un numero pari se e solo se 15 è un numero primo". È immediato verificare che tale frase risulta essere vera.

1.1.5 Interpretazioni delle fbf

Un'interpretazione v è una funzione che ad ogni fbf P associa un valore di verità $v(P)$. La funzione di interpretazione assegna un valore di verità ad ogni formula atomica che compare in P ed estende tali valori alla formula composta mediante la semantica dei connettivi.

In altri termini $v : \text{FBF} \mapsto \{0, 1\}$ è una interpretazione quando le seguenti condizioni sono verificate:

- se P è l'atomo $\bot$ allora $v(P) = 0$
- se P è l'atomo $\top$ allora $v(P) = 1$
- se P è della forma $\neg P_1$ allora $v(P) = 1 - v(P_1)$
- se P è della forma $P_1 \wedge P_2$ allora $v(P) = \min\{v(P_1), v(P_2)\}$
- se P è della forma $P_1 \vee P_2$ allora $v(P) = \max\{v(P_1), v(P_2)\}$
- se P è della forma $P_1 \rightarrow P_2$ allora

$$v(P) = \begin{cases} 1 & \text{se } v(P_1) \leq v(P_2) \\ 0 & \text{se } v(P_1) > v(P_2) \end{cases}$$

- se P è della forma $P_1 \leftrightarrow P_2$ allora

$$v(P) = \begin{cases} 1 & \text{se } v(P_1) = v(P_2) \\ 0 & \text{se } v(P_1) \neq v(P_2) \end{cases}$$

OSSERVAZIONE

Quello appena introdotto è un modo conciso di esprimere il comportamento dei connettivi senza ricorrere all'utilizzo delle tabelle di verità.

ESEMPIO

Sia P la fbf $A \wedge B \rightarrow C$. Nella seguente tabella vengono mostrate le 8 (cioè 2^3) possibili interpretazioni $v_0, v_1, \ldots, v_7$ di P ottenute al variare dei valori di verità associati alle formule atomiche A, B, C.

Interpretazioni	A	B	C	$A \wedge B$	$A \wedge B \rightarrow C$
v_0	0	0	0	0	1
v_1	0	0	1	0	1
v_2	0	1	0	0	1
v_3	0	1	1	0	1
v_4	1	0	0	0	1
v_5	1	0	1	0	1
v_6	1	1	0	1	0
v_7	1	1	1	1	1

ESERCIZI

- Sia P la formula $(A \vee B) \wedge \neg C$. Sia v un'interpretazione tale che $v(A) = 1, v(B) = 0, v(C) = 1$. Determinare $v(P)$.
- Sia P la formula $(A \vee B) \rightarrow \perp$. Trovare un'interpretazione v, se esiste, tale che $v(P) = 1$ e un'interpretazione w, se esiste, tale che $w(P) = 0$.

EQUIVALENZA SEMANTICA

Due fbf P e Q sono semanticamente equivalenti se, *per ogni interpretazione* v, risulta $v(P) = v(Q)$. In simboli $P \equiv Q$.

Si osservi che il simbolo $\equiv$ non appartiene all'alfabeto della logica proposizionale. Una scrittura del tipo $P \equiv Q$, dove P e Q denotano due fbf, non è una fbf.

ESEMPIO

Verifichiamo che $\perp \equiv A \wedge \neg A$.

A	$\neg A$	$A \wedge \neg A$
0	1	0
1	0	0

Dalla tabella abbiamo che $v(A \wedge \neg A) = 0$ per ogni v; per definizione abbiamo che $v(\bot) = 0$ per ogni v. Risulta quindi $v(\bot) = v(A \wedge \neg A)$ per ogni v, ovvero $\bot \equiv A \wedge \neg A$.

ESERCIZIO 3. Sia P la fbf $(A \to B) \vee B$ e sia Q la fbf $\neg A \vee B$. Determinare se P e Q sono semanticamente equivalenti.

1.1.6 Formule soddisfacibili, insoddisfacibili e tautologiche

Sia P una fbf e v una interpretazione; diremo che:

- v è un *modello* per P se $v(P) = 1$; in tal caso scriveremo $v \vDash P$ e diremo che v *soddisfa* P
- P è *soddisfacibile* se ha almeno un modello; in caso contrario è insoddisfacibile o contraddittoria.
- P è una *tautologia* se ogni interpretazione è un modello per P. In tale situazione scriveremo $\vDash P$.

Per determinare se una data formula P è contraddittoria, soddisfacibile o tautologica è sufficiente costruire la tavola di verità e valutare tutte le interpretazioni: se il risultato è costantemente 1 vuol dire che P è tautologica, se è costantemente 0 significa che P è contraddittoria, se è 1 almeno una volta allora P è soddisfacibile.

ESEMPI

Siano P_1 la fbf $A \wedge \neg A$, P_2 la fbf $A \vee \neg A$ e P_3 la fbf $A \to \bot$. Dalle tabelle di verità risulta che P_1 è contraddittoria in quanto nessuna interpretazione la rende vera, P_2 è tautologica e anche soddisfacibile in quanto tutte le interpretazioni la rendono vera, P_3 è soddisfacibile ma non tautologica in quanto alcune interpretazioni la rendono vera e altre la rendono falsa.

$P_1 =$

A	$\neg A$	$A \wedge \neg A$
0	1	0
1	0	0

$P_2 =$

A	$\neg A$	$A \vee \neg A$
0	1	1
1	0	1

$P_3 =$

A	$\bot$	$A \to \bot$
0	0	1
1	0	0

ESERCIZIO 4. Verificare quali delle seguenti fbf sono tautologiche, quali soddisfacibili e quali contraddittorie.

- $(A \vee B \vee C) \wedge (A \vee \neg B \vee \neg C)$
- $\neg A \to \bot \vee \top$
- $(A \leftrightarrow B) \wedge (\neg A \leftrightarrow \neg B)$
- $((A \to B) \to C) \leftrightarrow (A \to (B \to C))$

- $((A \vee \bot) \vee (B \vee \bot)) \wedge (C \wedge \bot)$
- $(A \wedge \bot) \rightarrow (A \rightarrow B)$

NOTAZIONE

Possiamo utilizzare un'altra notazione per indicare che una fbf P è una tautologia: possiamo scrivere $P \equiv \top$. Infatti, per la definizione di equivalenza semantica, risulta $v(P) = v(\top)$ per ogni interpretazione v; ma, per definizione, $v(\top) = 1$ per ogni v; questo implica che $v(P) = 1$ per ogni v, ovvero P è una tautologia.

OSSERVAZIONE

Se P e Q sono semanticamente equivalenti, la fbf $P \leftrightarrow Q$ è una tautologia. È vero anche il viceversa: se $P \leftrightarrow Q$ è una tautologia allora $P \equiv Q$.

1.1.7 Proprietà dei connettivi

Diamo qui di seguito una serie di equivalenze semantiche che esprimono le proprietà notevoli dei connettivi.

- idempotenza
$$P \vee P \equiv P$$
$$P \wedge P \equiv P$$
- commutatività
$$P \vee Q \equiv Q \vee P$$
$$P \wedge Q \equiv Q \wedge P$$
- associatività
$$(P \vee Q) \vee R \equiv P \vee (Q \vee R)$$
$$(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$$
- assorbimento
$$P \vee (P \wedge Q) \equiv P$$
$$P \wedge (P \vee Q) \equiv P$$
- distributività
$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$
$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$
- De Morgan
$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$
$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
- doppia negazione
$$\neg\neg P \equiv P$$

- riduzione a vero
$$P \vee \top \equiv \top$$
- riduzione a falso
$$P \wedge \bot \equiv \bot$$
- esistenza degli elementi neutri
$$P \vee \bot \equiv P$$
$$P \wedge \top \equiv P$$
- complemento
$$\neg\top \equiv \bot$$
$$\neg\bot \equiv \top$$
- eliminazione di vero (principio del terzo escluso)
$$P \vee \neg P \equiv \top$$
- eliminazione di falso (contraddizione)
$$P \wedge \neg P \equiv \bot$$
- eliminazione dell'implicazione
$$P \rightarrow Q \equiv \neg P \vee Q$$
- eliminazione dell'equivalenza
$$P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$$

Queste proprietà ci consentono di sostituire parti di fbf con altre semanticamente equivalenti e di ottenere trasformazioni sintattiche di fbf semanticamente equivalenti a quelle di partenza.

ESEMPIO

Sia P la fbf $\neg(A \vee \neg B) \vee B$. Applicando De Morgan e la doppia negazione otteniamo: $\neg(A \vee \neg B) \vee B \equiv (\neg A \wedge B) \vee B$. Per la proprietà di assorbimento si ha: $(\neg A \wedge B) \vee B \equiv B$. Pertanto, possiamo concludere che $\neg(A \vee \neg B) \vee B \equiv B$.

ESERCIZIO 5. Verificare, senza ricorrere alle tavole di verità che valgono le seguenti equivalenze semantiche.

- $\neg A \rightarrow \bot \vee \top \equiv \top$
- $(A \rightarrow B) \rightarrow \neg B \equiv \neg B$
- $(A \leftrightarrow \bot) \equiv \neg A$
- $A \vee (A \rightarrow B) \equiv \top$

- $A \vee (\neg(A \rightarrow B)) \equiv A$
- $\neg(A \vee B \vee C) \vee B \equiv (\neg A \wedge \neg C) \vee B$
- $\neg(A \vee B \vee C) \rightarrow B \equiv A \vee B \vee C$

1.1.8 Tautologie e metodi di dimostrazione

Le tautologie della logica proposizionale sono schemi di fbf che in base alla loro forma, o struttura, esprimono un valore di verità che è costantemente vero. Questa caratteristica è valida anche quando gli schemi vengono istanziati, cioè quando applichiamo una sostituzione alle variabili, a patto di sostituire la stessa espressione in corrispondenza di ogni occorrenza di una data variabile. Questa è la ragione per la quale, in matematica, i teoremi sono frasi dotate di una struttura tautologica. Le tecniche di dimostrazione, basate su tautologie, più comuni sono le seguenti:

- analisi per casi
- dimostrazione per contrapposizione
- dimostrazione per assurdo

Vediamo alcuni esempi concreti di applicazione di tali tecniche. Per ognuna di esse consideriamo dapprima una applicazione a sostegno di una argomentazione propria della vita quotidiana e, in seguito, in maniera più formale, faremo vedere come il suo utilizzo conduca alla dimostrazione di un teorema matematico.

ANALISI PER CASI

Si basa sulla tautologia $\vDash (p \rightarrow q) \wedge (\neg p \rightarrow q) \leftrightarrow q$.

Questo schema tautologico ci consente di trarre la conclusione q a patto che l'ipotesi p implichi q e che la sua negazione $\neg p$ implichi anch'essa q. In un certo senso q risulta vera indipendentemente dalle ipotesi.

Consideriamo il seguente esempio. Immaginiamo che Tizio annunci alla sua consorte l'impossibilità di andare in vacanza a causa dell'ingente somma di tasse da pagare. Di fronte all'incredulità della donna Tizio tenta di convincerla con la seguente argomentazione: "Per andare in vacanza abbiamo bisogno di 1000 euro. La somma a nostra disposizione ammonta a 1500 euro. Purtroppo dobbiamo pagare 1000 euro di tasse e la differenza, 500 euro, non ci consente di affrontare le spese della vacanza. A questo punto se pagassimo le tasse (p) non potremmo andare in vacanza (q). Se non pagassimo le tasse ($\neg p$) e avviassimo la pratica per il dilazionamento del pagamento, la somma da pagare comprensiva di tutte le spese, ammonterebbe a 600 euro. Anche in questo caso potremmo disporre di una somma (900 euro) che non è sufficiente per coprire le spese della vacanza (q). Pertanto, posso senz'altro asserire, nostro malgrado, che non possiamo andare in vacanza (q)."

Come ulteriore esempio, dimostriamo che nell'insieme dei numeri razionali $\mathbf{Q}$ vale la proprietà di Archimede, ovvero che $\forall x, y \in \mathbf{Q}$ con $x, y > 0, \exists n \in \mathbf{N}$ tale che $nx \geq y$.

Consideriamo due numeri razionali positivi x e y.

Caso A: $x \geq y$

In questo caso la proprietà è banalmente verificata per tutti i numeri $n \in \mathbf{N}$.

Caso B: $x < y$

Essendo x e y due numeri razionali positivi essi possono essere scritti come rapporto di due numeri naturali. Sia $x = \frac{r}{p}$ e $y = \frac{s}{t}$. Poiché $x < y$, risulta $rt < sp$. Sia $n = sp$. Verifichiamo che $nx \geq y$. Si ha:

$$nx = sp\frac{r}{p} = sr \geq s \geq \frac{s}{t} = y.$$

DIMOSTRAZIONE PER CONTRAPPOSIZIONE

Si basa sulla tautologia $\vDash (p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$. Per dimostrare la veridicità di una implicazione $(p \rightarrow q)$ è sufficiente dimostrare la veridicità di $(\neg q \rightarrow \neg p)$. Infatti ricordiamo che $\vDash (p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$ equivale a dire $(p \rightarrow q) \equiv (\neg q \rightarrow \neg p)$ ovvero le due fbf hanno lo stesso valore di verità per ogni interpretazione. Posso utilizzare tale schema anche per dimostrare la falsità di una implicazione $p \rightarrow q$: infatti se $\neg q \rightarrow \neg p$ risulta falsa allora anche $p \rightarrow q$ è falsa come conseguenza della relazione di equivalenza semantica.

Nel seguente esempio, Tizio sostiene che se una persona pratica una attività sportiva con costanza (p) allora dimagrisce. Caio non è convinto della validità di tale affermazione e prova a confutarla. Caio dice a Tizio: "se fosse vero il tuo ragionamento $(p \rightarrow q)$ allora dovrebbe essere anche vero che se una persona non dimagrisce $(\neg q)$ allora non pratica un'attività sportiva con costanza $(\neg p)$. Ma questo non è vero: ci sono centinaia e centinaia di atleti che, pur allenandosi costantemente, mantengono il loro peso forma. In realtà l'asserzione corretta è la seguente: se una persona consuma più calorie di quante ne assume (p) allora dimagrisce (q). Infatti, supponiamo che una persona non dimagrisce $(\neg q)$; allora questo implica che assume un numero di calorie almeno pari a quante ne consuma $(\neg p)$. Da qui deriva la correttezza di $p \rightarrow q$.

Dimostriamo ora, usando la dimostrazione per contrapposizione, che ogni sottoinsieme finito A di $\mathbf{N}$ ammette massimo ($M = \max\{A\}$ è un numero appartenente ad A tale che $\forall a \in A : a \leq M$).

Supponiamo che A non ammetta massimo. Allora comunque scelgo un elemento a di A posso trovare un elemento M' di A tale che $a \leq M'$. M' non è il massimo perché per ipotesi $\max\{A\}$ non esiste. Ciò implica che A contiene un elemento M'' tale che $M' < M''$. Questo procedimento non ha termine (perché, per ipotesi, A non possiede un elemento massimo). Concludiamo che A contiene un numero infinito di elementi.

DIMOSTRAZIONE PER ASSURDO

Si basa sulla tautologia $\vDash (\neg p \rightarrow \bot) \leftrightarrow p$. Per dimostrare p facciamo vedere che non può essere altrimenti, cioè che $\neg p$ conduce ad una contraddizione, ad un assurdo $(\bot)$. Abbiamo un assurdo ogniqualvolta in un ragionamento asseriamo che una frase q è vera e che anche $\neg q$ è vera. Tra le proprietà dei connettivi ne compare una detta *contraddizione* secondo

la quale $q \wedge \neg q \equiv \bot$. In definitiva, se supponendo $\neg p$ vera otteniamo una contraddizione, necessariamente $\neg p$ deve essere falsa e quindi p è vera.

Diamo un esempio. La Signora Tizio e suo marito hanno un negozio di abbigliamento. Durante il periodo dei saldi, vuoi per la stanchezza, vuoi per la confusione, Tizio pasticcia un pochino con gli sconti che sono fissati al 10%. Sua moglie si accorge che Tizio ha applicato una percentuale maggiore di sconto ad una cliente che aveva acquistato due capi. Al quesito della moglie, Tizio risponde così "la signorina ha comprato due capi ognuno con il 10% di sconto: 10% più 10% è uguale a 20%". La Signora Tizio gli fa capire che l'idea di sommare le due percentuali di sconto è completamente sbagliata dicendogli: "supponiamo vera la tua tesi ($\neg p$). Allora se la signorina avesse comprato 10 capi con lo sconto del 10% tu le avresti fatto lo sconto del 100%. Cioè glieli avresti regalati!! Ma questo è assurdo ($\bot$). In definitiva non è vero che la percentuale di sconto da applicare a due capi con lo sconto del 10% è del 20% (p).

Dimostriamo che $\sqrt{2}$ non è un numero razionale. Supponiamo per assurdo che $\sqrt{2} \in \mathbf{Q}$. Allora $\sqrt{2}$ può essere scritto come rapporto di interi $\frac{m}{n}$. Siano m ed n primi tra loro (in caso contrario potremmo operare una semplificazione e ricondurci alla ipotesi fatta). Pertanto $\sqrt{2} = \frac{m}{n}$, da cui si ha $2 = \frac{m^2}{n^2}$ e $m^2 = 2n^2$, quindi m^2 è pari; allora anche m è pari, ovvero, $m = 2k$. Sostituiamo $2k$ a m e effettuiamo i seguenti passaggi:

$$2 = \frac{(2k)^2}{n^2} \qquad 2 = \frac{4k^2}{n^2} \qquad 2n^2 = 4k^2 \qquad n^2 = 2k^2$$

ovvero n^2 è pari e quindi anche n è pari. Di conseguenza m e n risultano entrambi pari il che contraddice l'ipotesi che m e n fossero primi tra loro.

1.1.9 Basi di connettivi

Alcune delle proprietà dei connettivi rivestono un ruolo maggiormente significativo. Esse sono: l'eliminazione di vero, l'eliminazione di falso, l'eliminazione dell'implicazione e l'eliminazione dell'equivalenza.

Queste proprietà ci consentono di scrivere le formule della logica proposizionale utilizzando solamente i connettivi $\{\wedge, \vee, \neg\}$ mantenendo inalterato il potere espressivo. Di fatto eliminando i connettivi $\{\top, \bot, \rightarrow, \leftrightarrow\}$ applicando le rispettive regole di eliminazione otteniamo fbf equivalenti a quelle di partenza. Le trasformazioni sintattiche non alterano il valore semantico delle fbf.

ESEMPI

Sia $P = (A \leftrightarrow B) \vee C$. Applicando l'eliminazione dell'equivalenza otteniamo: $(A \rightarrow B \wedge B \rightarrow A) \vee C$. Applicando l'eliminazione dell'implicazione otteniamo: $(\neg A \vee B) \wedge (\neg B \vee A) \vee C$. Abbiamo ottenuto una formula in cui compaiono solo i connettivi $\{\wedge, \vee, \neg\}$ e che è semanticamente equivalente a P.

Sia $P = (A \rightarrow \bot) \wedge (A \rightarrow \top)$. Applicando l'eliminazione di falso e di vero otteniamo: $(A \rightarrow (\neg A \wedge A)) \wedge (\neg A \vee (\neg A \vee A))$. Semplifichiamo ulteriormente, applicando l'assorbimento e l'idempotenza, per avere: $\neg A \wedge (\neg A \vee A)$. Utilizzando ancora la proprietà dell'assobimento:

$\neg A$. Anche in questo caso abbiamo ottenuto una formula equivalente a P utilizzando solo i connettivi $\{\wedge, \vee, \neg\}$.

L'insieme dei connettivi $\{\wedge, \vee, \neg\}$ è un sottoinsieme proprio dei connettivi introdotti nella sintassi della logica proposizionale, dotato dello stesso potere espressivo dell'insieme completo.

Possiamo ridurre ulteriormente tale insieme grazie alle regole di De Morgan e della doppia negazione. Infatti data una fbf P costituita con i connettivi $\{\wedge, \vee, \neg\}$ è sempre possibile trovare P', P'' in cui compaiono rispettivamente solo i connettivi $\{\neg, \wedge\}$ e $\{\neg, \vee\}$ tali che $P' \equiv P$ e $P'' \equiv P$.

ESEMPIO

Sia $P = (A \wedge B) \vee \neg C \vee \neg B$. Applicando la doppia negazione otteniamo: $(\neg\neg A \wedge \neg\neg B) \vee \neg C \vee \neg B$. Applicando De Morgan, abbiamo $\neg(\neg A \vee \neg B) \vee \neg C \vee \neg B$. Abbiamo ottenuto una fbf equivalente a P costruita con il connettivi $\{\neg, \vee\}$. Ripetiamo il procedimento per avere una fbf in cui compaiono solo i connettivi $\{\neg, \wedge\}$. Applichiamo De Morgan a P per avere: $(A \wedge B) \vee \neg(C \wedge B)$. Applichiamo la doppia negazione e otteniamo: $\neg\neg(A \wedge B) \vee \neg(C \wedge B)$. Applicando nuovamente De Morgan: $\neg(\neg(A \wedge B) \wedge (C \wedge B))$.

Risulta evidente che gli insiemi $\{\wedge, \vee, \neg\}$, $\{\wedge, \neg\}$, $\{\vee, \neg\}$ hanno la stessa capacità di espressione di $\{\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \top, \bot\}$: ogni fbf P costruita utilizzando l'insieme completo dei connettivi può essere sostituita con una fbf P' costruita scegliendo uno dei tre sottoinsiemi su indicati in maniera tale che risulti semanticamente equivalente a P.

Chiameremo *base di connettivi* ogni sottoinsieme di $\{\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \top, \bot\}$ che abbia questa caratteristica.

1.1.10 Forma normale congiuntiva e disgiuntiva

Possiamo trasformare sintatticamente una fbf e ottenerne un'altra ad essa equivalente e che inoltre abbia una forma canonica prestabilita. Tale forma canonica è detta normale. Le forme normali che consideriamo sono le cosiddette forme normali congiuntiva e disgiuntiva.

Chiamiamo letterale una formula atomica o la sua negazione.

Una fbf P è in forma normale congiuntiva (FNC) se $P = P_1 \wedge P_2 \wedge \ldots \wedge P_n$ con $n \geq 1$ e ogni P_i è una disgiunzione di letterali.

Una fbf P è in forma normale disgiuntiva (FND) se $P = P_1 \vee P_2 \vee \ldots \vee P_n$ con $n \geq 1$ e ogni P_i è una congiunzione di letterali.

ESEMPI

La fbf $(A \wedge B) \vee (\neg B \wedge A)$ è in FND; la fbf $(\neg A \vee \neg B) \wedge C$ è in FNC; la fbf $\neg A \vee C \vee \neg B$ è in FND e anche in FNC; la fbf $B \wedge \neg C$ è sia in FND che in FNC.

Data una funzione $f : \{0,1\}^n \mapsto \{0,1\}$ è possibile trovare una fbf P in forma normale congiuntiva o disgiuntiva tale che la funzione espressa da P sia proprio f. Consideriamo un esempio concreto.

Sia $f : \{0,1\}^3 \mapsto \{0,1\}$ la seguente funzione:

A_1	A_2	A_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

f può essere espressa come disgiunzione o come congiunzione di altre funzioni. Infatti $f = f_0 \wedge f_1 \wedge f_2 \wedge f_3 \wedge f_4$ e $f = f_5 \vee f_6 \vee f_7$:

f_0	f_1	f_2	f_3	f_4	f
0	1	1	1	1	0
1	1	1	1	1	1
1	0	1	1	1	0
1	1	1	1	1	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	1

f_5	f_6	f_7	f
0	0	0	0
1	0	0	1
0	0	0	0
0	1	0	1
0	0	0	0
0	0	0	0
0	0	0	0
0	0	1	1

Vediamo come ottenere le funzioni f_i, con $0 \leq i \leq 7$. Consideriamo le funzioni del primo gruppo; f_0 è la funzione che ha tutti 1 e un solo 0 in corrispondenza dell'input $(0, 0, 0)$. La fbf $A_1 \vee A_2 \vee A_3$ esprime la funzione f_0: infatti $v(A_1 \vee A_2 \vee A_3) = 0$ solo quando $A_1 = 0$, $A_2 = 0$ e $A_3 = 0$; vale 1 per tutte le altre interpretazioni. Allo stesso modo si può verificare che le seguenti fbf $A_1 \vee \neg A_2 \vee A_3$, $\neg A_1 \vee A_2 \vee A_3$, $\neg A_1 \vee A_2 \vee \neg A_3$, $\neg A_1 \vee \neg A_2 \vee A_3$ esprimono rispettivamente le funzioni f_1, f_2, f_3, f_4.

Per le funzioni del secondo gruppo il ragionamento è analogo: f_5 ha tutti 0 e un solo 1 in corrispondenza di (0,0,1). La fbf $\neg A_1 \wedge \neg A_2 \wedge A_3$ esprime tale funzione: infatti l'unica interpretazione che rende vera tale fbf è $v(A_1) = 0$, $v(A_2) = 0$ e $v(A_3) = 1$; tutte le altre interpretazioni la rendono falsa. Le fbf corrispondenti a f_6 e f_7 sono rispettivamente $\neg A_1 \wedge A_2 \wedge A_3$ e $\neg A_1 \wedge \neg A_2 \wedge \neg A_3$.

In generale per costruire una fbf P in FNC associata ad una funzione $f : \{0, 1\}^n \mapsto \{0, 1\}$ procediamo nella maniera seguente: per ogni 0 della funzione f che compare in tabella scriviamo una disgiunzione di letterali A_i presi in forma vera se $v(A_i) = 0$ e in forma negata se $v(A_i) = 1$.

Analogamente, per costruire una fbf P in FND associata ad una funzione $f : \{0, 1\}^n \mapsto \{0, 1\}$ procediamo nella maniera seguente: per ogni 1 della funzione f che compare in tabella scriviamo una congiunzione di letterali A_i presi in forma negata se $v(A_i) = 0$ e in forma vera se $v(A_i) = 1$.

ESERCIZI

- Sia $f : \{0,1\}^2 \mapsto \{0,1\}$ la seguente funzione:

A_1	A_2	f
0	0	1
0	1	0
1	0	0
1	1	0

  Trovare la FND e la FNC.

- Sia $f : \{0,1\}^4 \mapsto \{0,1\}$ la funzione che vale 1 in corrispondenza di v_0, v_3, v_{10}, v_{15} e che vale 0 per le altre interpretazioni. Trovare la FND e la FNC.

- Sia $P = (A \vee B) \rightarrow (A \wedge C)$. Trovare la FND e la FNC.

1.1.11 Insiemi funzionalmente completi

Un insieme di connettivi A si dice *funzionalmente completo* se per ogni funzione $f : \{0,1\}^n \mapsto \{0,1\}$ esiste una fbf P costruita con i connettivi di A tale che la funzione espressa da P sia f.

La base di connettivi $\{\wedge, \vee, \neg\}$ è funzionalmente completa: infatti qualsiasi funzione $f : \{0,1\}^n \mapsto \{0,1\}$ può essere espressa tramite una fbf in forma normale congiuntiva o disgiuntiva. Una fbf in forma normale congiuntiva o disgiuntiva può essere trasformata in una fbf equivalente, tramite le leggi di De Morgan, in cui compaiono solo i connettivi $\{\wedge, \neg\}$ oppure $\{\vee, \neg\}$: risulta, pertanto, che gli insiemi $\{\wedge, \neg\}$ e $\{\vee, \neg\}$ sono funzionalmente completi.

A questo punto è spontaneo chiedersi se esistono insiemi di connettivi funzionalmente completi aventi un solo elemento. Consideriamo di nuovo la tabella delle funzioni binarie:

argomenti	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Introduciamo due connettivi che in prima istanza non sono stati considerati nella sintassi della logica proposizionale. Essi sono il *nor*, denotato con il simbolo $\oplus$, e il *nand*, denotato con $\otimes$, corrispondenti rispettivamente alle funzioni b_8 e b_{14}. Ebbene, gli insiemi $\{\oplus\}$ e $\{\otimes\}$ sono funzionalmente completi.

ESERCIZIO 6. Dimostrare che $\{\oplus\}$ e $\{\otimes\}$ sono funzionalmente completi. A tal fine si trovino due fbf costruite con il connettivo $\oplus$ che esprimano rispettivamente i connettivi $\neg$ e $\wedge$ (oppure $\neg$ e $\vee$). Analogamente per il connettivo $\otimes$.

1.1.12 Limiti della logica proposizionale

Il perno centrale della logica proposizionale, su cui ruotano i meccanismi del linguaggio, è costituito dai connettivi: questi, infatti, dal punto di vista sintattico, consentono di costruire frasi composte combinando frasi più semplici e, sotto l'aspetto semantico, forniscono regole che determinano il valore di verità delle proposizioni composte a partire dai valori di verità delle proposizioni componenti.

Le proposizioni atomiche vengono assunte come entità elementari: non si indaga sulla loro struttura interna e ciò rappresenta un limite perché i meccanismi di formalizzazione risultano insufficienti per gestire, ad esempio, i ragionamenti che riguardono generalità vale a dire la possibilità di esprimere che una certa proprietà sussiste per tutti gli oggetti del dominio del discorso o che esiste almeno un oggetto che gode di tale proprietà. Un esempio potrà chiarire questo aspetto.

Consideriamo il sillogismo di Socrate "ogni uomo è mortale, Socrate è un uomo, Socrate è mortale". Riscrivendo tale frase con la notazione simbolica della logica proposizionale otteniamo una fbf della forma $A \wedge B \rightarrow C$. Tale proposizione non è una tautologia e non esprime la intuitiva correttezza del sillogismo. In effetti il collegamento tra le frasi elementari del sillogismo non viene espresso attraverso le proposizioni atomiche A, B e C. Questo accade perché la logica proposizionale si limita a considerare le proposizioni composte come funzioni di quelle atomiche senza scomporre ulteriormente queste ultime, benché esse non costituiscano gli elementi più semplici del ragionamento e posseggano a loro volta una struttura interna.

1.2 Logica dei predicati

La logica dei predicati (o del primo ordine) risulta sufficientemente espressiva da permettere la formalizzazione di gran parte dei ragionamenti che ricorrono nel linguaggio naturale e nella matematica.

Essa consente di esprimere ("predicare") certe caratteristiche che riguardano gli elementi del dominio del discorso mediante i predicati. Esaminiamo come essi operano e quale relazione intercorre tra i predicati e le variabili proposizionali. Consideriamo ad esempio il seguente fatto: "Paolo ha un cappotto". Per formalizzare tale concetto nella logica proposizionale utilizziamo una variabile P la quale vale vero se Paolo ha effettivamente un cappotto oppure falso in caso contrario. Per esprimere il fatto che "Enrica ha un cappotto" dovremmo usare un'altra variabile P'. Se volessimo esprimere il fatto che tutte le persone hanno un cappotto dovremmo innanzitutto conoscere il nome di tali persone e, ammesso ciò, impiegare un numero molto grande di variabili proposizionali (una per ogni persona). Gestire un numero molto grande di variabili proposizionali rappresenta sicuramente un inconveniente; gestirne un numero infinito è invece impossibile. Questo caso si presenta quando i domini sono infiniti, come ad esempio i numeri: come potremmo esprimere il fatto che tutti i numeri naturali sono pari? La sequenza di variabili P_i per denotare che "i è un numero pari" non finirebbe mai!

I predicati consentono di esprimere in maniera concisa il fatto che gli elementi del dominio del discorso soddisfino, o meno, una certa proprietà. Sia $P(x)$ un predicato tramite il quale ad ogni elemento x del dominio viene associato un valore di verità: vero, se x soddisfa la proprietà P, falso, altrimenti.

Negli esempi precedenti:

- se, nel dominio degli individui, $P(x)$ denota "x ha un cappotto", allora P(Paolo) è vero se Paolo ha un cappotto, altrimenti è falso;
- se, nel dominio dei numeri naturali, $P(x)$ denota "x è un numero pari", allora $P(2)$ è vero e $P(1)$ è falso.

I predicati costituiscono una generalizzazione delle variabili proposizionali: queste ultime, infatti, possono essere considerate come predicati nullari, vale a dire senza argomenti. In generale, un predicato n-ario $P(x_1, \ldots, x_n)$ denota un valore di verità: vero, se $x_1, \ldots, x_n$ sono in relazione tra loro secondo P, falso altrimenti. I predicati da soli non consentono di formalizzare concetti del tipo "esiste un elemento del dominio che gode di una certa proprietà P" oppure "tutti gli elementi del dominio soddisfano la proprietà P". Vediamo perché. Fino ad ora abbiamo introdotto due concetti: il dominio (cioè un insieme di elementi) e i predicati, funzioni che associano alle n-uple degli elementi del dominio un valore di verità. Fissato ad esempio il dominio degli individui, la scrittura $P(x)$ indica "x ha un cappotto" e denota un valore di verità a seconda di come viene istanziata la variabile x. Invece P(Paolo) denota vero o falso a seconda che Paolo abbia o meno un cappotto. Tuttavia, il vantaggio che i predicati introducono rispetto alle variabili proposizionali si concretizza solo tramite l'uso dei quantificatori. Infatti se volessimo asserire "qualcuno ha un cappotto" oppure "tutti hanno un cappotto" potremmo provare rispettivamente con una disgiunzione di predicati del tipo P(Paolo)$\vee P$(Enrica)$\vee \ldots$ e con una congiunzione di predicati come P(Paolo)$\wedge P$(Enrica)$\wedge \ldots$. Ma queste soluzioni ripropongono gli stessi problemi delle variabili proposizionali:

- non è agevole scrivere un'espressione con un numero molto grande di predicati
  (è impossibile quando il dominio è infinito)
- non sempre conosciamo tutti i nomi dell'insieme degli elementi di cui parliamo

La logica dei predicati mette a disposizione l'operatore $\exists$ (esiste) che viene chiamato quantificatore esistenziale e l'operatore $\forall$ (per ogni) che viene chiamato quantificatore universale.

L'operatore $\exists$ ha il potere espressivo di una disgiunzione di un numero infinito di predicati: esso permette di costruire formule come $\exists x P(x)$ il cui significato è "esiste (almeno) un elemento del dominio che soddisfa P".

L'operatore $\forall$ ha il potere espressivo di una congiunzione di un numero infinito di predicati: il significato della formula $\forall x P(x)$ è "ogni elemento del dominio soddisfa P".

Quindi per esprimere il fatto "se fa freddo qualcuno ha un cappotto" possiamo scrivere $F \rightarrow \exists x P(x)$ dove F indica "fa freddo" e $P(x)$ "x ha un cappotto". Analogamente possiamo esprimenre che "se tutti hanno un cappotto allora almeno uno ha un cappotto" nella seguente maniera $\forall x P(x) \rightarrow \exists x P(x)$.

In definitiva, la logica dei predicati può essere considerata come un'estensione di quella proposizionale con le nozioni aggiuntive di: quantificatori, predicati, funzioni, variabili e costanti.

- I quantificatori universale ed esistenziale permettono di considerare rispettivamente le generalità degli oggetti del dominio del discorso e l'esistenza di oggetti in una data relazione.
- I predicati consentono di esprimere proprietà e relazioni sugli oggetti del dominio.
- Le funzioni rappresentano oggetti del dominio del discorso definiti univocamente a partire da altri. Ad esempio, nel dominio dei numeri naturali sia $s(x, y)$ la funzione somma che restituisce il risultato $x + y$. Se $P(x)$ è il predicato "x è un numero pari" allora $P(s(3, 2))$ è falso mentre $P(s(3, 3))$ è vero.
- Le variabili e le costanti denotano oggetti del dominio del discorso. Ad esempio nel caso dei numeri naturali, $1, 2, 3, \ldots$ sono costanti mentre $x, y, z, \ldots$ sono variabili.

1.2.1 Sintassi della logica dei predicati

L'alfabeto della logica dei predicati è composto da:

- simboli di costante: $\bot$, $\top$, $a, b, c, \ldots$
- un insieme infinito VAR di simboli di variabile: $x, y, z, \ldots$
- simboli di funzione: $f, g, h, \ldots$
- simboli di predicato: $A, B, C \ldots$
- connettivi: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
- quantificatori: $\forall, \exists$
- simboli ausiliari: (e).

TER è l'insieme dei termini ed è composto da:

- costanti
- variabili
- se $t_1, \ldots, t_n \in$ TER e f^n è un simbolo di funzione del linguaggio allora $f^n(t_1, \ldots, t_n) \in$ TER con $n \geq 1$.

Si noti che le costanti sono funzioni di arietà 0, cioè funzioni senza argomenti.
L'insieme FBF delle formule ben formate è così costituito:

- $\bot, \top \in$ FBF;
- se $t_1, \ldots t_n \in$ TER e A^n è un simbolo di predicato allora $A^n(t_1, \ldots t_n) \in$ FBF con $n \geq 0$;
- se $P \in$ FBF allora $\neg P \in$ FBF;

- se $P, Q \in$ FBF allora $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$, $P \leftrightarrow Q \in$ FBF;
- se $P \in$ FBF allora $((\forall x)P)$, $((\exists x)P) \in$ FBF.

Le fbf dei primi due punti sono dette *atomi*.

Nel seguito ometteremo abitualmente l'apice n nelle espressioni $f^n(t_1, \ldots, t_n)$ e $A^n(t_1, \ldots t_n)$ poiché è implicitamente definito dal numero degli argomenti.

REGOLE DI PRECEDENZA

Utilizziamo le seguenti regole di precedenza per eliminare, quando è possibile, le parentesi:

$$\forall \; = \; \exists \; = \; \neg \; > \; \wedge \; > \; \vee \; > \; \rightarrow \; > \; \leftrightarrow$$

In base a tale convenzione possiamo scrivere $\forall x(A(x) \rightarrow B(x)) \wedge \exists y \neg A(y)$ al posto di $(((\forall x)(A(x) \rightarrow B(x)) \wedge ((\exists y)(\neg A(x)))$.

SOTTOFORMULE

Intuitivamente una sottoformula di una fbf P è una porzione di P che è a sua volta una fbf. Formalmente:

- se P è $\bot$, $\top$ oppure $A^n(t_1, \ldots t_n)$ allora P stessa è la sua sottoformula;
- se P è $\neg P_1$ allora le sottoformule di P sono P stessa e quelle di P_1;
- se P è $P_1 \wedge P_2$, $P_1 \vee P_2$, $P_1 \rightarrow P_2$, $P_1 \leftrightarrow P_2$ allora le sue sottoformule sono P stessa e quelle di P_1 e P_2;
- se P è $((\forall x)P_1)$, $((\exists x)P_1)$ allora le sue sottoformule sono P stessa e quelle di P_1.

ESEMPIO

Sia P la fbf $\exists x(A(x) \rightarrow B(x)) \wedge \neg \forall y(B(f(y), x))$. Le sue sottoformule sono:

- P
- $\exists x(A(x) \rightarrow B(x))$
- $\neg \forall y(B(f(y), x))$
- $A(x) \rightarrow B(x)$, $A(x)$, $B(x)$
- $\forall y(B(f(y), x))$, $B(f(y), x)$.

1.2.2 Campo d'azione dei quantificatori

Le variabili nella logica dei predicati hanno una duplice funzione: compaiono in espressioni atomiche e in espressioni quantificate. Ad esempio in $P(x)$ la variabile x denota un generico elemento del dominio e il valore di verità di $P(x)$ dipende da quale elemento del dominio sostituiamo a x. Consideriamo il dominio dei naturali; sia $P(x)$ il predicato "x è un numero pari" e sia $s(x)$ la funzione successore. La fbf $P(x) \wedge P(s(x))$ è falsa qualunque elemento sostituiamo a x. Se nella fbf cambiamo una delle due occorrenze della x con un altro simbolo di variabile, ad esempio y, otteniamo un'altra fbf con un significato differente. Ad esempio, la fbf $P(x) \wedge P(s(y))$, nel dominio dei numeri naturali, è vera o falsa a seconda delle sostituzioni che operiamo su x e y: $P(2) \wedge P(s(2))$ è falsa e $P(2) \wedge P(s(3))$ è vera. Le due fbf non sono equivalenti e non è lecito operare tale sostituzione se vogliamo preservare l'equivalenza.

Consideriamo adesso una *fbf quantificata*: $\forall x P(x)$, il cui significato è "ogni elemento del dominio soddisfa P". Se cambiamo nome e utilizziamo la variabile y otteniamo $\forall y P(y)$ che denota "ogni elemento del dominio soddisfa P" cioè $\forall x P(x)$ ha lo stesso significato di $\forall y P(y)$. Analogamente per il quantificatore esistenziale. In questi casi la sostituzione di una variabile con un'altra è lecita (si parla più correttamente di *ridenominazione*) perché non cambia il significato della frase.

Consideriamo un'espressione del tipo $\forall x A(x) \vee B(x)$ e supponiamo di voler cambiare nome alla variabile quantificata, cioè alla x: bisogna capire, al fine di interpretare correttamente la scrittura, se la x relativa al quantificatore concerne solo l'atomo A o anche l'atomo B.

Definiamo, a tal proposito, il campo d'azione di un quantificatore come la sottoformula immediatamente alla sua destra. Esso definisce la portata della quantificazione ovvero la sottoespressione sintattica su cui essa ha effetto. Pertanto in $\forall x A(x) \vee B(x)$ il campo d'azione di $\forall$ è $A(x)$; nella fbf $\forall x(A(x) \vee B(x))$ il campo d'azione di $\forall$ è $(A(x) \vee B(x))$.

Inoltre ogni variabile appartiene al campo d'azione del quantificatore più interno che la menziona. Ad esempio nella fbf $\forall x((A(x) \vee \exists x B(x))$ la x in $(A(x)$ appartiene al campo d'azione del quantificatore universale mentre la x in $B(x)$ appartiene al campo d'azione del quantificatore esistenziale. Una espressione equivalente è $\forall z((A(z) \vee \exists y B(y))$. Infatti in entrambe la denotazione semantica è la seguente: per ogni elemento del dominio accade che A è vera per tale elemento oppure esiste un secondo elemento (ne esiste almeno uno per ogni elemento del dominio e può eventualmente coincidere con il primo) tale che B è vera.

Una interpretazione di tale fbf potrebbe essere la seguente. Consideriamo il dominio degli individui; siano $A(x)$ e $B(x)$ rispettivamente il predicato "x è femmina" e "x è maschio". La fbf $\forall z((A(z) \vee \exists y B(y))$, interpretata, diventa: per ogni individuo z, accade che z è femmina oppure esiste (per ogni z) un individuo y che è maschio.

Lasciamo al lettore il compito di stabilire se tale interpretazione rende vera la fbf. Si trovi inoltre un'altra interpretazione nel dominio dei numeri naturali.

1.2.3 Variabili libere e legate

Focalizziamo la nostra attenzione sulla distinzione tra variabili libere e legate che abbiamo accennato nel paragrafo precedente. Una variabile che rientra nel campo d'azione di un quantificatore si dice legata; libera, altrimenti:

- una variabile legata x denota un *generico* elemento del dominio cui diamo il nome x
- una variabile libera x denota un *ben determinato* elemento del dominio che chiamiamo x.

Precisiamo meglio questa affermazione. La semantica di una fbf del tipo $\forall x(A(x))$ è la seguente: per ogni elemento del dominio vale la proprietà A (questa affermazione può essere vera o falsa a seconda del dominio e del predicato A). La semantica di una fbf come $\exists x(A(x))$ è esiste (almeno) un elemento del dominio che soddisfa la proprietà A (anche qui l'affermazione risulta vera o falsa in relazione al dominio e al predicato A). Pertanto il significato delle espressioni quantificate è rappresentato da locuzioni verbali in cui x può anche non comparire (esiste (almeno) un elemento del dominio..., per ogni elemento del dominio...): vale a dire una variabile legata rappresenta un nome qualsiasi. Per contro, la variabile libera x in $B(x)$ rappresenta una variable nel senso "matematico" che conosciamo, la quale nell' interpretazione semantica, verrà istanziata: una variabile libera denota un preciso elemento del dominio.

Un esempio potrà chiarire meglio la differenza tra variabili libere e legate. Consideriamo la fbf $A(x) \vee B(x)$ e supponiamo che qualcuno sostenga che tale fbf è vera nel dominio dei naturali interpretando A come "x è pari" e B come "x è primo" e operando la seguente sostituzione $A(2) \vee B(3)$. La fbf risulta vera ma la sostituzione effettuata non è lecita: x denota lo stesso elemento! Per contro consideriamo la fbf $\exists x A(x) \vee \exists x B(x)$: in questo caso, affinché la fbf sia vera, deve esistere un elemento del dominio che renda vera $A(x)$ o un elemento del dominio che renda vera $B(x)$. Ebbene, sostituendo 2 ad x in $A(x)$ e 3 ad x in $B(x)$, otteniamo una fbf vera operando una sostituzione lecita.

Definiamo formalmente l'insieme delle variabili libere, FV (*Free Variables*) di una fbf.

Sia $t \in$ TER; l'insieme $\mathrm{FV}(t)$ delle variabili libere in t è definito come segue:

- $\mathrm{FV}(x) = \{x\}$ per una variabile x
- $\mathrm{FV}(c) = \emptyset$ per una costante c
- $\mathrm{FV}(f(t_1, \ldots, t_n)) = \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n)$ per una funzione n-aria f

Per ogni fbf P, l'insieme $\mathrm{FV}(P)$ delle variabili libere di P è definito da:

- $\mathrm{FV}(\bot) = \emptyset$, $\mathrm{FV}(\top) = \emptyset$
- $\mathrm{FV}(A(t_1, \ldots, t_n)) = \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n)$
- $\mathrm{FV}(\neg P_1) = \mathrm{FV}(P_1)$
- $\mathrm{FV}(P_1 \wedge P_2) = \mathrm{FV}(P_1) \cup \mathrm{FV}(P_2)$

- $\mathrm{FV}(P_1 \vee P_2) = \mathrm{FV}(P_1) \cup \mathrm{FV}(P_2)$
- $\mathrm{FV}(P_1 \to P_2) = \mathrm{FV}(P_1) \cup \mathrm{FV}(P_2)$
- $\mathrm{FV}(P_1 \leftrightarrow P_2) = \mathrm{FV}(P_1) \cup \mathrm{FV}(P_2)$
- $\mathrm{FV}(\forall x\ P_1) = \mathrm{FV}(P_1) \setminus \{x\}$
- $\mathrm{FV}(\exists x\ P_1) = \mathrm{FV}(P_1) \setminus \{x\}$

ESEMPIO

Vediamo quali sono le variabili libere della fbf $\exists x A(x,y) \leftrightarrow (\exists y B(y) \wedge C(a,z))$ applicando la definizione.

$\mathrm{FV}(\exists x\ A(x,y) \leftrightarrow (\exists y\ B(y) \wedge C(a,z))) = \mathrm{FV}(\exists x\ A(x,y)) \cup \mathrm{FV}(\exists y\ B(y) \wedge C(a,z))) =$
$(\mathrm{FV}(A(x,y)) \setminus \{x\}) \cup (\mathrm{FV}(\exists y\ B(y)) \cup \mathrm{FV}(C(a,z))) =$
$(\mathrm{FV}(x) \cup \mathrm{FV}(y) \setminus \{x\}) \cup (\mathrm{FV}(B(y)) \setminus \{y\}) \cup (\mathrm{FV}(a) \cup \mathrm{FV}(z)) =$
$(\{x\} \cup \{y\} \setminus \{x\}) \cup (\mathrm{FV}(y) \setminus \{y\}) \cup (\emptyset \cup \{z\}) = \{y\} \cup (\{y\} \setminus \{y\}) \cup \{z\} = \{y\} \cup \{z\} = \{y,z\}$

Una fbf P è detta chiusa se $\mathrm{FV}(P) = \emptyset$; aperta altrimenti.

Ovviamente una singola occorrenza di una variabile o è libera o è legata. Tuttavia, si noti che una stessa variabile può occorrere in una fbf sia libera che legata. Si consideri ad esempio la fbf $\forall x(A(x,y) \to B(x)) \wedge \forall y(\neg Q(x,y) \to \forall z R(z))$. Salta subito all'occhio, senza applicare rigorosamente la definizione, che le variabili libere sono $\{x,y\}$; quelle legate sono $\{x,y,z\}$. L'intersezione degli insiemi delle variabili libere e legate è non vuota. L'insieme $Bv(P)$ delle variabili legate di una formula P si ricava eliminando dall'insieme delle occorrenze di variabile di P, l'insieme delle occorrenze di variabile libere.

1.2.4 Sostituzione

Uno dei motivi per i quali si pone in rilievo la distinzione tra variabili libere e variabili legate è la sostituzione dei termini alle variabili libere.

Sia, ad esempio, P la seguente fbf: $\exists x\ A(s(x,y))$. Consideriamo il dominio dei naturali e interpretiamo $A(x)$ come "x è pari" e $s(x,y)$ come "$x+y$". Ci chiediamo se sostituendo un termine, ad esempio 2, alla variabile libera y, la formula risulta vera. Ovvero, $\exists x\ A(s(x,2))$ è vera? Esiste un numero naturale che sommato a 2 dia come risultato un numero pari? La risposta è si.

Nel dominio degli individui interpretiamo $S(x,y)$ come "il suocero di y nel caso in cui x e y siano sposati, y altrimenti" e $A(x)$ il predicato "x ha 100 anni". Consideriamo il termine Paolo e ci chiediamo se $\exists x\ A(s(x,\mathrm{Paolo}))$ è vera; distinguiamo due casi: se Paolo è sposato la fbf è vera se il padre della moglie di Paolo ha 100 anni, altrimenti, se Paolo non è sposato, la fbf è vera se Paolo ha 100 anni.

Diamo la definizione formale di sostituzione di una variabile x con un termine t in una formula P, che denoteremo con $P[\frac{t}{x}]$. Definiamo anzitutto la sostituzione all'interno dei termini.

Siano $s, t \in \text{TER}$, allora $s[\frac{t}{x}]$ è definito nel seguente modo:

- se s è una variabile y allora $y[\frac{t}{x}] = \begin{cases} y & \text{se } y \neq x \\ t & \text{se } y = x \end{cases}$
- se s è una costante c, allora $c[\frac{t}{x}] = c$
- $f(t_1, \ldots, t_n)[\frac{t}{x}] = f(t_1[\frac{t}{x}], \ldots, t_n[\frac{t}{x}])$

Siano $P \in \text{FBF}$ e $t \in \text{TER}$ allora $P[\frac{t}{x}]$ è definito nel seguente modo:

- $\bot[\frac{t}{x}] = \bot$, $\top[\frac{t}{x}] = \top$
- $A(t_1, \ldots, t_n)[\frac{t}{x}] = A(t_1[\frac{t}{x}], \ldots, t_n[\frac{t}{x}])$
- $(\neg P_1)[\frac{t}{x}] = \neg P_1[\frac{t}{x}]$
- $(P_1 \wedge P_2)[\frac{t}{x}] = P_1[\frac{t}{x}] \wedge P_2[\frac{t}{x}]$
- $(P_1 \vee P_2)[\frac{t}{x}] = P_1[\frac{t}{x}] \vee P_2[\frac{t}{x}]$
- $(P_1 \to P_2)[\frac{t}{x}] = P_1[\frac{t}{x}] \to P_2[\frac{t}{x}]$
- $(P_1 \leftrightarrow P_2)[\frac{t}{x}] = P_1[\frac{t}{x}] \leftrightarrow P_2[\frac{t}{x}]$
- $(\forall y\ P_1)[\frac{t}{x}] = \begin{cases} \forall y\ P_1[\frac{t}{x}] & \text{se } x \neq y \text{ e } y \notin \text{Fv}(t) \\ \forall z\ P_1[\frac{z}{y}][\frac{t}{x}] & \text{se } x \neq y, y \in \text{Fv}(t) \text{ e } z \text{ non occorre in } P_1, t \\ \forall y\ P_1 & \text{se } x = y \end{cases}$
- $(\exists y\ P_1)[\frac{t}{x}] = \begin{cases} \exists y\ P_1[\frac{t}{x}] & \text{se } x \neq y \text{ e } y \notin \text{Fv}(t) \\ \exists z\ P_1[\frac{z}{y}][\frac{t}{x}] & \text{se } x \neq y, y \in \text{Fv}(t) \text{ e } z \text{ non occorre in } P_1, t \\ \exists y\ P_1 & \text{se } x = y \end{cases}$

OSSERVAZIONI

- La sostituzione ha effetto solo se si applica alle variabili libere.
- Se nel termine t compare la variabile del quantificatore occorre cambiare nome a tale variabile in maniera opportuna. Questo è il motivo per il quale abbiamo bisogno di un insieme VAR infinito.

ESEMPI

- $\forall x(A(x) \to B(y))[\frac{f(y)}{y}] = \forall x(A(x) \to B(f(y)))$
- $\forall x(A(x) \to B(y))[\frac{f(x)}{y}] = \forall z(A(z) \to B(f(x)))$
- $\forall x(A(x) \to B(y))[\frac{c}{x}] = \forall x(A(x) \to B(y))$

ESERCIZI

Effettuare le seguenti sostituzioni:

- $(\forall x C(x,y) \vee A(y))[\frac{f(y)}{y}]$
- $(\forall x C(x,y) \vee A(y))[\frac{f(x)}{y}]$
- $(\exists x(A(y) \to B(x)))[\frac{g(x,y)}{y}]$
- $(\exists x(A(x) \vee B(y)))[\frac{h(x,y)}{x}]$
- $(\exists x(A(x) \vee B(y)) \to \forall y B(x))[\frac{c}{x}]$
- $(\exists x A(x) \vee B(y) \to \forall y B(y))[\frac{c}{y}]$
- $\forall x \exists y C(x,y)[\frac{z}{y}]$
- $\forall x \exists y C(x,z)[\frac{y}{z}]$

1.2.5 Semantica

Abbiamo già in parte accennato alla interpretazione semantica delle formule della logica dei predicati: diamo ora un assetto alle idee e ai concetti introdotti precedentemente.

Per interpretare una formula della logica del primo ordine per prima cosa dobbiamo specificare un *dominio* D, cioè l'insieme degli oggetti cui si riferiscono i termini che compaiono nella formula; secondo poi dobbiamo dare, mediante un *assegnamento*, un significato ai simboli di costante, di funzione e di predicato.

Un assegnamento prevede che:

- ad ogni simbolo di costante sia associato un elemento del dominio D;
- ad ogni simbolo di funzione f^k, con $k \geq 1$, sia associata una funzione da D^k in D;
- ad ogni simbolo di predicato B^k, con $k \geq 1$, sia associata una funzione che ad ogni elemento di D^k associa 1 se gli elementi della k-upla sono in relazione tra loro nella relazione B, 0 altrimenti.

Infine abbiamo bisogno di un *ambiente* nel quale ad ogni simbolo di variabile è associato un elemento del dominio.

Questi tre elementi (dominio, assegnamento e ambiente) costituiscono un'*interpretazione*. Un'interpretazione ci consente di attribuire un valore di verità ad una fbf.

Ad esempio sia P la fbf $\exists x\, \forall y\, A(x,y) \vee B(c)$. Consideriamo il dominio dei numeri naturali. Assegnamo a c il numero 2. Siano $A(x,y)$ il predicato "$x < y$ " e $B(x)$ il predicato "x è pari". La fbf è così interpretata: esiste un numero naturale che è strettamente minore di ogni altro numero naturale (questa sottoformula è falsa perché non è vero che $0 < 0$) oppure 2 è un numero pari. La fbf P risulta vera nell'interpretazione scelta. Si noti che assegnando a c il numero 3 otteniamo un'interpretazione che la rende falsa. Modifichiamo l'assegnamento relativo ai predicati: sia $A(x,y)$ il predicato "$x + y$ è un numero naturale" e $B(x)$ "x è un multiplo di 31". In questo caso la fbf P risulta vera comunque assegnamo un valore alla costante c.

Sia adesso P la fbf $\forall x(A(x) \vee \neg A(x))$. Tale fbf è sempre vera comunque scegliamo il dominio e l'assegnamento (non è necessario specificare l'ambiente perché non compaiono variabili libere). Infatti P è vera se, per ogni elemento del dominio, delle due una: vale A oppure non vale A. Quindi per ogni interpretazione la fbf P risulta vera: si dice che P è una *tautologia*.

Consideriamo P' la fbf $\forall x(A(x) \wedge \neg A(x))$. P' risulta falsa per ogni interpretazione. Infatti, per rendere vera P', per ogni elemento del dominio dovrebbero essere vere contemporaneamente la proprietà A e la proprietà $\neg A$. Si dice che P' è una *contraddizione*. Si osservi che nel caso di P e di P' è stato semplice capire che siamo in presenza di una tautologia e di una contraddizione. In generale, stabilire se una fbf Q sia o meno una tautologia, è un problema tutt'altro che banale in quanto il numero di possibili interpretazioni è infinito. Ricordiamo, per inciso, che nella logica proposizionale si può verificare se una fbf Q con n atomi è una tautologia, passando in rassegna tutte le interpretazioni (di numero pari a 2^n).

Vediamo ora come si definisce formalmente, in una data interpretazione $\mathcal{I}$, il valore di verità di una fbf P (denotato con $v_{\mathcal{I}}(P)$):

- $v_{\mathcal{I}}(\bot) = 0$, $v_{\mathcal{I}}(\top) = 1$
- $v_{\mathcal{I}}(A(t_1, \ldots, t_n)) = \begin{cases} 1 & \text{se gli elementi del dominio associati ai simboli } t_1, \ldots, t_n \text{ sono in relazione tra loro secondo } A \\ 0 & \text{altrimenti} \end{cases}$
- $v_{\mathcal{I}}(P_1 \wedge P_2) = \min\{v_{\mathcal{I}}(P_1), v_{\mathcal{I}}(P_2)\}$
- $v_{\mathcal{I}}(P_1 \vee P_2) = \max\{v_{\mathcal{I}}(P_1), v_{\mathcal{I}}(P_2)\}$
- $v_{\mathcal{I}}(P_1 \rightarrow P_2) = \begin{cases} 1 & \text{se } v_{\mathcal{I}}(P_1) \leq v_{\mathcal{I}}(P_2) \\ 0 & \text{altrimenti} \end{cases}$
- $v_{\mathcal{I}}(P_1 \leftrightarrow P_2) = \begin{cases} 1 & \text{se } v_{\mathcal{I}}(P_1) = v_{\mathcal{I}}(P_2) \\ 0 & \text{altrimenti} \end{cases}$

- $v_{\mathcal{I}}(\forall x\ P_1) = \begin{cases} 1 & \text{se comunque associamo alla variabile } x \text{ (libera in } P_1\text{) un elemento del dominio risulta } v_{\mathcal{I}}(P_1) = 1 \\ 0 & \text{altrimenti} \end{cases}$
- $v_{\mathcal{I}}(\exists x\ P_1) = \begin{cases} 1 & \text{se esiste almeno un elemento del dominio che associato alla variabile } x \text{ rende vero } P_1 \text{ (ovvero risulta } v_{\mathcal{I}}(P_1) = 1) \\ 0 & \text{altrimenti} \end{cases}$

ESERCIZI

Trovare un'interpretazione che renda vere le seguenti fbf nel dominio degli interi **Z**, nel dominio degli animali mammiferi (escluso l'essere umano), nel dominio delle forme geometriche piane(quadrato, rettangolo, pentagono, rombo, ecc.) e nel dominio dei parenti (familiari, parenti e affini).

- $\exists x\ A(x, y) \vee \forall y\ B(y, c)$
- $\forall x\ A(x) \rightarrow \exists z\ B(z)$
- $A(x) \wedge B(x) \leftrightarrow \bot$

1.2.6 Proprietà dei quantificatori

Diamo la definizione di equivalenza semantica, analogamente a quanto accade nella logica proposizionale: diremo che due fbf P e Q sono semanticamente equivalenti ($P \equiv Q$) se per tutte le interpretazioni $v_{\mathcal{I}}$ si ha che $v_{\mathcal{I}}(P) = v_{\mathcal{I}}(Q)$.

Esaminiamo ora alcune proprietà dei quantificatori che consentono di ottenere fbf semanticamente equivalenti.

Ridenominazione

- $\exists x P \equiv \exists z P[\frac{z}{x}]$
- $\forall x P \equiv \forall z P[\frac{z}{x}]$

Usiamo questa proprietà quando vogliamo cambiare nome alle variabili quantificate, per necessità, come ad esempio nel caso dei conflitti di nome nelle sostituzioni dei termini alle variabili libere, o per avere una notazione più familiare.

Esempio: $\exists m(A(m) \vee \forall m B(m)) \equiv \exists z(A(z) \vee \forall z\ B(z))$.

De Morgan

- $\neg\forall x P \equiv \exists x \neg P$
- $\neg\exists x P \equiv \forall x \neg P$

L'effetto della negazione di una formula quantificata può essere traslato sulla sottoformula interna a patto di cambiare il quantificatore. Ciò vuol dire che "non è vero che per ogni elemento del dominio vale la proprietà P" equivale a "esiste (almeno) un elemento del dominio che non soddisfa la proprietà P"; e, inoltre, "non è vero che esiste un elemento del dominio che soddisfa P" equivale a "tutti gli elementi del dominio non soddisfano P".

Esempio: $\neg\forall x\ (A(x) \vee B(y)) \equiv \exists x\ (\neg(A(x) \vee B(y))) \equiv \exists x\ (\neg A(x) \wedge \neg B(y))$

Doppio Gancio

- $\forall x\ P \equiv \neg\exists x\ \neg P$
- $\exists x\ P \equiv \neg\forall x\ \neg P$

Questa proprietà deriva da quella di De Morgan: "per ogni elemento del dominio vale la proprietà P" equivale a "non è vero che esiste un elemento del dominio che non soddisfa P" e "esiste un elemento del dominio che soddisfa P" equivale a "non è vero che ogni elemento del dominio non soddisfa P".

Esempio: $\neg\forall x\ (A(x) \rightarrow \neg B(x)) \equiv \neg\forall x\ (\neg A(x) \vee \neg B(x)) \equiv \neg\forall x\ (\neg(A(x) \wedge B(x)) \equiv \exists x\ (A(x) \wedge B(x))$.

Scambio

- $\forall x\ \forall y\ P \equiv \forall y\ \forall x\ P$
- $\exists x\ \exists y\ P \equiv \exists y\ \exists x\ P$

I quantificatori dello stesso tipo possono essere scambiati di posto se sono consecutivi. In questi casi, per comodità si può scrivere $\forall xy\ P$ al posto di $\forall x\ \forall y\ P$ (e di $\forall y\ \forall x\ P$) e $\exists xy\ P$ al posto di $\exists x\ \exists y\ P$ (e di $\exists y\ \exists x\ P$).

Lo scambio dei quantificatori non è lecito se essi non sono dello stesso tipo. Ad esempio, siano P e Q rispettivamente $\forall x\ \exists y\ B(x, y)$ e $\exists y\ \forall x\ B(x, y)$. Consideriamo il dominio dei naturali e sia B il predicato "$x < y$". La fbf P asserisce che "per ogni numero naturale ne esiste un altro che è strettamente maggiore"; questo fatto è vero, come si può verificare considerando per ogni numero il suo successore. D'altra parte la fbf Q dice che "esiste un numero che è strettamente minore di tutti i numeri naturali"; ma ciò non è vero: infatti 0 è strettamente minore di tutti i numeri naturali tranne che di se stesso.

Eliminazione dei quantificatori

- $\forall x P \equiv P$ se $x \notin \mathrm{FV}(P)$
- $\exists x P \equiv P$ se $x \notin \mathrm{FV}(P)$

Questa proprietà ci consente di eliminare i quantificatori inutili che non legano alcuna variabile. Nel caso di scritture del tipo $\forall y\ \exists x\ B(x)$, $\forall y\ \exists y\ B(y)$ e $\forall y\ B(z)$, che sono legittime, in quanto rispettano le regole sintattiche delle formule ben formate, possiamo eliminare il quantificatore $\forall$ ottenendo $\exists x\ B(x)$, $\exists y\ B(y)$ e $B(z)$, semanticamente equivalenti alle prime.

Distributività

- $\forall x(P_1 \wedge P_2) \equiv \forall x\ P_1 \wedge \forall x\ P_2$
- $\exists x(P_1 \vee P_2) \equiv \exists x\ P_1 \vee \exists x\ P_2$

Questa proprietà è diretta conseguenza del fatto che i quantificatori universale e esistenziale denotano rispettivamente una congiunzione infinita e una disgiunzione infinita di formule atomiche al variare degli elementi del dominio (infinito). Se per ogni elemento del dominio vale P_1 e per ogni elemento del dominio vale P_2 è naturale dedurre che per ogni elemento del dominio valgono P_1 e P_2. Analogamente, se esiste un elemento del dominio per il quale vale P_1 o P_2 allora ciò equivale a dire che esiste un elemento del dominio che soddisfa P_1 oppure esiste un elemento del dominio che soddisfa P_2.

Esempio 1: $\forall x\ (A(x) \wedge B(y)) \equiv \forall x\ A(x) \wedge \forall x\ B(y) \equiv \forall x\ A(x) \wedge B(y)$

Esempio 2: $\exists x\ (A(x) \rightarrow \neg B(x)) \equiv \exists x\ (\neg A(x) \vee \neg B(x)) \equiv \exists x\ \neg A(x) \vee \exists x\ \neg B(x) \equiv \forall x\ A(x) \vee \forall x\ B(x)$

ESERCIZI

Verificare l'esattezza delle seguenti scritture:

- $\forall z\ \exists y\ (A(x) \rightarrow A(x)) \equiv \top$
- $\neg(\neg\forall y\ (B(y, f(x)) \vee \exists y\ \neg A(y))) \equiv \forall y\ \exists z\ (B(y, f(x)) \wedge A(y))$
- $\neg(\neg\forall y\ (B(y, f(x))) \vee \exists y\ \neg A(y)) \equiv \forall y\ \exists z\ (B(y, f(x)) \wedge A(y))$
- $\forall x\ \exists y\ A(x, y) \leftrightarrow B(x) \equiv (\exists x\ \forall y\ \neg A(x, y) \vee B(x) \vee \bot) \wedge (\neg B(x) \vee \neg\exists x\ \forall y\ \neg A(x, y))$

Capitolo 2

Linguaggi Formali e Automi

2.1 Linguaggi formali

Un linguaggio formale è costituito da una collezione di sequenze di simboli appartenenti ad un insieme dato. Tali sequenze vengono considerate, in questo contesto, solo sotto l'aspetto sintattico (formale), senza fare riferimento alla semantica dei simboli. Allo scopo di dare una definizione più precisa di linguaggio formale, introduciamo le nozioni di alfabeto, stringa e concatenazione di stringhe.

Definizione 1. *Un alfabeto è un insieme finito non vuoto i cui elementi sono detti simboli.*

Esempi di alfabeti sono l'alfabeto delle lettere $\{a, b, c, .., z\}$, l'alfabeto delle cifre binarie $\{0, 1\}$, l'alfabeto del linguaggio Pascal (costituito dalle lettere, dai numeri, dalle parole chiave, come *begin*, *end*, e da altri simboli come le parentesi, il punto e virgola, i due punti, ecc.).

Definizione 2. *Una stringa sull'alfabeto A è una sequenza finita o nulla di simboli di A. La lunghezza di una stringa x, indicata con $|x|$, è il numero di simboli che la compongono: la sequenza nulla, indicata con ϵ, ha lunghezza zero.*

Le stringhe seguenti sono esempi di stringhe sull'alfabeto del linguaggio Pascal: "*begin;end;end;*", "*aa begin repeat until*", "*begin((*". Si noti che una parola chiave del linguaggio Pascal è un unico simbolo dell'alfabeto. Ad esempio la cardinalità della stringa "*begin((*"è 3.

Definizione 3. *La concatenazione di due stringhe è l'operazione che dà luogo alla stringa formata giustapponendo le stringhe.*

Consideriamo l'alfabeto $\{a, b, .., z\}$: se $x =$"verde" e $y =$"rame", allora $xy =$"verderame" e $yx =$"rameverde". La concatenazione non è commutativa. È, invece, associativa: $\forall x, y, z\ ((xy)z) = (x(yz))$. La stringa vuota ϵ è l'elemento neutro: $\forall x\ x\epsilon = \epsilon x = x$.

Definizione 4. *Sia A^* l'insieme di tutte le stringhe finite su A.*

Si noti che $A^* \neq \emptyset$ per qualunque alfabeto A, in quanto $\epsilon \in A^*$ (se $A = \emptyset$, $A^* = \{\epsilon\}$). Inoltre, A^* è chiuso rispetto alla concatenazione ovvero $\forall x, y \in A^*$ risulta $xy \in A^*$.

NOTAZIONE

Indicheremo, laddove si renderà necessario per una migliore leggibilità, l'operazione di concatenazione con il simbolo $\circ$.

Definizione 5. *Un linguaggio L su un alfabeto A è un sottoinsieme di A^*.*

Esempi di linguaggi sull'alfabeto $A = \{a, b, c\}$ sono:

- $\emptyset$ il linguaggio vuoto;
- $\{\epsilon\}$ il linguaggio che contiene solo la stringa vuota;
- $\{a, b, c\}$ il linguaggio contenente solo l'alfabeto A;
- $\{a, aa, aaa, \ldots\}$ il linguaggio contenente le stringhe a^n con $n \geq 1$;
- $\{ab, aab, aaab, \ldots, abb, aabbb, \ldots\}$ il linguaggio contenente le stringhe $a^n b^m$ con $n, m \geq 1$.

2.2 Operazioni sui linguaggi

I linguaggi vengono definiti a partire da un alfabeto A come sottoinsiemi di A^*. Pertanto, le usuali operazioni di unione, intersezione e complemento sono implicitamente definite:

1. dati due linguaggi L_1 e L_2, l'unione $L_1 \cup L_2$ è l'insieme $\{w \in A^* \mid w \in L_1 \text{ oppure } w \in L_2\}$;
2. dati due linguaggi L_1 e L_2, l'intersezione $L_1 \cap L_2$ è l'insieme $\{w \in A^* \mid w \in L_1 \text{ e } w \in L_2\}$;
3. il linguaggio $\bar{L}$ complementare di L è l'insieme: $\{w \in A^* \mid w \notin L\}$.

Oltre alle operazioni insiemistiche siamo interessati a due operazioni specifiche per i linguaggi: la concatenazione e la chiusura.

Dati due linguaggi L_1 e L_2 la concatenazione $L_1 \circ L_2$ è l'insieme $\{w \in A^* \mid w = xy, x \in L_1 \text{ e } y \in L_2\}$. Se uno tra L_1 e L_2 è l'insieme vuoto allora $L_1 \circ L_2 = \emptyset$. Se $L_1 = \{\epsilon\}$ allora $L_1 \circ L_2 = L_2$. Analogamente, se $L_2 = \{\epsilon\}$, $L_1 \circ L_2 = L_1$. La concatenazione di linguaggi è associativa ma non commutativa.

La chiusura di L, che si indica con L^*, è l'insieme:

$$L^* = \bigcup_{n \geq 0} L^n$$

dove $L^0 = \{\epsilon\}$ e $L^n = L \circ L^{n-1}$ per $n \geq 1$. La chiusura positiva di L, che si indica con L^+, è l'insieme:

$$L^+ = \bigcup_{n \geq 1} L^n.$$

Risulta:

$$L^+ = L^* \setminus \{\epsilon\} \qquad (L^* = L^+ \cup \{\epsilon\})$$
$$L^* = L \circ L^* = L^* \circ L$$

L'operatore * si chiama stella di Kleene. L^* si chiama chiusura di L in quanto ogni linguaggio ottenuto da L mediante unione e concatenazione è contenuto in L^*.

OSSERVAZIONE

A^*, l'insieme di tutte le stringhe sull'alfabeto A, coincide con la chiusura di A.

2.3 Rappresentazione di linguaggi

Abbiamo definito i linguaggi come sottoinsiemi di A^*, dove A è un alfabeto. Questa definizione, tuttavia, non ci permette di rappresentare in maniera effettiva un linguaggio, a meno che esso abbia un numero finito di elementi: in tal caso essi possono essere rappresentati in un elenco. Nella logica proposizionale e in quella del primo ordine, ad esempio, se ci fossimo limitati a definire i rispettivi linguaggi come sottoinsiemi delle stringhe sugli alfabeti dati, invece di fornire una regola di costruzione delle fbf, non avremmo potuto sviluppare alcun discorso. Abbiamo bisogno di meccanismi che ci consentano di avere una rappresentazione finita di linguaggi di caridinalità infinita. Grammatiche a struttura di frase e automi a stati finiti sono gli argomenti di cui ci occuperemo ora: le prime consentono di generare le frasi di un linguaggio, i secondi di riconoscerle.

2.4 Le grammatiche generative

Immaginiamo di trovarci nel paese Chissadove e di avere intenzione di comunicare pur non conoscendo la lingua. Immaginiamo di avere a disposizione un vocabolario speciale che contiene tutte le parole della lingua chissadovese (l'alfabeto, in chiave di linguaggi formali) senza la traduzione a fianco, con l'indicazione della categoria sintattica cui appartengono (verbo, sostantivo, ecc.). Ebbene, un modo per poter articolare frasi del linguaggio corrette dal punto di vista grammaticale, sebbene possano anche non avere senso compiuto, è quello di consultare un libro di grammatica e di osservarne le regole[1]. Questo è il metodo utilizzato dalle grammatiche generative per generare le frasi legali di un linguaggio.

Definizione 6. *Una grammatica a struttura di frase è una quadrupla $G = \langle A, N, P, S\rangle$ in cui:*

- *A è un insieme finito e non vuoto di simboli, detto alfabeto terminale, i cui elementi sono detti simboli terminali*
- *N è un insieme finito e non vuoto di simboli, detto alfabeto nonterminale, i cui elementi sono detti simboli nonterminali (o categorie sintattiche); denotiamo con V l'insieme $A \cup N$*

[1] Ricordiamo che in questo contesto ci interessa solo l'aspetto formale, ovvero grammaticale, del linguaggio; ciò significa che le frasi "la mela mangia il cane" o "2 è un numero dispari" sono corrette se rispettano le regole della grammatica, indipendentemente dal loro significato.

- *P, detto insieme delle produzioni, è una relazione binaria di cardinalità finita su $V^* \circ N \circ V^* \times V^*$ in cui tutte le coppie $\langle \alpha, \beta \rangle \in P$ presentano in α almeno un simbolo nonterminale*
- *$S \in N$ è detto assioma ed è il simbolo nonterminale iniziale, ossia la categoria sintattica più generale.*

NOTAZIONE

Una coppia $\langle \alpha, \beta \rangle \in P$ si indica generalmente con la notazione $\alpha \to \beta$. Una regola del tipo $\alpha \to \epsilon$ prende il nome di ϵ-produzione. Un insieme di regole di produzione del tipo $\alpha \to \beta_1$, $\alpha \to \beta_2$,...,$\alpha \to \beta_n$ viene convenzionalmente indicato con $\alpha \to \beta_1 | \beta_2 | \ldots | \beta_n$.

Introduciamo ora due relazioni fra le stringhe su V.

Definizione 7. *Una stringa $\psi \in V^*$ deriva direttamente da $\phi \in V^*$ (denotato con $\phi \Rightarrow \psi$) se esistono $\alpha \in V^* \circ N \circ V^*$ e $\beta, \gamma, \delta \in V^*$ tali che $\phi = \gamma\alpha\delta$, $\psi = \gamma\beta\delta$ e $\alpha \to \beta \in P$.*

ESEMPIO

Supponiamo che la stringa ϕ sia "La mamma VERBO tutti", dove "VERBO" è un simbolo nonterminale e "La mamma"e "tutti" sono simboli terminali. Se P contiene la produzione VERBO $\to$ promuove, allora da ϕ deriva direttamente la stringa "La mamma promuove tutti".

Definizione 8. *Una stringa $\psi \in V^*$ deriva da $\phi \in V^*$ in un numero finito di passi (denotato con $\phi \overset{*}{\Rightarrow} \psi$) se esiste una sequenza (detta derivazione) $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ per $n \geq 0$ e $\phi = \alpha_1$, $\psi = \alpha_n$.*

Si noti che $\phi \overset{*}{\Rightarrow} \psi$ per $n = 0$ vuol dire che $\phi = \psi$ e quindi per ogni stringa ϕ si ha $\phi \overset{*}{\Rightarrow} \phi$

Definizione 9. *Un linguaggio L generato da una grammatica G (denotato con $L(G)$) è l'insieme delle stringhe sull'alfabeto A (stringhe di terminali) che possono essere derivate da S in un numero finito di passi. In simboli: $L(G) = \{w \in A^* \mid S \overset{*}{\Rightarrow} w\}$.*

ESEMPI

Sia $G = \langle \{a, b\}, \{S, B, C\}, P, S \rangle$, il cui insieme P è composto dalle seguenti regole di produzione:

$$\begin{array}{rcl} S & \to & aS \mid B \\ B & \to & bB \mid bC \\ C & \to & cC \mid c \end{array}$$

Allora $L(G) = \{a^n b^m c^h \mid n \geq 0,\ m, h \geq 1\}$. La generica stringa $a^n b^m c^h$ si ottiene applicando:

- $n \geq 0$ volte la produzione $S \to aS$:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow \ldots \Rightarrow a^n S$$

- la produzione $S \to B$:

$$a^n S \Rightarrow a^n B$$

- $m - 1$ volte la produzione $B \to bB$:

$$a^n B \Rightarrow a^n bB \Rightarrow a^n bbB \Rightarrow \ldots \Rightarrow a^n b^{m-1} B$$

- la produzione $B \to bC$:

$$a^n b^{m-1} B \Rightarrow a^n b^m C$$

- $h - 1$ volte la produzione $C \to cC$:

$$a^n b^m C \Rightarrow a^n b^m cC \Rightarrow \ldots \Rightarrow a^n b^m c^{h-1} C$$

- la produzione $C \to c$:

$$a^n b^m c^{h-1} C \Rightarrow a^n b^m c^h$$

Sia $G = \langle \{a, b\}, \{S, A\}, P, S \rangle$, il cui insieme P è composto dalle seguenti regole di produzione:

$$\begin{array}{rcl} S & \to & aAb \\ aA & \to & aaAb \\ A & \to & \varepsilon \end{array}$$

Allora $L(G) = \{a^n b^n \mid n \geq 1\}$. La generica stringa $a^n b^n$ si ottiene applicando:

- una volta la produzione $S \to aAb$;
- $n - 1$ volte la produzione $aA \to aaAb$:

$$aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow \ldots \Rightarrow a^n Ab^n$$

- una volta la produzione $A \to \epsilon$:

$$a^n Ab^n \Rightarrow a^n b^n$$

2.5 Gli automi a stati finiti

Torniamo per un momento nel paese Chissadove; questa volta, meno fortunati, non abbiamo a disposizione una grammatica: possiamo utilizzare solo il vocabolario speciale contenente l'alfabeto del linguaggio. Come fare per poter comporre le frasi legali della lingua chissadovese? Un modo potrebbe essere quello di costruire frasi a caso, utilizzando i termini dell'alfabeto, e chiedere ad un cittadino chissadovese di ascoltarle. Dall'espressione del suo volto si potrebbe

capire se la frase è legale oppure no. In maniera analoga un linguaggio formale può essere definito come l'insieme delle stringhe riconosciute da un accettore a stati finiti.

L'accettore a stati finiti è una specializzazione del concetto di automa a stati finiti. Per prima cosa, dunque, diamo la definizione generale di automa a stati finiti. Un automa a stati finiti (AF) è un modello che rappresenta sistemi che transitano da uno stato a un altro per effetto di un qualche input. Per avere un'immagine più concreta, possiamo pensare che un AF sia un dispositivo dotato di uno stato interno e di un canale attraverso cui riceve simboli dall'esterno: a seconda del simbolo letto e dello stato corrente, il dispositivo modifica il suo stato interno. Un AF può essere rappresentato tramite un grafo etichettato: i nodi costituiscono gli stati e gli archi, le cui etichette rappresentano l'input, costituiscono le transizioni da uno stato ad un altro.

ESEMPIO

L'AF rappresentato in Figura 2.1 descrive il comportamento di un interruttore.

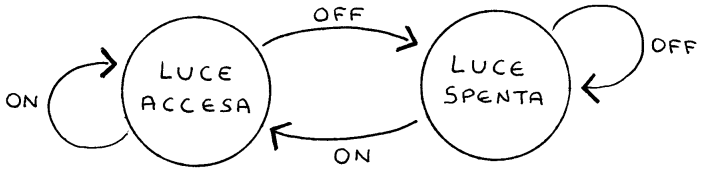


Figura 2.1: AF che descrive un interruttore

Definizione 10. *Un AF è una tripla* $\langle Q, A, \delta \rangle$ *dove:*

- Q *è un insieme finito di stati*
- A *è un alfabeto*
- $\delta : Q \times A \mapsto Q$ *è la funzione di transizione, eventualmente parziale, che associa ad una coppia stato-simbolo un nuovo stato.*

Nell'esempio precedente abbiamo:

- $Q = \{q_0, q_1 \mid q_0 = \text{LUCE ACCESA},\ q_1 = \text{LUCE SPENTA}\}$
- $A = \{\text{ON, OFF}\}$
- $\delta = \{\langle q_0, \text{ON}\rangle \mapsto q_0, \langle q_0, \text{OFF}\rangle \mapsto q_1, \langle q_1, \text{ON}\rangle \mapsto q_0, \langle q_1, \text{OFF}\rangle \mapsto q_1\}$

La funzione $\delta^* : Q \times A^* \mapsto Q$ viene ottenuta estendendo δ al dominio $Q \times A^*$ allo scopo di definire il comportamento dell'AF quando riceve in input una stringa di A^*. δ^* è definito

come segue:

$$\delta^*(q, \epsilon) = q$$
$$\delta^*(q, xi) = \delta(\delta^*(q, x), i) \qquad x \in A^*, i \in A$$

Ad esempio, nel caso dell'interruttore, la funzione δ^* con la stringa "OFFOFF" in ingresso porta, partendo dallo stato LUCE ACCESA, l'AF nello stato LUCE SPENTA. Infatti:

δ^*(LUCE ACCESA, OFFOFF) = $\delta(\delta^*$(LUCE ACCESA, OFF), OFF) =
$\delta(\delta(\delta^*$(LUCE ACCESA, $\epsilon)$, OFF), OFF) = $\delta(\delta$(LUCE ACCESA, OFF), OFF) =
δ(LUCE SPENTA, OFF) = LUCE SPENTA

ESERCIZI

- Costruire un AF che descriva la seguente situazione: Tizio e Caio tirano una moneta; se esce testa vince Tizio, altrimenti vince Caio.
- Tizio è un tipo un po' particolare: risponde alle domande di Caio solo se questi le formula in una certa maniera. Innanzitutto se Caio non comincia col dire "salve", Tizio ignora la domanda. Se Caio dice "salve" e continua con "per favore", allora Tizio è disposto ad ascoltare la domanda, altrimenti no. Se Tizio ascolta la domanda, fornisce la risposta. Descrivere il comportamento di Tizio mediante un AF.

Operiamo una differenziazione degli stati: consideriamo uno stato iniziale q_0 e un insieme di stati finali F. Diamo ora la definizione di una classe di automi a stati finiti chiamati riconoscitori o accettori che per comodità continueremo a indicare con AF.

Definizione 11. *Un accettore a stati finiti (AF) è una quintupla* $\langle Q, A, \delta, q_0, F\rangle$ *dove:*

- Q *è un insieme finito di stati*
- A *è un alfabeto*
- $\delta : Q \times A \mapsto Q$ *è la funzione di transizione, eventualmente parziale, definita (ed estesa a una funzione* $\delta^* : Q \times A^* \mapsto Q$*) come nel caso degli automi a stati finiti*
- $q_0 \in Q$ *è lo stato iniziale*
- $F \subseteq Q$ *è l'insieme degli stati finali*

Una stringa $x \in A^*$ *è accettata (riconosciuta) da un AF se e solo se* $\delta^*(q_0, x) \in F$ *ovvero se e solo se l'automa, partendo dallo stato iniziale, transita in uno stato finale per effetto dell'input* x*.*

Definizione 12. *Il linguaggio $L(\mathcal{A})$ riconosciuto (o accettato) da un accettore a stati finiti $\mathcal{A}$ è l'insieme delle stringhe accettate da $\mathcal{A}$. In simboli:* $L(\mathcal{A}) = \{w \in A^* \mid \delta^*(q_0, w) \in F\}$.

OSSERVAZIONE

Come fatto in precedenza, possiamo dare un'immagine concreta di un accettore pensandolo come un dispositivo dotato di una testina di lettura che legge l'input scritto su un nastro di ingresso e di un controllo che governa i cambiamenti di stato (vedi Figura 2.2).

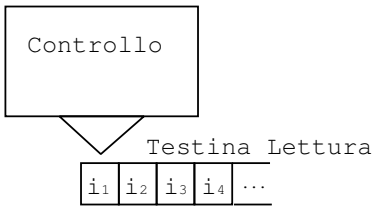


Figura 2.2: rappresentazione "operazionale" di un accettore

In principio la testina di lettura è posizionata all'inizio della stringa di ingresso scritta sul nastro; q_o è lo stato iniziale del dispositivo. Ad ogni passo la testina legge un carattere i e si sposta a destra; il dispositivo transita allo stato $\delta(q, i)$, dove q è lo stato corrente ed i è il simbolo letto. Se $\delta(q, i)$ è indefinita, la macchina si ferma. Se la stringa di ingresso viene completamente letta e l'automa, dopo aver letto l'ultimo simbolo, si trova in uno stato finale, la stringa viene accettata, altrimenti viene rifiutata.

ESEMPIO

L'automa in Figura 2.3 riconosce il linguaggio $L = \{a^{2n}b^{3m}c \mid n \geq 1, m \geq 0\}$. Una stringa di L consiste di un numero pari di a, maggiore di zero, seguita da un numero di b multiplo di 3, seguito a sua volta da un'unica c.

NOTA

Lo stato iniziale q_0 è indicato con una freccia. Lo stato finale q_5 con un doppio cerchio.

ESERCIZI

Si costruiscano gli automi che riconoscano i seguenti linguaggi:

- $L_1 = \{0, 1\}^* \circ \{00\} \circ \{0, 1\}^*$
- $L_2 = \{a^n b^m \mid n \geq 1, m \geq 1\}$
- $L_3 = L_1 \circ L_2$
- $L_4 = \{a^n \mid n \geq 0\}$

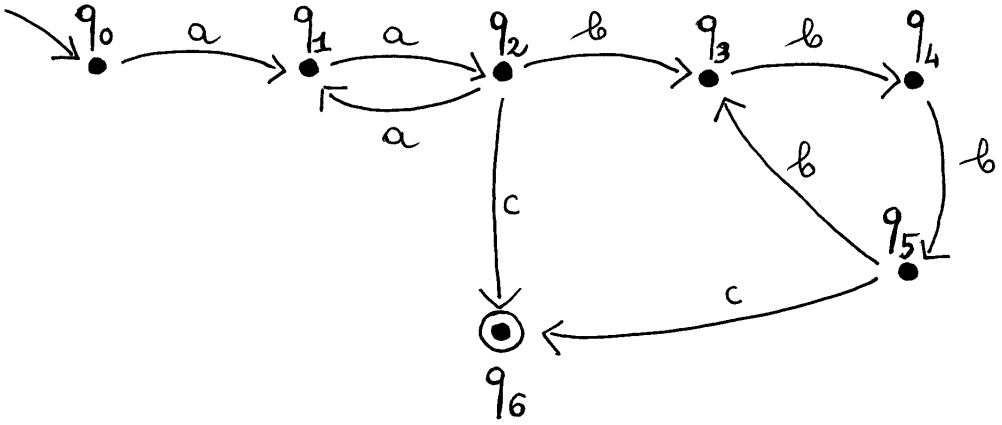


Figura 2.3: automa che riconosce $L = \{a^{2n}b^{3m}c \mid n \geq 1, m \geq 0\}$

- $L_5 = \{abc, acb, abbb, c\}$
- $L_6 = \{(ab)^n ccc \mid n \geq 0\}$
- $L_7 = \{x \in \{a, b, c\}^* \mid |x| = 3\}$

2.6 Relazione tra grammatiche a struttura di frase e automi a stati finiti

Abbiamo detto che una grammatica a struttura di frase G è in grado di generare un linguaggio $L(G)$. Un automa a stati finiti $\mathcal{A}$ permette di riconoscere tutte le stringhe del linguaggio $L(\mathcal{A})$. Che relazione c'è tra linguaggi generati e linguaggi riconosciuti? Data una grammatica G è sempre possibile costruire un automa $\mathcal{A}$ tale che $L(G) = L(\mathcal{A})$? Dato un automa $\mathcal{A}$ è sempre possibile trovare una grammatica G tale che $L(G) = L(\mathcal{A})$? Per rispondere a queste domande osserviamo che le grammatiche a struttura di frase si differenziano a seconda delle restrizioni applicate alle produzioni.

Definizione 13. *Sia $G = \langle A, N, P, S\rangle$ una grammatica a struttura di frase. Se per ogni produzione $\alpha \to \beta$ in P, risulta:*

- $|\alpha| = 1$ *(ciò implica $\alpha \in N$)*
- *β è ϵ oppure β è della forma aB o della forma a, con $B \in N, a \in A$*

allora G viene detta grammatica regolare.

Automi a stati finiti e grammatiche regolari sono modelli equivalenti: cioè data una grammatica regolare G è possibile costruire un automa $\mathcal{A}$ tale che $L(G) = L(\mathcal{A})$ e viceversa.

Tuttavia, in generale, le grammatiche a struttura di frase costituiscono un modello più potente di quello degli automi a stati finiti, nel senso che possono generare linguaggi per i quali non sarà mai possibile costruire l'automa riconoscitore.

Consideriamo ad esempio il linguaggio $L = \{a^n b^n \mid n \geq 1\}$ che come abbiamo visto è generato dalla grammatica $G = (\{a, b\}, \{A, S\}, P, S)$ con P dato da:

$$S \to aAb$$
$$aA \to aaAb$$
$$A \to \epsilon$$

Osserviamo che G non è una grammatica regolare perché le prime due produzioni non rispettano i vincoli prescritti dalla definizione.

Per verificare che non esiste alcun automa $\mathcal{A}$ in grado di riconoscere L dobbiamo fare alcune osservazioni. Dato un linguaggio L, riconosciuto da un automa $\mathcal{A}$, $L(\mathcal{A})$ ha cardinalità infinita (finita) se e solo se il grafo di transizione di $\mathcal{A}$ presenta (non presenta) cicli. Infatti, il ciclo può essere percorso un numero arbitrario di volte consentendo all'automa di riconoscere una quantità illimitata di stringhe. Inoltre il ciclo parte da uno stato e termina nel medesimo stato: il "riconoscimento parziale"(chiamiamolo così) delle sottostringhe relative al ciclo non muta lo stato dell'automa. Vediamo un esempio.

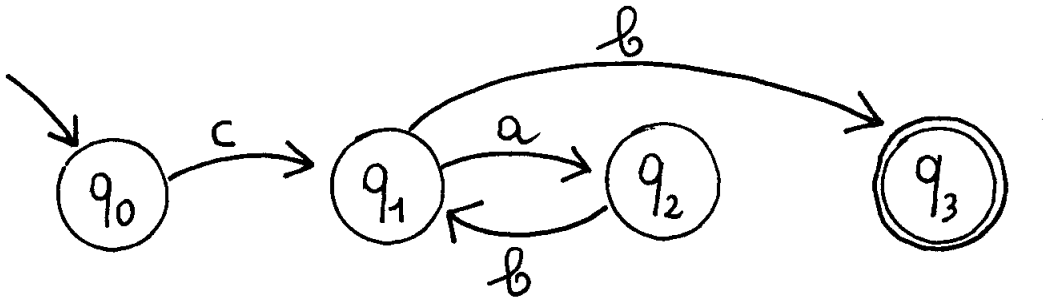


Figura 2.4: automa che riconosce $L = \{c(ab)^n b | n \geq 0\}$

Il linguaggio riconosciuto dall'automa descritto dal grafo di transizione in Figura 2.4 è $L = \{c(ab)^n b | n \geq 0\}$. Il ciclo genera una quantità illimitata di sottostringhe $(ab)^n$ e l'automa, dopo aver riconosciuto ciascuna di tali sottostringhe, si trova nello stato q_1.

Torniamo al linguaggio $L = \{a^n b^n | n \geq 1\}$. Supponiamo per assurdo che esista un automa $\mathcal{A}$ tale che $L(\mathcal{A}) = L$. $\mathcal{A}$ contiene sicuramente un ciclo che comincia e termina in un certo stato q e che genera le sottostringhe a^n con $n \geq 1$. Inoltre, partendo da q, l'automa riconosce le sottostringhe b^n con $n \geq 1$ e termina in uno stato finale. Ebbene, consideriamo la stringa $a^h b^k$ con $h \geq 1$, $k \geq 1$ e $h \neq k$. L'automa $\mathcal{A}$ (che riconosce le stringhe $a^n b^n$ con $n \geq 1$) è in grado di riconscere la stringa $a^h b^k$? La risposta è si. Infatti il ciclo riconosce le sottostringhe

a^n per qualsiasi valore $n \geq 1$, e quindi riconosce anche la sottostringa a^h portando l'automa in uno stato q. Sappiamo che l'automa, a partire dallo stato q, è in grado di riconoscere le sottostringhe b^n con $n \geq 1$ transitando in uno stato finale, e quindi riconosce la sottostringa b^k. Dunque $\mathcal{A}$ riconosce $a^h b^k$ con $h \neq k$. Ma allora $\mathcal{A}$ accetta stringhe non appartenenti a L; pertanto $\mathcal{A}$ non è l'automa riconoscitore di L.

2.7 Relazione tra linguaggi e insiemi di numeri

Un linguaggio L su un alfabeto A è un sottoinsieme di A^*. Possiamo associare ad ogni stringa di A^* un numero naturale: di conseguenza L può essere considerato come un insieme di numeri naturali. Vediamo come realizzare tale corrispondenza. Sia n la cardinalità di A e fissiamo una biiezione fra A e l'insieme $\{1, 2, \ldots, n\}$; ad ogni simbolo $a_i \in A$ associamo il numero c_i. Ad una stringa $x = a_k a_{k-1} \ldots a_1 a_0$ è associato il numero

$$m = c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \ldots + c_1 \cdot n^1 + c_0 \cdot n^0$$

Viceversa, ad ogni numero m risulta associata la stringa $x = a_k a_{k-1} \ldots a_1 a_0$ dove a_i è il simbolo associato al $(k-i)$-esimo resto della divisione di m per n. Ad esempio, consideriamo il codice ASCII costituito da 128 caratteri. La lettera "a" corrisponde al numero 65, "b" a 66, "c" a 67 e così via. Sfruttando questa corrispondenza associamo alla stringa *casa* il numero $67 \cdot 128^3 + 65 \cdot 128^2 + 83 \cdot 128^1 + 65 \cdot 128^0$ (cioè 141.584.833); al numero 1.403.713 corrisponde la stringa *avu* (infatti $1.403.713 = 128 \cdot ((128 \cdot 85) + 86) + 65$; 85, 86, 65 sono i resti della divisione di 1.403.713 per 128, cui corrispondono rispettivamente i caratteri "u", "v", "a").

OSSERVAZIONE

La corrispondenza tra linguaggi e insiemi di numeri può essere effettuata in vari modi. Quella che abbiamo fornito è solo un esempio.

Capitolo 3

Cardinalità degli Insiemi e Algoritmi

3.1 Cardinalità degli Insiemi

3.1.1 Insiemi Finiti e Infiniti

Sia A un insieme. La cardinalità di A, indicata con $|A|$, è il numero di elementi che compongono A. Se A è un insieme finito, la cardinalità di A è denotata da un numero naturale n. Se, invece, A è un insieme infinito, la sua cardinalità viene detta *transfinita* e viene indicata con $\aleph_k$ con $k \in \mathbf{N}$.

Gli insiemi infiniti differiscono da quelli finiti per caratteristiche e proprietà che a volte contrastano col senso comune. Ad esempio, sia A un insieme e B un suo sottoinsieme proprio. Risulta che $|B| \leq |A|$. Nel caso di insiemi finiti vale $|B| < |A|$, in conformità alla nozione comune di Euclide, secondo cui "il tutto è maggiore della parte". Quando si parla di insiemi che contengono un numero infinito di elementi può accadere che $|B| = |A|$: in realtà, questo fatto non è casuale, anzi costituisce una caratterizzazione degli insiemi infiniti, vale a dire che un insieme A è infinito se esiste un suo sottoinsieme proprio B tale che $|B| = |A|$.

La tecnica che ci permette di contare gli elementi di un insieme (sia esso finito o infinito) è la corrispondenza biunivoca. Diremo che due insiemi A e B sono equinumerosi se esiste una biiezione tra essi. Ad esempio, gli insiemi {gennaio, febbraio,..., dicembre} e $\{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23\}$ sono equinumerosi. Dato un insieme finito A, la sua cardinalità è così definita:

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

Dato un insieme infinito A diremo che A è numerabile, oppure che A ha la potenza del numerabile, se esso è equinumeroso a $\mathbf{N}$. La cardinalità degli insiemi infiniti numerabili è indicata con $\aleph_0$. Gli insiemi infiniti non hanno tutti la stessa cardinalità ci sono diversi livelli di infinito indicati con $\aleph_0$, $\aleph_1$, $\aleph_2$, ... e così via. Tra questi prenderemo in considerazione il livello $\aleph_1$, successivo ad $\aleph_0$, detto infinito non numerabile o potenza del continuo.

3.1.2 Insiemi Numerabili

È facile convincersi che ogni insieme infinito A include un sottoinsieme infinito numerabile. Infatti è possibile sottrarre da A un primo elemento, un secondo, e così via, ed associare all'i-esimo elemento sottratto il numero naturale i. Tale procedimento non termina perché, per ipotesi, A è infinito. È altrettanto intuitivo che ogni sottoinsieme infinito di un insieme infinito numerabile è anch'esso numerabile. Un insieme numerabile non può contenere un insieme infinito di cardinalità maggiore di $\aleph_0$. Diamo alcuni esempi di insiemi numerabili.

- L'insieme $\mathbf{Z}$ degli interi è numerabile. Infatti, la biiezione $f : \mathbf{Z} \mapsto \mathbf{N}$ può essere definita come segue:
$$f(i) = \begin{cases} -2i & \text{se } i \leq 0 \\ 2i - 1 & \text{se } i > 0 \end{cases}$$

- L'insieme P dei numeri pari è numerabile.

  Infatti, P è un insieme infinito, sottoinsieme proprio di un insieme numerabile: pertanto, P è numerabile.

- L'insieme $\mathbf{Q}$ dei numeri razionali è numerabile. Per dimostrare ciò utilizziamo la *tecnica di enumerazione di Cantor*, chiamata anche coda di colomba. Si può applicare tale tecnica a tutti gli insiemi $X = \{x_{ij} \mid i, j \geq 0\}$, cioè tali che i loro elementi possono essere elencati in una tabella infinita bidimensionale come la seguente:

	0	1	2	$\cdots$	j	$\cdots$
0	x_{00}	x_{01}	x_{02}	$\cdots$	x_{0j}	$\cdots$
1	x_{10}	x_{11}	x_{12}	$\cdots$	x_{1j}	$\cdots$
2	x_{20}	x_{21}	x_{22}	$\cdots$	x_{2j}	$\cdots$
$\vdots$	$\vdots$				$\vdots$	
i	x_{i0}	x_{i1}	x_{i2}	$\cdots$	x_{ij}	$\cdots$
$\vdots$	$\vdots$				$\vdots$	

  La biiezione $f : X \mapsto \mathbf{N}$ è definita nel seguente modo:

	0		1		2		$\cdots$
0	0		1		3		6
		$\swarrow$		$\swarrow$		$\swarrow$	
1	2		4		$\cdots$		
		$\swarrow$		$\swarrow$			
2	5		$\cdots$				

  Più formalmente all'elemento x_{ij} viene associato il numero $\frac{(i+j)(i+j+1)}{2} + i$. Nel caso dell'insieme $\mathbf{Q}$ dei numeri razionali si procede come segue: ogni numero razionale è individuato da una coppia di interi (p, q) con $p \in \mathbf{Z}$ e $q \in \mathbf{Z}^+$ ($\mathbf{Z}^+$ è l'insieme dei numeri interi maggiori di 0); c'è una biiezione tra $\mathbf{N}$ e l'insieme $\mathbf{Z} \times \mathbf{Z}^+$ come risulta in

	1		2		3		4	$\cdots$
0	$(0,1)$	$\swarrow$	$(0,2)$	$\swarrow$	$(0,3)$	$\swarrow$	$(0,4)$	
1	$(1,1)$	$\swarrow$	$(1,2)$	$\swarrow$	$(1,3)$			
2	$(2,1)$	$\swarrow$	$(2,2)$					
3	$(3,1)$							
$\vdots$	$\vdots$							

tabella. Nella tabella 3.1.2, al posto dell'insieme **Z** compare l'immagine di **Z** attraverso la funzione:

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

Tale immagine è **N**, ed è necessario usarla al fine di rappresentare l'insieme dei numeri interi su una semiretta infinita (invece che su una retta infinita) per poter applicare la tecnica a coda di colomba. Pertanto **Q** è numerabile.

Osserviamo che **N**, **Z** e **Q** hanno la stessa potenza di infinito, vale a dire che sono equinumerosi. Tuttavia, tra due numeri naturali (o due interi) non ci sono numeri naturali (interi). Ci sono però infiniti numeri razionali. Tra due numeri razionali c'è ancora un'infinità di numeri razionali. Eppure i numeri razionali sono tanti quanti i numeri naturali (interi). Questo fatto può essere alquanto sorprendente se non si tiene a mente che le regole e le proprietà degli insiemi infiniti sono diverse da quelle proprie degli insiemi finiti.

ESERCIZIO 7.

Utilizzando la tecnica a coda di colomba, si dimostra che:

- il prodotto cartesiano di due insiemi numerabili è numerabile;
- l'unione di una quantità numerabile di insiemi numerabili è ancora un insieme numerabile.

- Sia A un alfabeto. Consideriamo A^*, l'insieme di tutte le stringhe di lunghezza arbitraria costruibili con i simboli di A. A^* è un insieme numerabile. Infatti, è possibile ordinare le stringhe di A^* in base alla lunghezza e, in caso di pari lunghezza, seguendo un ordinamento lessicografico (dato ai simboli dell'alfabeto). All'elemento che occupa il posto i-esimo in tale ordinamento, viene associato il numero naturale i. Ad esempio, sia $A = \{0, 1\}$. Adottiamo l'ordinamento naturale sui simboli di

A secondo il quale 0 precede 1. L'ordinamento sulle stringhe di A^* è il seguente: 0,1,00,01,10,11,000,001,010...

3.1.3 Insiemi non Numerabili

Il livello di infinito successivo al numerabile è il continuo. La potenza del continuo, indicata con $\aleph_1$, è la cardinalità dei numeri reali. Annunciamo che, nelle dimostrazioni che seguono, per provare che un insieme A è non numerabile, utilizzeremo la *tecnica di diagonalizzazione* di Cantor che consiste nel supporre A numerabile, elencandone gli elementi, e nel costruire un elemento (detto diagonale) in modo tale che differisca da tutti gli altri elementi.

- L'insieme $\mathbf{R}$ dei numeri reali (razionali più irrazionali) è non numerabile. Abbiamo visto che l'insieme dei numeri razionali (numeri decimali che hanno una sequenza di cifre dopo la virgola finita oppure, se infinita, periodica) è numerabile. L'insieme dei numeri irrazionali (numeri decimali con una sequenza di cifre dopo la virgola infinita e aperiodica) è non numerabile.

  Per dimostrare la non numerabilità di $\mathbf{R}$ si utilizza la tecnica di diagonalizzazione di Cantor. Cominciamo col dimostrare che l'insieme aperto $(0,1)$ è un insieme continuo. Infatti, se $(0,1)$ fosse numerabile potremmo enumerare i numeri decimali come d_0,d_1,...e rappresentare ognuno di essi come la sequenza $d_i = 0, d_{i0}d_{i1}d_{i2}\ldots$ È possibile allora costruire il numero decimale "diagonale" $\widehat{d} = 0, \widehat{d_0}\widehat{d_1}\ldots$ dove $\widehat{d_i} = (d_{ii} + 1) \mod 10$.

	0	1	2	$\cdots$	i	$\cdots$
d_0	d_{00}	d_{01}	d_{02}	$\cdots$	d_{0i}	$\cdots$
d_1	d_{10}	d_{11}	d_{12}	$\cdots$	d_{1i}	$\cdots$
d_2	d_{20}	d_{21}	d_{22}	$\cdots$	d_{2i}	$\cdots$
$\vdots$	$\vdots$				$\vdots$	
d_i	d_{i0}	d_{i1}	d_{i2}	$\cdots$	d_{ii}	$\cdots$
$\vdots$	$\vdots$				$\vdots$	

  Il numero decimale $\widehat{d}$ appartiene a $(0,1)$ ma non fa parte della numerazione, il che falsifica l'ipotesi di numerabilità di $(0,1)$. La biiezione $f : \mathbf{R} \mapsto (0,1)$ definita come $f(x) = \frac{1}{2^x+1}$ dimostra che $\mathbf{R}$ è equinumeroso a $(0,1)$, cioè che $\mathbf{R}$ è un insieme continuo.

- Diamo un altro esempio di insieme non numerabile. L'insieme delle parti 2^A di un insieme numerabile A non è numerabile. Osserviamo che gli elementi di 2^A sono in corrispondenza biunivoca con le sequenze di 0 e 1 di lunghezza infinita. Consideriamo, a titolo dimostrativo, l'insieme $\mathbf{N}$. La stringa 01010101... rappresenta il sottoinsieme P dei numeri pari. La stringa 01100000... rappresenta il sottoinsieme $\{2,3\}$. La stringa 11111111... rappresenta $\mathbf{N}$. Dimostriamo adesso, utilizzando nuovamente la tecnica di diagonalizzazione, che l'insieme B delle stringhe infinite di 0 e di 1 non è numerabile. Se B fosse numerabile potremmo rappresentare le stringhe b_i nella seguente tabella:

	0	1	2	$\cdots$	i	$\cdots$
b_0	b_{00}	b_{01}	b_{02}	$\cdots$	b_{0i}	$\cdots$
b_1	b_{10}	b_{11}	b_{12}	$\cdots$	b_{1i}	$\cdots$
b_2	b_{20}	b_{21}	b_{22}	$\cdots$	b_{2i}	$\cdots$
$\vdots$	$\vdots$				$\vdots$	
b_i	b_{i0}	b_{i1}	b_{i2}	$\cdots$	b_{ii}	$\cdots$
$\vdots$	$\vdots$				$\vdots$	

Consideriamo la stringa $b = b_{00}b_{11}b_{22}\ldots$ costituita dagli elementi della diagonale e facciamone il complemento:

$$\overline{b} = \overline{b_{00}}\,\overline{b_{11}}\,\overline{b_{22}}\ldots \text{ dove } \overline{b_{ii}} = 1 - b_{ii}$$

$\overline{b}$ appartiene a B (è una sequenza di 0 e 1), ma non compare nell'elenco: infatti differisce per costruzione con qualsiasi sequenza b_i perché al posto della cifra b_{ii} presenta $\overline{b_{ii}}$. In conclusione l'insieme delle parti di un insieme numerabile è non numerabile.

- Dimostriamo che l'insieme F delle funzioni dai naturali ai naturali è un insieme continuo. Consideriamo F_b l'insieme delle funzioni binarie che associano ad ogni naturale un elemento di $\{0, 1\}$. F_b è un sottoinsieme proprio di F ed è un insieme non numerabile. Infatti, se tale insieme fosse numerabile potrebbe essere rappresentato in tabella come segue dove F_i rappresenta l'i-esima funzione binaria :

	0	1	2	$\cdots$	i	$\cdots$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$\cdots$	$f_0(i)$	$\cdots$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$\cdots$	$f_1(i)$	$\cdots$
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$\cdots$	$f_2(i)$	$\cdots$
$\vdots$	$\vdots$				$\vdots$	
f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	$\cdots$	$f_i(i)$	$\cdots$
$\vdots$	$\vdots$				$\vdots$	

Applicando la tecnica diagonale si dimostra che la funzione $\overline{f} : \mathbf{N} \mapsto \mathbf{N}$ definita come

$$\overline{f}(x) = \{0 \to \overline{f_0(0)}, 1 \to \overline{f_1(1)}, 2 \to \overline{f_2(2)}, \ldots\}$$

non compare nell'elenco.

F ha cardinalità maggiore o uguale a quella di F_b. D'altra parte essendo ogni funzione da $\mathbf{N}$ in $\mathbf{N}$ un sottoinsieme di $\mathbf{N}^2$ risulta $|F| \leq |2^{\mathbf{N}^2}|$. La cardinalità di $2^{\mathbf{N}^2}$ è $\aleph_1$. Segue che F ha la potenza del continuo.

3.2 Algoritmi e funzioni calcolabili

3.2.1 Il concetto di algoritmo

Intuitivamente, per algoritmo si intende la procedura di risoluzione di un problema mediante un dispositivo automatico di calcolo. Il concetto di algoritmo è essenzialmente informale: tuttavia si possono enunciare diverse proprietà che costituiscono i requisiti di ogni definizione di algoritmo:

- un algoritmo deve contenere una sequenza finita di istruzioni.
- deve esistere un agente di calcolo capace di eseguire le istruzioni.
- deve esserci un limite finito al numero e alla complessità delle istruzioni eseguibili dall'agente di calcolo. Ogni istruzione deve, cioè essere eseguita in una quantità di tempo limitata.
- l'agente di calcolo deve avere una memoria dove immagazzinare i risultati intermedi. Non deve esserci alcun limite finito alla quantità di memoria disponibile, poichè al crescere della dimensione dei dati in ingresso cresce anche la quantità di memoria necessaria per memorizzare i dati intermedi.
- la computazione è discreta, ossia l'informazione è codificata in forma digitale e la computazione procede per passi discreti.
- non deve esistere un limite finito sui dati in ingresso e di uscita.
- il tipo di calcolo che consideriamo è quello *deterministico* in cui ogni passo di calcolo viene determinato esclusivamente e in maniera univoca in base all'esito dei passi eseguiti in precedenza. Tratteremo, inoltre, il calcolo *non deterministico* in cui ad ogni passo l'agente di calcolo può scegliere tra diverse vie, ognuna delle quali può essere seguita indifferentemente per effettuare il calcolo. Il calcolo non è *probabilistico*, cioè alla modalità di esecuzione del calcolo non è associata nessuna legge di probabilità. Nel calcolo probabilistico ognuna di queste vie è seguita in base a una certa misura di probabilità.
- Non esiste un limite sul numero di passi discreti richiesti per effettuare un calcolo.
- Sono ammesse esecuzioni con un numero di passi infinito (cioè esecuzioni che non terminano mai).

3.2.2 Non-esistenza di algoritmi per tutte le funzioni

Sia S un qualunque modello di calcolo (linguaggio di programmazione, macchina RAM, macchina di Turing, grammatica formale, automa a stati finiti, ...).

Caratteristica comune a tutti i modelli di calcolo è quella di avere un linguaggio, tramite il quale descrivere gli algoritmi, e un meccanismo che ne consenta l' esecuzione. Ricordiamo che un linguaggio (formale) è un sottoinsieme di A^* dove A è un alfabeto finito.

Ad ogni algoritmo $\mathcal{A}$ possiamo associare una funzione $f_{\mathcal{A}}$ avente come dominio di definizione l'insieme dei possibili risultati in ingresso e come codominio l'insieme dei possibili risultati finali. Tale funzione può essere totale, se $\mathcal{A}$ restituisce un output per ogni possibile input, parziale se, per qualche input, l'esecuzione di $\mathcal{A}$ non termina. L'input e l'output possono essere codificati tramite numeri naturali, pertanto un algoritmo $\mathcal{A}$ rappresenta una funzione $f_{\mathcal{A}} : \mathbf{N} \mapsto \mathbf{N}$. La funzione $f_{\mathcal{A}}$ è la funzione *calcolata* da $\mathcal{A}$. Si noti che ad ogni algoritmo $\mathcal{A}$ è associata un'unica funzione $f_{\mathcal{A}}$ (in generale parziale), mentre la stessa funzione f può essere associata a più algoritmi. Verifichiamo tale affermazione e osserviamo che una funzione f calcolata da $\mathcal{A}$, $f_{\mathcal{A}}$ è uguale alla funzione $f'_{\mathcal{A}}$ in cui $\mathcal{A}'$ indica l'algoritmo $\mathcal{A}$ così modificato: si effettua l'inserimento dell'istruzione *null* in un punto qualsiasi dell'algoritmo. $\mathcal{A}$ e $\mathcal{A}'$ sono algoritmi diversi che calcolano la stessa funzione. Si noti che questo procedimento può essere iterato un numero arbitrario di volte (modificando $\mathcal{A}$ inserendo di volta in volta un numero arbitrario di istruzioni *null*) sicché risulta che un numero arbitrario di algoritmi (un numero infinito) calcolino la stessa funzione.

Diremo che una funzione f è calcolabile attraverso un modello di calcolo S se esiste un algoritmo $\mathcal{A}$ in S tale che $f = f_{\mathcal{A}}$. Sia F_S l'insieme delle funzioni calcolabili in S. La cardinalità di F_S dipende dal sistema formale S adottato: tuttavia, comunque scegliamo S, l'insieme F_S delle funzioni calcolabili in S avrà al più una quantità numerabile di elementi, in quanto ogni algoritmo è descritto da una stringa di W^*, dove W è l'alfabeto di S, e W^* è un insieme numerabile: la cardinalità di F_S è strettamente minore della cardinalità dell'insieme F contenente tutte le funzioni dai naturali ai naturali (ricordiamo che l'insieme F ha cardinalità infinita non numerabile). Questo vuol dire che ci sono funzioni alle quali non è possibile associare alcun algoritmo; esistono cioè funzioni non calcolabili. Possiamo inoltre affermare che la "maggior parte" delle funzioni non è calcolabile: le funzioni calcolabili costituiscono un "piccolissimo" sottoinsieme delle funzioni dai naturali ai naturali, allo stesso modo in cui i numeri naturali sono un "piccolissimo" sottoinsieme dei numeri reali.

Capitolo 4

Elementi di Teoria della Calcolabilità

4.1 La macchina di Turing

Per indagare la nozione di calcolabilità abbiamo bisogno di fissare un *modello di calcolo*, ovvero un formalismo nel quale descrivere gli algoritmi. Tra le molte possibilità, scegliamo la *macchina di Turing*, un modello che ha il vantaggio di poter essere pensato come un dispositivo fisico, reale, cosa che può facilitarne notevolmente la comprensione.

4.1.1 Descrizione modellistica

Una macchina di Turing (MT) può essere vista come un dispositivo dotato di un controllo a stati finiti, un nastro di lunghezza infinita e una testina di lettura/scrittura. Il nastro è suddiviso in caselle (o celle), ognuna delle quali contiene un simbolo appartenente a un alfabeto $\Sigma = \{\#, a_1, \ldots, a_n\}$: $\#$ denota assenza di scrittura e viene chiamato blanc. Si suppone sempre che il nastro sia tutto blanc tranne una parte di lunghezza finita. La testina è posizionata su una delle caselle del nastro e consente di leggere e scrivere un simbolo in tale casella. La MT esegue ripetutamente (fino a che, come specificheremo tra breve, non si arresta) *passi computazionali*, a seguito dei quali lo stato e il contenuto del nastro vengono modificati. Più esattamente, un passo computazionale è costituito dal seguente insieme di operazioni: la MT legge il simbolo che si trova nella casella su cui è posizionata la testina e, in risposta al simbolo letto e allo stato del controllo, scrive nella casella un nuovo simbolo (eventualmente uguale a quello letto), sposta la testina di una casella a destra o a sinistra oppure ne lascia immutata la posizione, transita in un nuovo stato (eventualmente uguale a quello precedente).

La situazione in cui si trova una MT dopo l'esecuzione di una serie di passi computazionali, viene descritta da una *configurazione istantanea* (o, più semplicemente configurazione) che è l'insieme delle informazioni costituito da:

- il contenuto del nastro
- la posizione della testina sul nastro

- lo stato corrente

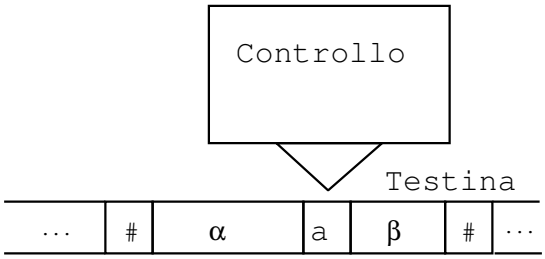


Figura 4.1: Macchina di Turing

Ad ogni passo computazionale, la MT transita da una configurazione istantanea ad un'altra. Per computazione di una MT intendiamo una sequenza finita di passi computazionali. Equivalentemente, possiamo considerare una computazione come una sequenza di configurazioni istantanee ognuna delle quali è ottenuta dalla precedente in un passo computazionale.

4.1.2 Descrizione matematica

Definizione 14. *Una macchina di Turing $\mathcal{M}$ è una quadrupla $\mathcal{M} = \langle \Sigma, Q, q_0, \delta \rangle$ dove:*

- $\Sigma = \{\#, a_1, \ldots, a_n\}$ *è un alfabeto;*
- Q *è un insieme finito e non vuoto di stati;*
- $q_0 \in Q$ *è lo stato iniziale*
- δ *è una funzione, detta di transizione, definita come* $\delta : Q \times \Sigma \mapsto Q \cup F \times \Sigma \times \{D, S, I\}$ *in cui* $F = \{H, Y, N\}$ *è l'insieme degli stati finali e* $\{D, S, I\}$ *indicano, rispettivamente, lo spostamento a destra, lo spostamento a sinistra e l'immobilità (assenza di spostamento) della testina (*$H, Y, N \notin Q$ *e* $D, S, I \notin \Sigma$*)*

Nel seguito, guarderemo spesso alla funzione δ come ad un insieme di coppie ingresso-uscita. Nel caso di δ, l'ingresso è a sua volta una coppia e l'uscita è una tripla. Per non appesantire la notazione con n-uple annidate, con un leggero abuso di notazione considereremo la funzione δ come l'insieme delle *quintuple* (q, a_j, q', a_l, X) tali che $q \in Q$, $q' \in Q \cup F$, $a_j, a_l \in \Sigma$, $X \in \{D, S, I\}$ e $\delta(q, a_j) = (q', a_l, X)$: cioè la MT, quando si trova nello stato q_i e legge il simbolo a_j, transita nello stato q_{ij}, sostituisce il simbolo a_j con il simbolo a_{ij} e modifica la posizione della testina a seconda del simbolo x_{ij}. Naturalmente, non ci possono essere due quintuple aventi i primi due simboli uguali (questo perchè δ è una funzione): ne deriva che il modo di operare della MT è deterministico.

Per rappresentare in modo compatto una *configurazione istantanea*, useremo una stringa del tipo $\alpha q a \beta$ in cui:

- $\alpha a \beta$ è il contenuto del nastro ($\alpha, \beta \in \Sigma^*$, $a \in \Sigma$);
- la testina è posizionata sul simbolo a;
- $q \in Q \cup F$ è lo stato corrente.

La conoscenza di una configurazione c, unita a quella della funzione di transizione δ della MT, consente di determinare la configurazione successiva c'. Introduciamo la relazione di derivazione $\vdash$ fra configurazioni ($c \vdash c'$) che vale quando c' è ottenuta da c in un passo computazionale.

Definizione 15. *Sia $\mathcal{M} = \langle \Sigma, Q, q_0, F, \delta \rangle$ una MT. La* relazione di derivazione $\vdash$ *è definita come segue:*

- $\alpha q a \beta \vdash \alpha\ a' q' \beta$ *se* $\delta(q, a) = (q', a', D)$
- $\alpha q a \beta \vdash \alpha\ q' a' \beta$ *se* $\delta(q, a) = (q', a', I)$
- $\alpha \bar{a} q a \beta \vdash \alpha\ q' \bar{a} a' \beta$ *se* $\delta(q, a) = (q', a', S)$

Scriviamo $c \vdash^* c'$ per dire che c' è ottenuta da c in un numero finito (eventualmente nullo) di passi computazionali.

Un configurazione si dice *iniziale* se $c = q_0 \beta$ con $\beta \in (\Sigma \setminus \{\#\})^*$, ovvero la MT si trova nello stato iniziale q_0 e la testina è posizionata sul primo carattere della stringa di input. Si osservi che viene richiesto che all'inizio sul nastro ci sia una stringa che non contiene #. Se non imponessimo questa restrizione, non saremmo in grado di capire qual è l'ultimo carattere della stringa in ingresso e non potremmo mai essere certi di averla letta tutta.

Una configurazione si dice *finale* se $c = \alpha q \beta$ con $q \in F$. Segue dalla definizione di δ che, a partire da una configurazione finale, una MT non può transitare in alcuna altra configurazione. Quando una MT transita in una configurazione finale diciamo che essa *si arresta*.

ESERCIZIO 8. Dimostrare che se c è una configurazione finale, non esiste una configurazione c' tale che $c \vdash c'$.

Definiamo, inoltre, *computazione* di una MT $\mathcal{M} = \langle \Sigma, Q, q_0, F, \delta \rangle$ una sequenza (non necessariamente finita) di configurazioni $c_0, c_1, \ldots, c_j, \ldots$ dove c_0 è una configurazione iniziale e $c_i \vdash c_{i+1}$ per ogni $i \geq 0$. Se una computazione $c_0, c_1, \ldots$ è infinita, diciamo che la MT *non si arresta* a partire dalla configurazione c_0, o anche, poiché $c_0 = q_0 \beta$, che la MT non si arresta con la stringa β in ingresso. Banalmente, se una computazione è infinita, non contiene configurazioni finali e diciamo che essa *non termina*.

4.1.3 Rappresentazione della funzione di transizione

La funzione di transizione δ di una MT $\mathcal{M} = \langle \Sigma, Q, q_0, \delta \rangle$ può essere descritta mediante una tabella, detta *matrice di transizione*, che ne raggruppa le quintuple.

$Q \times \Sigma$	$\#$	a_1	$\cdots$	a_j	$\cdots a_n$	
q_0	$\delta(q_0, \#)$	$\delta(q_0, a_1)$	$\cdots$	$\delta(q_0, a_j)$	$\cdots$	$\delta(q_0, a_m)$
$\vdots$				$\vdots$		
q_i	$\delta(q_i, \#)$	$\delta(q_i, a_1)$	$\cdots$	$\delta(q_i, a_j)$	$\cdots$	$\delta(q_i, a_m)$
$\vdots$				$\vdots$		
q_m	$\delta(q_m, \#)$	$\delta(q_m, a_1)$	$\cdots$	$\delta(q_m, a_j)$	$\cdots$	$\delta(q_m, a_m)$

Un'altra rappresentazione è costituita dai *grafi di transizione*. I nodi del grafo costituiscono gli stati in Q. Esiste un arco (q_i, q_{ij}) cui è associata l'etichetta (a_i, a_{ij}, x_{ij}) se e solo se $\delta(q_i, a_j) = (q_{ij}, a_{ij}, x_{ij})$.

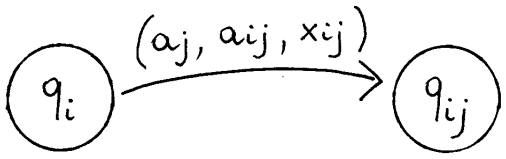


Figura 4.2: Rappresentazione di una transizione

Per non appesantire troppo esempi ed esercizi eviteremo di specificare δ per ogni possibile coppia di valori in ingresso adottando la seguente convenzione: per ogni coppia $(q, a) \in Q \times \Sigma$, se $\delta(q, a)$ non è specificata, si assume che $\delta(q, a) = (H, a, I)$, ovvero la MT transita nello stato finale H senza modificare il nastro né muovere la testina.

Consideriamo la funzione di transizione δ descritta nella seguente matrice:

	a_1	a_2	$\#$
q_0	$q_0 a_1 D$		$H \# I$
q_1			

Il corrispondente grafo di transizione è rappresentato in Figura 4.3.

NOTAZIONE

Indichiamo con una freccia lo stato iniziale e con un doppio cerchio gli stati finali.

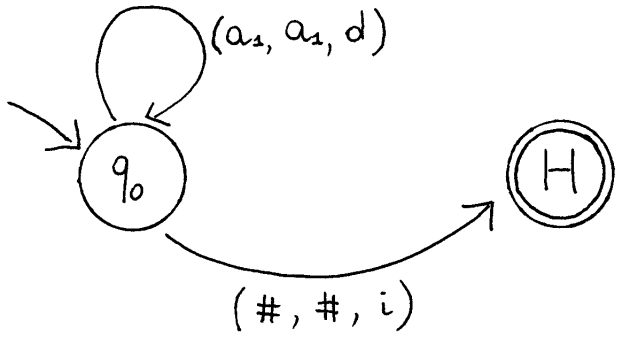


Figura 4.3: Grafo di transizione

4.1.4 Esempi di macchine di Turing

CALCOLO DEL SUCCESSIVO DI UN NUMERO

Il procedimento di calcolo del numero successivo di $c_1c_2\ldots c_k \in \mathbf{N}$ può essere schematizzato come segue:

- si esamina l'ultima cifra c_k del numero;
- se $c_k < 9$ si incrementa c_k di 1 e ci si ferma;
- se $c_k = 9$ si sostituisce con 0 e si passa ad esaminare la cifra a sinistra.

La configurazione iniziale è costituita:

- dal numero che si vuole incrementare scritto sul nastro;
- dalla testina posizionata sulla prima cifra;
- dal controllo nello stato iniziale q_0.

$\mathcal{M} = \langle \Sigma, Q, q_0, \delta \rangle$, dove:

$$\Sigma = \{\#, 0, 1, \ldots, 9\},\ Q = \{q_0, q_1\}$$

e δ è definita come segue:

- spostamento della testina a destra:

$$\begin{aligned} \delta(q_0, a) &= (q_0, a, D) \quad \forall a \in \Sigma \setminus \{\#\} \\ \delta(q_0, \#) &= (q_1, \#, S) \end{aligned}$$

- incremento:

$$\begin{aligned} \delta(q_1, 0) &= (H, 1, I) \\ &\dots \\ \delta(q_1, 8) &= (H, 9, I) \\ \delta(q_1, 9) &= (q_1, 0, S) \\ \delta(q_1, \#) &= (H, 1, I) \end{aligned}$$

CALCOLO DEL NUMERO DI OCCORRENZE DI 1 IN UNA SEQUENZA BINARIA

Il procedimento di calcolo può essere schematizzato come segue:

- Si usa il simbolo ausiliario \$ per denotare l'inizio della sequenza binaria;
- la sequenza binaria viene letta da sinistra verso destra;
- ogni volta che si incontra un 1:
  - si cambia l'1 in 0;
  - si scorre verso sinistra il nastro fino ad incontrare il simbolo \$;
  - si incrementa di 1 il numero alla sinistra del \$;
  - si scorre nuovamente il nastro verso destra fino alla lettura di 1 o #;
- il procedimento termina con la lettura di # e la cancellazione dell'input residuo.

La configurazione iniziale è costituita:

- dalla sequenza binaria di cui si vuole contare il numero di occorrenze di 1;
- dalla testina posizionata sulla prima cifra binaria;
- dal controllo nello stato iniziale q_0.

$\mathcal{M} = \langle \Sigma, Q, q_0, \delta \rangle$, dove:

- $\Sigma = \{\#, 0, 1, \dots, 9, \$\}$
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- δ è definita come segue.
- inserimento del simbolo \$

$$\begin{aligned} \delta(q_0, 0) &= (q_0, 0, S) \\ \delta(q_0, 1) &= (q_0, 1, S) \\ \delta(q_0, \#) &= (q_1, \$, D) \end{aligned}$$

- verifica occorrenza di 1:

$$\begin{aligned}\delta(q_1, 0) &= (q_1, 0, D)\\ \delta(q_1, \#) &= (q_5, \#, S)\\ \delta(q_1, 1) &= (q_2, 0, S)\end{aligned}$$

- spostamento della testina a sinistra:

$$\begin{aligned}\delta(q_2, 0) &= (q_2, 0, S)\\ \delta(q_2, \$) &= (q_3, \$, S)\end{aligned}$$

- incremento del contatore:

$$\begin{aligned}\delta(q_3, \#) &= (q_4, 1, D)\\ \delta(q_3, 1) &= (q_4, 2, D)\\ &\dots\\ \delta(q_3, 8) &= (q_4, 9, D)\\ \delta(q_3, 9) &= (q_3, 0, S)\end{aligned}$$

- spostamento della testina a destra:

$$\begin{aligned}\delta(q_4, a) &= (q_4, a, D) \quad \forall a \in \Sigma \setminus \{\$\}\\ \delta(q_4, \$) &= (q_1, \$, D)\end{aligned}$$

- cancellazione dell'input:

$$\begin{aligned}\delta(q_5, 0) &= (q_5, \#, S)\\ \delta(q_5, \$) &= (H, \#, I)\end{aligned}$$

4.1.5 Funzioni calcolabili mediante macchine di Turing

Definizione 16. *Sia* Π *un alfabeto tale che* $\# \notin \Pi$ *e* $g : \Pi^* \mapsto \Pi^*$ *una funzione sull'insieme delle stringhe su* Π. *Si dice che una MT* $\mathcal{M} = \langle \Sigma, Q, q_0, \delta \rangle$, *con* $\Sigma = \Pi \cup \{\#\}$, *calcola* g *se* $\forall x \in \Pi^*$*:*

- *se* $g(x) = y$ *allora* $q_0 x \vdash^* \alpha q \beta$, *con* $q \in F$ *e* $\alpha\beta = y$;
- *se* $g(x)$ *è non è definita allora* $\mathcal{M}$ *non si arresta con ingresso* x.

Ad ogni MT $\mathcal{M}$ può essere associata una funzione $h : \mathbf{N} \mapsto \mathbf{N}$ una volta fissate delle convenzioni consistenti per la codifica dei dati iniziali e del risultato finale. Ciò può essere fatto in molti modi. Consideriamo un esempio.

Definiamo una codifica dei numeri naturali in termini delle configurazioni iniziali (C_I) della MT tramite la funzione $i : \mathbf{N} \mapsto C_I$ così definita: ad ogni numero naturale n corrisponde una configurazione iniziale $q_0\alpha$ in cui α è una sequenza $aaa\ldots a$, di cardinalità n, scelto a un qualunque simbolo di Σ diverso da #. Ad esempio $i(2)$ è $\ldots\#q_0aa\#\ldots$. Anche se non tutte le configurazioni iniziali sono codifica di qualche input, ciò non importa. Invece, ad ogni naturale deve naturalmente corrispondere un'unica descrizione iniziale.

Le MT possono essere utilizzate anche per calcolare funzioni sui numeri naturali, una volta fissate delle convenzioni consistenti per la codifica dei dati iniziali e del risultato finale. Ciò può essere fatto in molti modi. Consideriamo un esempio.

Definiamo una decodifica tra le configurazioni finali (C_F) e i numeri naturali mediante una funzione $f : C_F \mapsto \mathbf{N}$ definita come segue: ad ogni configurazione finale $\alpha q\beta$ corrisponde il numero di simboli a consecutivi a destra della testina, compreso il simbolo puntato dalla testina, scelto a un qualunque simbolo di Σ diverso da #. Ad esempio $f(\ldots\#qaaa\#b\#)$ è uguale a 3, mentre $f(\ldots\#q\#a\#)$ è uguale a 0.

La funzione $h : \mathbf{N} \mapsto \mathbf{N}$ calcolata da una MT è la composizione delle funzioni i, g, e f: $h(n) = f(g(i(n)))$.

Diremo che una funzione $h : \mathbf{N} \mapsto \mathbf{N}$ è Turing calcolabile (T-calcolabile) se esiste una MT che la calcola.

La nozione di funzione T-calcolabile può essere estesa ad un qualunque numero di argomenti.

Definizione 17. *Sia* $g : (\Pi^*)^k \mapsto \Pi^*$ $(k \geq 1)$. *Una MT* $\mathcal{M} = \langle \Sigma, Q, q_0, \delta \rangle$ *calcola* g *se* $\forall x_1, \ldots, x_k \in \Sigma^*$ *accade che* $g(x_1, \ldots, x_k) = y$ *implica* $q_0x_1\#x_2\#\ldots\#x_k \vdash^* \alpha q\beta$ *con* $q \in F$ *e* $\alpha\beta = y$.

Ad ogni MT $\mathcal{M}$ può essere associata una funzione $h : \mathbf{N}^k \mapsto \mathbf{N}$ tale che $h = f \circ g \circ i$ dove g è la funzione della definizione precedente, e f e i sono definite come segue:

- $i : \mathbf{N}^k \mapsto C_I$ con $i(x_1, \ldots, x_k) = \#q_0 \underbrace{a\ldots a}_{x_1\text{volte}}\#\underbrace{a\ldots a}_{x_2\text{volte}}\#\ldots\#\underbrace{a\ldots a}_{x_n\text{volte}}\#\#$
- $f : C_F \mapsto \mathbf{N}$ con $f(\alpha q\beta)$= numero di a consecutive a destra della testina compreso il simbolo letto dalla testina.

Definizione 18. *Diremo che una funzione* $h : \mathbf{N}^k \mapsto \mathbf{N}$ *è T-calcolabile se esiste una MT che la calcola.*

OSSERVAZIONE

In seguito faremo riferimento alle funzioni T-calcolabili come a funzioni f scalari monoargomentali ($f : \mathbf{N} \mapsto \mathbf{N}$). Le funzioni $f : \mathbf{N}^k \mapsto \mathbf{N}$ (come pure le funzioni $f : \mathbf{N}^k \mapsto \mathbf{N}^h$) sono contemplate nel caso delle funzioni $f : \mathbf{N} \mapsto \mathbf{N}$. Infatti possiamo codificare una k-upla di numeri naturali mediante un solo numero naturale e, viceversa, dato un numero naturale possiamo associargli una h-upla di numeri naturali.

Il metodo mediante il quale realizziamo questa corrispondenza si basa sull'unicità della fattorizzazione di un numero in numeri primi. Data una n-upla $(x_1, \ldots, x_n)$ associamo ad

essa il numero $x = 2^{x_1} \cdot 3^{x_2} \cdot \ldots \cdot p_n^{x_n}$ dove p_n è l'n-esimo numero primo. Viceversa ad un numero x, la cui scomposizione in numeri primi è $2^{x_1} \cdot 3^{x_2} \cdot \ldots \cdot p_n^{x_n}$, associamo la n-upla $(x_1, \ldots, x_n)$.

4.1.6 Macchina di Turing multinastro

Le MT che abbiamo definito nelle sezioni precedenti ci permettono di calcolare una classe di funzioni che abbiamo chiamato T-calcolabili. Sorge spontaneo chiedersi se sia possibile in qualche modo estendere la classe delle funzioni calcolabili modificando la definizione di MT. Sono state studiate numerose varianti di MT. Ne elenchiamo alcune:

- MT con nastri multipli (multinastro)
- MT con testine multiple
- MT con nastri semi-infiniti (cioè con un limite in una direzione)
- MT con alfabeto di cardinalità due
- MT con due soli stati (non finali)
- MT con regole diverse nell'esecuzione del passo computazionale.

Tutte queste varianti della MT "standard", risultano computazionalmente equivalenti ad essa, ovvero per ogni MT "non-standard" si può costruire una MT "standard" che calcola la stessa funzione. Quindi l'insieme delle funzioni calcolate assumendo come modello una qualsiasi variante di MT, è sempre l'insieme delle funzioni T-calcolabili.

Consideriamo in dettaglio una di queste varianti: la macchina di Turing multinastro (MTM). La possibilità di disporre di più nastri potrebbe sembrare una variante non da poco. Ed in effetti, i vantaggi ci sono: gli algoritmi realizzati utilizzando una MTM risultano più efficienti e notevolmente più semplici da scrivere rispetto a quelli di una MT a nastro singolo. Ma come ribadiremo nel seguito, dal punto di vista del potere computazionale del modello, le MTM sono equivalenti alle MT.

Definizione 19. *Una MT a k nastri ($k \geq 2$) è una quadrupla $\mathcal{M}^k = \langle \Sigma, Q, q_0, \delta^k \rangle$ in cui Σ contiene il simbolo speciale Z_0 (oltre al solito #), e la funzione di transizione δ^k è definita come $\delta^k : Q \times \Sigma^k \mapsto (Q \cup F) \times \Sigma^k \times \{S, D, I\}^k$.*

La macchina multinastro esegue una transizione a partire da uno stato interno q e con le k testine (una per nastro) posizionate sui caratteri $a_1, \ldots, a_k$. Se $\delta^k(q, a_1, \ldots, a_k) = (p, b_1, \ldots, b_k, x_1, \ldots, x_k)$ allora la MTM:

- si porta nello stato p;
- sostituisce i caratteri $a_1, \ldots, a_k$ rispettivamente con $b_1, \ldots, b_k$;
- fa compiere alle testine i rispettivi spostamenti (a destra, a sinistra o nessuno spostamento), come specificato dagli $x_i \in \{D, S, I\}, i = 1, \ldots, k$.

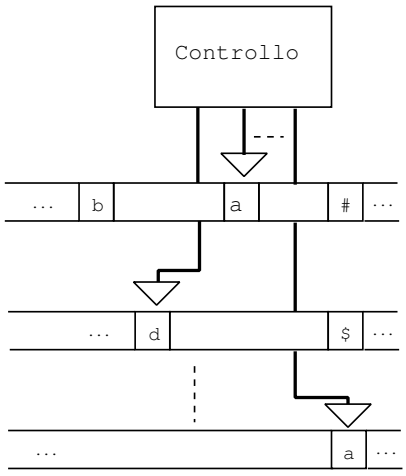


Figura 4.4: Macchina di Turing Multinastro

Definizione 20. *Una configurazione istantanea di una MTM $\mathcal{M}^k$ è una stringa del tipo $q\$\alpha_1 \downarrow \beta_1\$\alpha_2 \downarrow \beta_2\$ \ldots \$\alpha_k \downarrow \beta_k$ in cui α_i, $\beta_i \in \Sigma^*$ per $i = 1, \ldots, k$ essendo $\downarrow$ il simbolo che indica la posizione di ogni testina, \$ un separatore che marca l'inizio di ogni stringa $\alpha_i\beta_i$ ($\downarrow, \$ \notin \Sigma$).*

All'inizio di una computazione l'input è contenuto sul primo nastro e tutti gli altri nastri contengono solo un simbolo iniziale $Z_0 \in \Sigma$.

- Una configurazione istantanea si dice iniziale se $\alpha_i = \epsilon$ per $i = 1, \ldots, k$, $\beta_1 \in \Sigma^*$, $\beta_i = Z_0$ per $i = 2, \ldots, k$ e $q = q_0$.
- Una configurazione iniziale è una stringa del tipo $q_0\$ \downarrow \beta_1\$ \downarrow Z_0\$ \ldots \$ \downarrow Z_0$.
- Una configurazione istantanea di una MTM si dice finale se $q \in F$.

Sia Π un alfabeto tale che $\#, Z_0 \notin \Pi$ e $f : \Pi^* \mapsto \Pi^*$ una funzione sull'insieme delle stringhe su Π. Una MTM $\mathcal{M}^k = \langle \Sigma, Q, q_0, \delta^k \rangle$, con $\Sigma = \Pi \cup \{\#, Z_0\}$ calcola f se $\forall x \in \Pi^*$ accade che $f(x) = y$ implica $q_0\$ \downarrow x\$ \downarrow Z_0\$ \ldots \$ \downarrow Z_0 \vdash^* q\$\alpha_1 \downarrow \beta_1\$\alpha_2 \downarrow \beta_2\$ \ldots \$\alpha_k \downarrow \beta_k$ dove $q \in F$ e $\alpha_2\beta_2 = y$, mentre $f(x)$ indefinita implica che $\mathcal{M}$ non si arresta con ingresso x. Si osservi che, per comodità utilizziamo il secondo nastro come nastro di uscita della MT.

La funzione $f : \mathbf{N} \mapsto \mathbf{N}$ calcolata da una macchina di Turing multinastro è definita come nel caso della macchina a nastro singolo.

È immediato riconoscere che una MT è un caso particolare di MTM, e che pertanto ogni funzione T-calcolabile può essere calcolata da una MTM. Ciò può far pensare che la MTM sia un dispositivo di calcolo "più potente" di una MT. Benché la possibilità di usare più di un nastro consenta di progettare MT più efficienti (cioè che necessitano di meno passi per portare a termine il calcolo) e renda molto più agevole la realizzazione di un algoritmo attraverso una MT, in realtà tale possibilità non rende una MTM "più potente" di una MT,

perlomeno se attribuiamo a "più potente" il consueto significato di "capace di calcolare più funzioni". Infatti, si può dimostrare che per ogni MTM a k nastri esiste e si può costruire una MT equivalente, ovvero una MT che calcola la stessa funzione. Dunque l'insieme delle funzioni calcolabili dalle MTM coincide con l'insieme delle funzioni T-calcolabili.

ESEMPIO

Realizziamo una MTM a 2 nastri che duplica una stringa, ovvero che implementa la funzione $f : \Pi^* \mapsto \Pi^*$ definita come $f(\omega) = \omega\omega$ per ogni $\omega \in \Pi^*$.

Poiché abbiamo convenuto di utilizzare il secondo nastro di una MTM per rappresentare il risultato delle computazioni, il nostro compito è copiare due volte la stringa di ingresso sul secondo nastro. Dunque la MT per prima cosa copia una prima volta l'input sul secondo nastro:

$$\begin{aligned} \delta(q_0, x, Z_0) &= (q_1, x, x, D, D) \quad \forall x \in \Sigma \\ \delta(q_1, x, \#) &= (q_1, x, x, D, D) \quad \forall x \in \Sigma \setminus \{\ \#\} \\ \delta(q_1, \#, \#) &= (q_2, \#, \#, S, I) \end{aligned}$$

Poi la MTM riposiziona la testina del primo nastro sul primo simbolo della stringa di ingresso:

$$\begin{aligned} \delta(q_2, x, \#) &= (q_2, x, \#, S, I) \quad \forall x \in \Sigma \setminus \{\ \#\} \\ \delta(q_2, \#, \#) &= (q_3, \#, \#, D, I) \end{aligned}$$

Infine copia per la seconda volta l'input e si arresta:

$$\begin{aligned} \delta(q_3, x, \#) &= (q_3, x, x, D, D) \quad \forall x \in \Sigma \setminus \{\ \#\} \\ \delta(q_3, \#, \#) &= (H, \#, \#, I, I) \end{aligned}$$

4.1.7 Gödelizzazione delle macchine di Turing

Nella sezione 3.2 abbiamo annunciato che l'insieme delle funzioni calcolabili è enumerabile. Vediamo ora in dettaglio che ciò è vero per le funzioni T-calcolabili, costruendo esplicitamente una corrispondenza tra tali funzioni e i numeri naturali.

Alla base della costruzione c'è un procedimento noto come Gödelizzazione che consente di *descrivere* o *codificare* una MT mediante un numero. Per semplicità, da ora in avanti, consideriamo MT che hanno un alfabeto costituito da tre soli simboli (ad esempio $\{0, 1, \#\}$). Come abbiamo già detto, questa non è una restrizione al potere computazionale della MT (cioè l'insieme delle funzioni calcolabili da MT con tale restrizione è sempre l'insieme delle funzioni T-calcolabili). Infatti data una qualunque MT $\mathcal{M}$, possiamo facilmente costruire (e invitiamo il lettore a immaginare da sè i dettagli di questa costruzione) un'altra MT che ha un alfabeto di tre (o addirittura due) soli simboli e che calcola esattamente la stessa funzione calcolata da $\mathcal{M}$.

Osserviamo che per descrivere una MT $\mathcal{M}$ è sufficiente descriverne la funzione di transizione δ in quanto una descrizione di δ contiene anche l'insieme dei simboli e degli stati utilizzati da $\mathcal{M}$.

Sia dunque $\mathcal{M} = \langle\{0, 1, \#\}, \{q_1, \ldots, q_n\}, q_1, \delta\rangle$. Fissiamo un ordinamento sui simboli $\{0, 1, \#\}$ e sui simboli di movimento $\{D, S, I\}$ come segue:

$$x_1 = 0, x_2 = 1, x_3 = \#$$
$$z_1 = D, z_2 = S, z_3 = I$$

Indichiamo inoltre, rispettivamente, con $q_{n+1}, q_{n+2}, q_{n+3}$ gli stati finali Y, N, I. Descriviamo la funzione di transizione δ mediante una stringa appartenente a $\{0, 1\}^*$. La generica quintupla $q_i x_j q_k x_l z_m$ può essere descritta mediante i cinque numeri i, j, k, l, m. Rappresentiamo ognuno di questi numeri con una codifica unaria, ovvero rappresentiamo il numero x mediante la concatenazione di x simboli 0. Per distinguere tra loro le codifiche unarie dei cinque numeri, le intervalliamo da simboli 1. Dunque la quintupla $q_i x_j q_k x_l z_m$ viene codificata dalla stringa $0^i 10^j 10^k 10^l 10^m$. Per rappresentare tutta la funzione δ concateniamo le codifiche di tutte le quintuple: stabiliamo un ordine (per esempio lessicografico) tra le codifiche delle quintuple e formiamo la stringa binaria $111P_1 11P_2 11 \ldots 11P_r 111$, dove P_i, con $i = 1, \ldots, r$, è la i-esima codifica di una quintupla. Ogni stringa binaria può essere considerata come codifica di al più una MT; molte stringhe binarie non codificano alcuna MT. La decodifica di ogni stringa (ben formata) è unica. Chiamiamo $\rho(\mathcal{M})$ la codifica della MT $\mathcal{M}$.

Come sappiamo, ad ogni stringa possiamo associare, attraverso una biiezione, un numero. Pertanto, la Gödelizzazione delle MT definisce una biiezione tra $\mathbf{N}$ e l'insieme delle MT che ci consente di *enumerare* l'insieme delle MT (ovvero assegnare ad ogni MT un numero naturale detto numero di Gödel della MT). Se indico con $\mathcal{M}_x$ la MT corrispondente al numero x e con φ_x la funzione calcolata da $\mathcal{M}_x$, La corrispondenza tra x, $\mathcal{M}_x$ e φ_x definisce una enumerazione delle funzioni T-calcolabili:

$$0 \mapsto \mathcal{M}_0 \mapsto \varphi_0$$
$$1 \mapsto \mathcal{M}_1 \mapsto \varphi_1$$
$$2 \mapsto \mathcal{M}_2 \mapsto \varphi_2$$
$$\ldots$$
$$x \mapsto \mathcal{M}_x \mapsto \varphi_x$$
$$\ldots$$

Si noti che l'enumerazione delle MT è una biiezione tra l'insieme delle MT ed $\mathbf{N}$, ma l'enumerazione delle funzioni T-calcolabili non è biiettiva in quanto come detto esistono infinite MT (ognuna con il proprio numero di Gödel) che calcolano la medesima funzione.

4.1.8 Macchina di Turing Universale

Consideriamo una enumerazione delle MT; definiamo una funzione $g : \mathbf{N} \times \mathbf{N} \mapsto \mathbf{N}$, dove $g(x, y) = \varphi_y(x)$: $g(x, y)$ è il numero naturale, se esiste, calcolato dalla y-esima MT con input il numero x. La funzione $g(x, y)$ si può anche considerare come una funzione $g : \mathbf{N} \mapsto \mathbf{N}$, purché sia definita una opportuna biiezione MT-computabile da $\mathbf{N} \times \mathbf{N}$ a $\mathbf{N}$.

Ci chiediamo: la funzione g è calcolabile? Esiste un indice $i \in \mathbf{N}$ tale che $g = \varphi_i$?

La risposta è affermativa: mostreremo nel seguito che è possibile costruire una MT che calcola φ_i. Una tale MT viene detta *macchina di Turing universale* in quanto è in grado

di calcolare il risultato prodotto da una qualunque funzione T-calcolabile con un qualunque valore di input. L'idea alla base della costruzione è quella di *simulare* (o *emulare*, come vuole il gergo della tecnologia informatica) una MT mediante un'altra MT.

Un bravo attore che debba interpretare un personaggio realmente esistito, ha bisogno di prepararsi studiandone foto, filmati, biografie, e magari conoscendolo di persona se questi è ancora vivo. In altre parole ha bisogno di una *descrizione*, quanto più possibile dettagliata, del personaggio. Allo stesso modo, è necessario avere una descrizione dettagliata della MT $\mathcal{M}$ che si intende simulare. Una tale descrizione ci viene fornita dalla codifica $\rho(\mathcal{M})$ introdotta nella sezione 4.1.7.

Definiamo una MT $\mathcal{M}_u$ a tre nastri, che prende in input stringhe del tipo $\rho(\mathcal{M})w$ dove $\rho(\mathcal{M})$ è la codifica della MT che si vuole simulare e w è una stringa binaria che rappresenta il valore che si vuole dare in input a $\mathcal{M}$:

- il primo nastro contiene l'input $\rho(\mathcal{M})w$;
- il secondo nastro simula il nastro di $\mathcal{M}$;
- il terzo nastro contiene la codifica 0^i dello stato corrente q_i di $\mathcal{M}$.

La simulazione di $\mathcal{M}$ da parte di $\mathcal{M}_u$ procede come segue:

1. Inizializzazione dei nastri: $\mathcal{M}_u$ controlla che sul primo nastro ci sia effettivamente la codifica di una MT, ossia $\rho(\mathcal{M})$ deve avere la forma $111P_111P_211\ldots11P_r111$ in cui ogni P_i con $i = 1,\ldots,r$ è del tipo $0^i10^j10^k10^l10^m$ con $j,l,m \in \{1,2,3\}$ poiché tre sono i simboli dell'alfabeto ($\{0,1,\#\}$) e tre sono i simboli di spostamento della testina ($\{D,S,I\}$).

   Se la stringa sul nastro 1 non è ben formata, la $\mathcal{M}_u$ entra in uno stato che sposta continuamente la testina a destra, e pertanto $\mathcal{M}_u$ non termina la computazione. Altrimenti copia la stringa w sul nastro 2 e inizializza il nastro 3 a 0 (che è la codifica di q_1, stato iniziale di $\mathcal{M}$). Infine riposiziona la testina di ciascun nastro sul carattere più a sinistra.

2. $\mathcal{M}_u$ procede alla lettura del simbolo puntato dalla testina sul nastro 2: se tale simbolo è x_j e lo stato corrente (contenuto nel nastro 3) è 0^i, $\mathcal{M}_u$ cerca sul primo nastro la sottostringa 110^i10^j1 che identifica univocamente la quintupla che descrive la transizione $\delta(q_i,x_j)$ di $\mathcal{M}$. Se, invece, $0^i10^j10^k10^l10^m$ è la codifica di tale quintupla, $\mathcal{M}_u$ scrive la stringa 0^k sul nastro 3, sostituisce, sul nastro 2, il simbolo x_i con x_l e sposta la testina nella direzione indicata da z_m.

3. Se il nastro 3 contiene la codifica di uno stato finale di $\mathcal{M}$, $\mathcal{M}_u$ entra nel corrispondente stato finale, altrimenti torna al passo 2.

La macchina $\mathcal{M}_u$ è dunque in grado di simulare il comportamento di una qualunque altra MT con qualunque valore in ingresso. Si noti che $\mathcal{M}_u$ non calcola esattamente la funzione g sopra definita. Tuttavia è un semplice esercizio che lasciamo al lettore costruire una MT che calcoli g utilizzando come "componente" $\mathcal{M}_u$. Tra i modelli di calcolo che prenderemo in esame, $\mathcal{M}_u$ è, per un aspetto importante, quello che più si avvicina ad un ordinario (e reale) calcolatore *general purpose*. Mentre le MT e le RAM (che tratteremo nella prossima sezione)

assomigliano piuttosto ad un calcolatore per uso speciale, avendo un programma "cablato" al loro interno, $\mathcal{M}_u$ introduce la possibilità di leggere dalla memoria ed eseguire un programma, rappresentato dalla codifica $\rho(\mathcal{M})$ di una MT.

4.2 Linguaggi decidibili e accettabili

4.2.1 Linguaggi decidibili e accettabili

Le MT possono essere usate in qualità di riconoscitori o accettori allo scopo di definire linguaggi. Puntualizziamo che, in questo contesto, a differenza di quanto detto per gli AF, i termini *riconoscitore* e *accettore* non sono sinonimi. Una MT che riconosce un linguaggio L definito su un alfabeto A è in grado di *decidere* se una generica stringa x di A^* appartiene o meno a L. Una MT che *accetta* un linguaggio L definito su un alfabeto A è in grado di confermare l'appartenenza di una stringa $x \in A^*$ al linguaggio L solo se $x \in L$ mentre nel caso che x non appartenga a L la MT può anche non terminare la sua computazione.

I linguaggi riconosciuti dalle MT vengono detti *decidibili*; i linguaggi accettati dalle MT vengono detti *accettabili*. Diamo un esempio di linguaggio decidibile.

ESEMPIO

Si consideri il linguaggio $L = \{a^n b^n c^n \mid n \geq 1\}$. L è decidibile. Infatti la MT $\mathcal{M}$ a due nastri che riconosce L opera nel seguente modo:

- Quando $\mathcal{M}$ si trova nello stato iniziale q_0 l'input è contenuto nel primo nastro e la testina è posizionata sul primo carattere; il secondo nastro è vuoto e la testina è posizionata sul simbolo Z_0.
- Leggendo a, $\mathcal{M}$ passa allo stato q_1 e muove la testina verso destra.
- $\mathcal{M}$ legge tutte le a fino a che non viene raggiunta la prima b (se esiste). Nel fare ciò, $\mathcal{M}$ scrive tanti $*$ alla destra di Z_0 nel secondo nastro quante sono le a del nastro di input.
- $\mathcal{M}$ legge tutte le b. Nel fare ciò, muove verso sinistra la testina del secondo nastro di tante posizioni quante sono le b del nastro di input. In questo modo, dopo aver letto tutte le b, la testina è posizionata su Z_0 se e solo se il numero di b eguaglia il numero delle a.
- $\mathcal{M}$ legge tutte le c (se ve ne sono). Nel fare ciò muove la testina del suo nastro di memoria verso destra di tante posizioni quante sono le c lette in ingresso. In questo modo, quando si raggiunge il primo carattere bianco a destra della c posta più a sinistra, la testina del nastro di memoria raggiunge il primo bianco alla destra dei $*$ se e solo se il numero delle c uguaglia il numero delle b. In tal caso $\mathcal{M}$ riconosce la stringa di ingresso.

In tutti gli altri casi $\mathcal{M}$ rifiuta la stringa di ingresso.

Per definire formalmente il concetto di linguaggio decidibile e linguaggio accettabile, dobbiamo stabilire una convenzione in base alla quale la MT segnala il riconoscimento o il non riconoscimento di una stringa. L'idea più naturale è definire una funzione che ci dica se una stringa appartiene al linguaggio o meno. Se la funzione può essere calcolata, diremo che il linguaggio ad essa associato è decidibile:

Definizione 21. *Un linguaggio $L \subseteq A^*$ è decidibile se e solo se la funzione $f_L : A^* \mapsto \{1, 0\}$ è calcolabile, dove:*

$$f_L(x) = \begin{cases} 1 & \text{se } x \in L \\ 0 & \text{se } x \notin L \end{cases}$$

Potremmo calcolare la funzione f_L mediante una MT usando le usuali convenzioni per la rappresentazione dell'output di una funzione. Avremo dunque che un linguaggio è decidibile se e solo se esiste un MT $\mathcal{M}$ tale che:

- l'alfabeto di $\mathcal{M}$ contiene $A \cup \{s, i, n, o\}$, cioè deve comprendere i simboli di A per rappresentare ogni stringa di L e i simboli s,i,n,o per rappresentare l'output della funzione
- $\forall x \in L,\ q_0 x \vdash^* \alpha q \beta,\ q \in F$ e $\alpha\beta = 1$ (ogni stringa $x \in L$ è accettata da $\mathcal{M}$)
- $\forall x \in A^* \setminus L,\ q_0 x \vdash^* \alpha q \beta,\ q \in F$ e $\alpha\beta = 0$ (ogni stringa $x \in A^* \setminus L$ è rifiutata da $\mathcal{M}$)

Tuttavia, risulta più comodo utilizzare una diversa convenzione, operando una differenziazione tra gli stati finali della MT e utilizzando proprio gli stati per codificare l'output. Al lettore attento non sarà sfuggito che nella definizione 14 abbiamo introdotto tre diversi stati finali e che finora sono stati utilizzati indifferentemente, come se fossero un solo stato. È giunto il momento di sfruttare i vantaggi offerti dal disporre di più stati finali. Useremo lo stato finale Y (da *Yes*) per rappresentare il riconoscimento di una stringa e lo stato finale N (da *No*) per rappresentare il rifiuto. Abbiamo pertanto la seguente definizione formale:

Definizione 22. *Un linguaggio $L \subseteq A^*$ è decidibile se e solo se esiste una MT $\mathcal{M}$ tale che:*

- *l'alfabeto di $\mathcal{M}$ contiene A, cioè deve comprendere i simboli di A per rappresentare ogni stringa di L*
- *$\forall x \in L,\ q_0 x \vdash^* \alpha q \beta$ con $q = Y$ (ogni stringa $x \in L$ è riconosciuta da $\mathcal{M}$)*
- *$\forall x \in A^* \setminus L,\ q_0 x \vdash^* \alpha q \beta$ con $q = N$ (ogni stringa $x \in A^* \setminus L$ è rifiutata da $\mathcal{M}$)*

Si noti che viene richiesto che la MT $\mathcal{M}$ si fermi per qualunque input x. Dovrebbe risultare evidente, ai 2^5 lettori che ci hanno seguito fin qui, che utilizzare una convenzione piuttosto che un'altra non modifica la decidibilità o meno di un linguaggio. Chi desiderasse una rassicurazione rigorosa, può pensare che le due convezioni corrispondano a due varianti di MT, chiamate MT *riconoscitore_nastro* e MT *riconoscitore_stato*) e dimostrare in modo formale che per ogni MT *riconoscitore_nastro* esiste una MT *riconoscitore_stato* che decide lo stesso linguaggio, e viceversa.

ESERCIZIO 9. Si dimostri in modo formale che per ogni MT *riconoscitore_nastro* esiste una MT *riconoscitore_stato* che decide lo stesso linguaggio, e viceversa.

Questo fatto ci conferma ancora una volta che varianti del genere non sono sufficienti a cambiare la sostanza del potere computazionale del modello, e che pertanto non sono significative. Possiamo scegliere liberamente la variante che riteniamo più comoda o "più simpatica".

Definizione 23. *Un linguaggio $L \subseteq A^*$ è accettabile se e solo se esiste una MT $\mathcal{M}$ tale che:*

- *l'alfabeto di $\mathcal{M}$ contiene A*
- $\forall x \in A^*,\ x \in L \leftrightarrow q_0 x \vdash^* \alpha q \beta$ *con* $q = Y$ *(ogni stringa $x \in L$ è riconosciuta da $\mathcal{M}$)*

Il caso dell'accettazione è più debole: infatti si richiede che la MT termini la propria computazione solo su input $x \in L$. Altrimenti, se $x \notin L$, la MT può fermarsi e rifiutare x oppure non terminare la sua computazione (c'è, in questo caso, assenza di decisione).

Il problema del calcolo di una funzione $f : \mathbf{N} \mapsto \mathbf{N}$ si può ricondurre al problema di riconoscere un determinato linguaggio L, cioè decidere l'appartenenza di una stringa x ad L. Il calcolo di $f : \mathbf{N} \mapsto \mathbf{N}$ si può ricondurre a quello del riconoscimento di $L = \{a^x b^{f(x)} \mid x \geq 0\}$. In altre parole, $f(x) = y$ se e solo se $a^x b^y \in L$.

ESEMPIO

Data la funzione $f : \mathbf{N} \mapsto \mathbf{N}$ tale che $f(x) = 2x$, il problema del calcolo di f si può ricondurre a quello del riconoscimento del linguaggio

$$L = \{a^x b^{2x} \mid x \geq 0\} = \{\epsilon, abb, aabbbb, \ldots, a^i b^{2i}, \ldots\}$$

.

Decidere l'appartenenza di una stringa x ad un linguaggio L definito su un alfabeto A (così come il calcolo di una funzione) non può essere sempre risolto mediante un programma. Infatti:

- A^* è un insieme numerabile (ricordiamo che le stringhe di A^* possono essere enumerate in base all'ordinamento lessicografico) e, di conseguenza, anche $L \subseteq A^*$ lo è
- l'insieme di tutti i linguaggi di A^* è continuo (perché è l'insieme delle parti di un insieme numerabile)
- l'insieme dei programmi che possono essere scritti utilizzando un qualunque linguaggio di programmazione è numerabile

L'insieme dei programmi non può essere messo in corrispondenza con l'insieme dei linguaggi: esistono più linguaggi da *riconoscere* che programmi che *riconoscano*. In altre parole esistono linguaggi per i quali non esiste alcun programma di riconoscimento.

4.2.2 Proprietà di chiusura di linguaggi decidibili e accettabili

Teorema 24. *Il complemento $\overline{L}$ di un linguaggio decidibile L è decidibile.*

DIMOSTRAZIONE. Se L un linguaggio decidibile, allora esiste una MT $\mathcal{M} = \langle \Sigma, Q, q_0, \delta \rangle$, che si arresta su tutti gli input, tale che:

- $\forall x \in L$, $q_0 x \vdash^* \alpha q \beta$, con $q = Y$;
- $\forall x \in \Sigma^* \setminus L$, $q_0 x \vdash^* \alpha q \beta$, con $q = N$.

É possibile definire una MT $\mathcal{M}' = \langle \Sigma, Q, q_0, \delta' \rangle$ che decide il complemento di L, a partire da $\mathcal{M}$, come segue. $\forall q \in Q, \forall a \in \Sigma$:

- se $\delta(q, a) = (p, b, z)$, con $p \notin F$,
  allora $\delta'(q, a) = \delta(q, a) = (p, b, z)$;
- se $\delta(q, a) = (Y, b, z)$, allora $\delta'(q, a) = (N, b, z)$;
- se $\delta(q, a) = (N, b, z)$, allora $\delta'(q, a) = (Y, b, z)$.

□

Teorema 25. *L'unione di due linguaggi decidibili è decidibile. L'unione di due linguaggi accettabili è accettabile.*

DIMOSTRAZIONE. Siano L_1 e L_2 due linguaggi decisi dalle MT $\mathcal{M}_1$ e $\mathcal{M}_2$. La MT $\mathcal{M}$ che decide $L_1 \cup L_2$ agisce come segue:

- simula $\mathcal{M}_1$;
- se $\mathcal{M}_1$ accetta, allora anche $\mathcal{M}$ accetta;
- se $\mathcal{M}_1$ rifiuta, allora $\mathcal{M}$ simula $\mathcal{M}_2$ ed accetta solo se $\mathcal{M}_2$ accetta.

Poiché per ipotesi le MT si fermano per ogni input anche $\mathcal{M}$ si ferma e chiaramente decide $L_1 \cup L_2$.

Se L_1 ed L_2 sono accettati dalle MT $\mathcal{M}_1$ e $\mathcal{M}_2$, la costruzione precedente non funziona, perché la MT $\mathcal{M}_1$ potrebbe non fermarsi.

In questo caso, la MT $\mathcal{M}$ può simultaneamente simulare $\mathcal{M}_1$ e $\mathcal{M}_2$ su nastri distinti. Allora $\mathcal{M}$ accetta se almeno una delle due MT accetta. □

Teorema 26. *Se un linguaggio L ed il suo complemento $\overline{L}$ sono entrambi accettabili allora L (e quindi $\overline{L}$) è decidibile.*

DIMOSTRAZIONE. Siano $\mathcal{M}_1$ e $\mathcal{M}_2$ le MT che accettano L e $\overline{L}$. Una MT $\mathcal{M}$ che decide L, simula simultaneamente $\mathcal{M}_1$ e $\mathcal{M}_2$. $\mathcal{M}$ accetta w se $\mathcal{M}_1$ accetta w, rifiuta w se $\mathcal{M}_2$ accetta w. Ogni $w \in \Sigma^*$ è in $L \cup \overline{L}$, per cui esattamente una MT fra $\mathcal{M}_1$ e $\mathcal{M}_2$ accetterà w. □

I teoremi 23 e 25 hanno importanti conseguenze. Siano L e $\overline{L}$ una coppia di linguaggi complementari. Allora solo una delle seguenti situazioni può essere vera:

- entrambi L e $\overline{L}$ sono decidibili;
- entrambi L e $\overline{L}$ non sono accettabili;
- uno tra L e $\overline{L}$ è accettabile ma indecidibile, l'altro non è accettabile.

Un'importante tecnica per dimostrare che un linguaggio L è indecidibile è provare per diagonalizzazione che $\overline{L}$ non è accettabile.

4.2.3 Linguaggi accettabili e non accettabili

- Un linguaggio non accettabile

  Sia $w_1, w_2, \ldots, w_i, \ldots$ una enumerazione di tutte le stringhe in $\{0,1\}^*$ (ad es., stringhe in ordine lessicografico).

  Sia $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_j, \ldots$ una enumerazione delle macchine di Turing.

  Possiamo definire una tabella infinita che indica, per ogni stringa $w_j \in \{0,1\}^*$ e per ogni MT $\mathcal{M}_i$, se $w_j \in L(\mathcal{M}_i)$, ovvero se w_j è accettata dalla MT $\mathcal{M}_i$.

	w_1	w_2	w_3	$\ldots$	w_j	$\ldots$
$\mathcal{M}_1$	b_{11}	b_{12}	b_{13}	$\ldots$		
$\mathcal{M}_2$	b_{21}	b_{22}	b_{23}	$\ldots$		
$\mathcal{M}_3$	b_{31}	b_{32}	b_{33}	$\ldots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$			
$\mathcal{M}_i$					b_{ij}	
$\ldots$						
$\mathbf{L_d}$	$\mathbf{d_1}$	$\mathbf{d_2}$	$\mathbf{d_3}$	$\ldots$	$\mathbf{d_j}$	

$$b_{ij} = \begin{cases} 1 & \text{se } w_j \in L(\mathcal{M}_i) \\ 0 & \text{altrimenti} \end{cases}$$

Sia L_d il linguaggio costruito per diagonalizzazione nel seguente modo:

$$w_j \in L_d \text{ se e soltanto se } w_j \notin L(\mathcal{M}_j)$$

In altre parole:

$$d_j = 1 \text{ se e soltanto se } b_{jj} = 0$$

Teorema 27. *Il linguaggio*

$$L_d = \{w_i \in \{0,1\}^* \mid w_i \notin L(\mathcal{M}_i)\}$$

chiamato linguaggio diagonale *non è accettabile.*

DIMOSTRAZIONE. Supponiamo per assurdo che L_d sia accettabile.

Se L_d fosse accettabile, allora dovrebbe esistere una MT che lo accetta. Sia k l'indice di tale MT nell'enumerazione delle macchine di Turing considerata nella tabella.

L'ipotesi $L_d = L(\mathcal{M}_k)$ genera la seguente contraddizione:

$$w_k \in L(\mathcal{M}_k) \text{ se e soltanto se } w_k \notin L(\mathcal{M}_k)$$

pertanto possiamo concludere che la nostra assunzione è falsa.

□

- Un linguaggio accettabile e non decidibile

  Sia $\rho(\mathcal{M})$ la codifica della MT $\mathcal{M}$.

  Teorema 28. *Il seguente linguaggio, chiamato* linguaggio universale

  $$L_u = \{\rho(\mathcal{M})w \mid w \in \{0,1\}^* \wedge w \in L(\mathcal{M})\}$$

  è accettabile.

  DIMOSTRAZIONE. Definiamo una MT $\mathcal{M}'$ a 3 nastri il cui funzionamento può essere schematizzato come segue:

  - il nastro 1 contiene l'input $\rho(\mathcal{M})w$;
  - il nastro 2 simula il nastro di $\mathcal{M}$;
  - il nastro 3 contiene la codifica 0^i dello stato attuale q_i di $\mathcal{M}$.

  La simulazione di $\mathcal{M}$ da parte di $\mathcal{M}'$ procede come segue:

  1. inizializza i nastri:
     - controlla sul nastro 1 che la stringa in input contenga la codifica di una MT, ossia $\rho(\mathcal{M})$ deve avere la struttura

       $$111M_1 11 \ldots 11M_r 111$$

       dove ogni mossa M_x $(1 \leq x \leq r)$ è del tipo

       $$0^i 10^j 10^k 10^l 10^m,$$

       con $j, l, m \in \{1, 2, 3\}$.
       In caso negativo $\mathcal{M}'$ si ferma e rifiuta la stringa, altrimenti:
     - copia w sul nastro 2;
     - inizializza il nastro 3 a 0 (codifica di q_1, stato iniziale di $\mathcal{M}$);
     - riposiziona le testine dei tre nastri sul carattere più a sinistra;

2. se x_j è il simbolo scandito sul nastro 2 e 0^i è il contenuto corrente del nastro 3, $\mathcal{M}'$ cerca sul primo nastro la sottostringa 110^i10^j1:
   - in caso negativo $\mathcal{M}'$ si ferma e rifiuta la stringa;
   - altrimenti, se $0^i10^j10^k10^l10^m$ è la codifica del passo da simulare:
     * scrive la stringa 0^k sul nastro 3;
     * scrive il simbolo x_l sul nastro 2;
     * sposta la testina del nastro 2 nella direzione z_m;
3. se il nastro 3 contiene la codifica di uno stato finale di $\mathcal{M}$, $\mathcal{M}'$ entra nel corrispondente stato finale, accettando la stringa, altrimenti torna al passo 2.

□

4.3 La macchina a registri - RAM

4.3.1 Descrizione

La macchina RAM è un modello di calcolo basato sulla struttura classica degli elaboratori: è un'astrazione di un'architettura convenzionale di Von Neumann. Essa è dotata di:

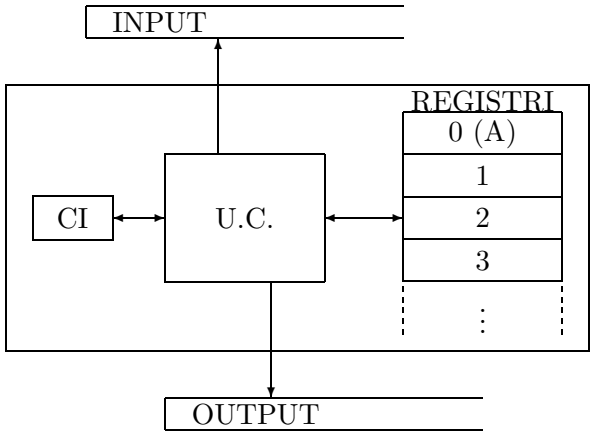
- un nastro di ingresso e uno di uscita mediante i quali la macchina scambia informazioni con il mondo esterno; tali nastri sono composti da un numero illimitato di celle ognuna delle quali può contenere un numero naturale grande a piacere. L'accesso ai nastri è di tipo sequenziale: ogni operazione di lettura (scrittura) fa avanzare la testina del nastro di una cella verso destra.

- una memoria ad accesso casuale composta da un numero illimitato di registri (celle di memoria) ciascuno dei quali può contenere un numero naturale grande a piacere ed è accessibile in base al proprio numero d'ordine. Indichiamo con $R[i]$ il contenuto del registro i-esimo.

- un registro particolare, il numero 0 nell'ordinamento, chiamato accumulatore (A) che si usa per contenere il valore di uno dei due operandi su cui agiscono le istruzioni della macchina. Indichiamo con $R[0]$ il contenuto dell'accumulatore.

- un registro CI, chiamato contatore delle istruzioni, che contiene il numero d'ordine dell'istruzione successiva da eseguire.

- un'unità centrale (UC) capace di eseguire le istruzioni del linguaggio di programmazione.



Il programma della RAM non è contenuto in memoria: la RAM è una macchina cablata a programma fisso. Le istruzioni sono di vario tipo: di input/output, di salto, di trasferimento, aritmetiche e di arresto. In genere gli operandi su cui agiscono le istruzioni sono contenuti nei registri.

Un operando può essere:

- un dato direttamente espresso nell'istruzione (operando costante, denotato con $= n$);
- un dato contenuto in un registro indirizzato in modo diretto tramite il corrispondente numero d'ordine, denotato con n (l'operando è $R[n]$);
- un dato indirizzato in modo indiretto tramite il contenuto di un altro registro, denotato con $*n$ (l'operando è il contenuto del registro di indirizzo $R[n]$, ovvero $R[R[n]]$).

Il contenuto del registro CI varia nell'insieme $\{1, \ldots, m\}$, dove m è la dimensione del programma. Il riferimento alle istruzioni del programma, che sono memorizzate in un'area apposita della macchina e non nei registri, avviene mediante etichette (nomi simbolici) alle quali corrispondono valori di CI.

4.3.2 Istruzioni della RAM

Le istruzioni della macchina RAM sono:

- istruzioni di trasferimento
  - LOAD[= |∗]⟨naturale⟩
    trasferisce nell'accumulatore un operando
  - STORE[∗]⟨naturale⟩
    trasferisce in un registro il contenuto dell'accumulatore
- istruzioni aritmetiche: permettono di eseguire le quattro operazioni aritmetiche tra il contenuto dell'accumulatore e un operando. Il risultato rimane nell'accumulatore.
  - ADD[= |∗]⟨naturale⟩

  - SUB[= |∗|⟨naturale⟩ (x SUB $y = 0$ se $y > x$)
  - MULT[= |∗|⟨naturale⟩
  - DIV[= |∗|⟨naturale⟩ (divisione intera)

- istruzioni di salto: permettono di alterare l'esecuzione sequenziale delle istruzioni
  - JUMP⟨label⟩
    modifica il contenuto di CI con il valore corrispondente all'etichetta ⟨label⟩
  - JGTZ⟨label⟩
    modifica il contenuto di CI con il valore corrispondente all'etichetta ⟨label⟩ solo se il valore contenuto nell'accumulatore è maggiore di zero.
  - JZERO⟨label⟩ modifica il contenuto di CI con il valore corrispondente all'etichetta ⟨label⟩ solo se il valore contenuto nell'accumulatore è uguale a zero.

- istruzioni di input/output:
  - READ[∗]⟨intero⟩
    trasferisce il numero naturale letto dalla testina sul nastro di ingresso, in un registro
  - WRITE[= |∗|⟨intero⟩
    scrive il valore dell'operando sul nastro di uscita

- istruzioni di arresto:
  - HALT
    arresta l'esecuzione del programma. La macchina si ferma in uno stato corretto quando viene raggiunta l'istruzione HALT.

ESEMPIO

Definire un programma per RAM per il calcolo di x^y (gli interi $x > 0$ e $y \geq 0$ sono forniti sul nastro di input).

```
          READ    101     {leggi x}
          READ    102     {leggi y}
          LOAD    =1      {inizializza RIS a 1
          STORE   103     ... }
loop      LOAD    102     {if y = 0 then salta a exit
          JZERO   exit    ... }
          LOAD    101     {RIS := RIS * x
          MULT    103     ...
          STORE   103     ... }
          LOAD    102     { y := y - 1
          SUB     =1      ...
```

```
          STORE   102     ... }
          JUMP    loop
exit      WRITE   103     {scrivi RIS}
          HALT
```

ESEMPIO

Definire un programma per RAM che calcoli la somma di $n \geq 0$ numeri interi $a_1, \ldots, a_n$.

```
          READ    103     {leggi n}
          LOAD    103     {if n = 0 then HALT
          JZERO   exit    ...}
          LOAD    103     {I := n
          STORE   101     ... }
          LOAD    =104    {IND := 104
          STORE   102     ... }
loop1     READ    *102    {leggi a_{n-I+1}}
          LOAD    101     {I := I - 1
          SUB     =1      ...
          STORE   101     ...}
          JZERO   compute
          LOAD    102     {IND := IND + 1
          SUM     =1      ...
          STORE   102     ...}
          JUMP    loop1
compute   LOAD    =0      {RIS := 0
          STORE   101     ...}
          LOAD    =104    {IND := 104
          STORE   102     ... }
loop2     LOAD    101     {RIS := RIS + a_i
          SUM     *102    ...
          STORE   101     ...}
          LOAD    102     {IND := IND + 1
          SUM     =1      ...
          STORE   102     ...}
          LOAD    103     {n := n - 1
          SUB     =1      ...
          STORE   103     ...}
          LOAD    103     {if n = 0 then salta a 33
          JZERO   output  ...}
          JUMP    loop2
output    WRITE   101     {scrivi RIS}
```

exit HALT

4.3.3 Equivalenza tra MT e RAM

I programmi per la RAM si possono interpretare come funzioni. Un programma che legge in ingresso n valori naturali e produce in uscita m valori naturali rappresenta una funzione $f : \mathbf{N}^n \mapsto \mathbf{N}^m$. Analogamente a quanto si è fatto per le macchine di Turing è possibile definire un concetto di funzione calcolabile con macchine a registri.

Definizione 29. *Una funzione $f : \mathbf{N}^n \mapsto \mathbf{N}$ è calcolabile con RAM se esiste un programma P per RAM tale che :*

- *se $f(x_1, \ldots, x_n) = y$ allora il programma P, inizializzato con $x_1, \ldots, x_n$ sul nastro di input, termina con y sul nastro di output;*
- *se $f(x_1, \ldots, x_n)$ non è definita allora il programma P non termina.*

Dimostriamo che l'insieme delle funzioni calcolabili con macchine a registri coincide con l'insieme delle funzioni Turing-calcolabili.

Teorema 30. *Data una MT $\mathcal{M}$ con nastro seminfinito e alfabeto $\sum = \{0, 1\}$, esiste una RAM con relativo programma P tale che se $\mathcal{M}$ esegue una computazione dalla configurazione iniziale q_0x a una configurazione finale $\alpha q \beta$ e se la RAM è inizializzata con la stringa x sul nastro di ingresso, al termine della computazione la RAM avrà sul nastro di uscita la stringa $\alpha\beta$.*

DIMOSTRAZIONE.

- La RAM inizia la simulazione copiando la stringa x dal nastro di ingresso in una serie di registri, ad esempio dal registro 100 al registro $100 + |x| - 1$.
- Un ulteriore registro (e.g. 99) viene usato per simulare la testina, ovvero per indirizzare la cella correntemente letta dalla testina della MT $\mathcal{M}$.
- Per ogni stato di $\mathcal{M}$ il programma P conterrà una sequenza di istruzioni che realizzano la funzione di transizione. Ad esempio, se $\delta(p, 0) = (q, 1, d)$ e $\delta(p, 1) = (p, 1, s)$, P contiene il seguente frammento (che supponiamo inizi con l'etichetta p):

```
        ...
p       LOAD    *99
        JGTZ    p'
        LOAD    =1
        STORE   *99
        LOAD    99
        ADD     =1
        STORE   99
```

```
        JUMP    q
p'      LOAD    99
        SUB     =1
        STORE   99
        JUMP    p
        ...
```

- Lo spostamento della testina di solito viene realizzato con una semplice modifica del registro usato per simulare la testina (nel nostro caso il registro 99). Si deve però porre attenzione al fatto che mentre il nastro della MT è infinito "in entrambe le direzioni", i registri che usiamo per simulare tale nastro sono infiniti ma "in una sola direzione". Nel nostro esempio usiamo i registri dal numero 100 in poi per simulare il nastro della MT. In particolare il registro 100 rappresenta la cella del nastro su cui si trova la testina della MT all'inizio della computazione. Cosa succede se la MT sposta la testina a sinistra di tale cella? La RAM tenterebbe di simulare questo spostamento scrivendo nel registro 99, ma poiché tale registro viene usato per contenere la posizione della testina, ciò provocherebbe un grave malfunzionamento della RAM. Pertanto quando si sposta la testina è necessario controllare che non si vada a sovrascrivere il registro 99 (o altri registri con indice minore di 100): se la testina si trova sul simbolo contenuto nel registro 100 e viene spostata verso sinistra, si deve far scorrere "verso il basso" il contenuto dei registri (cioè si sposta il contenuto dell'i-esimo registro nel registro $i+1$-esimo, in modo da "liberare" il registro 100 per ospitare la nuova cella usata dalla testina.
- Quando $\mathcal{M}$ entra in uno stato finale, il contenuto dei registri usati durante la simulazione viene copiato sul nastro di uscita.

□

Teorema 31. *Data una RAM con programma P che calcola la funzione $f : \mathbf{N}^n \mapsto \mathbf{N}$ esiste una MT $\mathcal{M}$ tale che:*

- *se $f(x_1, \ldots, x_n) = y$ e sul nastro di input sono memorizzati in binario gli interi $x_1, \ldots, x_n$ (separati tra loro da un opportuno marcatore), la MT $\mathcal{M}$ termina con la rappresentazione binaria di y (ricordiamo che l'alfabeto di $\mathcal{M}$ è $\{0, 1\}$) sul nastro di output;*
- *se $f(x_1, \ldots, x_n)$ non è definita la macchina $\mathcal{M}$ non termina.*

DIMOSTRAZIONE. Per la simulazione utilizziamo una MTM con 5 nastri:

- I nastri 1 e 2 di $\mathcal{M}$ contengono, rispettivamente, il nastro di ingresso e quello di uscita della RAM
- Sul nastro 3 sono memorizzati tutti i contenuti dei registri effettivamente utilizzati dalla RAM, ciascuno preceduto dal numero d'ordine del registro stesso.

- Per ogni registro R_i della RAM, si rappresenta la coppia (i,c_i), dove i è l'indice del registro e c_i il suo contenuto. I numeri i e c_i sono rappresentati in codifica binaria.

  Per separare le coppie relative a registri diversi si utilizza il carattere speciale \$, mentre per separare l'indice di ogni registro dal contenuto si usa il carattere speciale ",".

- Il nastro 4 contiene la codifica binaria del valore memorizzato nell'accumulatore

- Il nastro 5 viene usato come nastro di lavoro ausiliario.

Ogni istruzione del programma della RAM viene simulata da una serie di stati e dalle relative transizioni.

- Le istruzioni di trasferimento vengono simulate copiando la porzione del nastro 3 che rappresenta il registro interessato dall'istruzione, nel nastro 4, o viceversa.

  In dettaglio:

  - Per prima cosa si cerca all'interno del nastro 3 la coppia (i, c_i), usando il valore i come chiave di ricerca.
  - Completata la ricerca, l'operazione di lettura (LOAD) non presenta problemi.
  - Per la scrittura (STORE), c'è il rischio di sovrascrivere anche parti del nastro 3 relative ad altri registri nel caso in cui la codifica del nuovo valore sia di dimensione maggiore del precedente.
  - In questi casi la coppia (i, c_i) deve essere cancellata dalla posizione corrente (scrivendo un opportuno simbolo speciale non #) e all'estremità destra del nastro si deve aggiungere la coppia (i, c'_i) dove c'_i è il nuovo valore del registro i;

- Le istruzioni aritmetiche vengono simulate manipolando il valore contenuto nel nastro 4 mediante una opportuna serie di transizioni tra stati.

- Le istruzioni di I/O (READ e WRITE) vengono simulate mediante copie dal nastro 1 nel nastro 3 (READ) o dal nastro 3 al nastro 2 (WRITE);

- Le istruzioni di salto vengono simulate verificando l'eventuale condizione di salto sul nastro 4 ed effettuando una transizione nel primo degli stati che simula l'istruzione RAM da eseguire.

□

4.4 La macchina di Turing non deterministica - MTND

4.4.1 Descrizione

Una importante variante della MT è costituita dalla cosiddetta macchina di Turing non deterministica (MTND), in cui le transizioni delle configurazioni non sono univocamente

determinate, ovvero dato uno stato interno della macchina e un simbolo letto, la macchina può avere più di una possibilità di transizione. Il modello non deterministico è equivalente, dal punto di vista computazionale, al precedente modello deterministico.

Definizione 32. *Una macchina di Turing non deterministica MTND è una quadrupla* $\mathcal{M} = \langle \Sigma, Q, q_0, \delta_N \rangle$ *in cui, a differenza di una MT deterministica,* δ_N *è una relazione del tipo:*

$$\delta_N \subseteq Q \times \Sigma \times (Q \cup F) \times \Sigma \times \{D, S, I\}$$

Il funzionamento di una MTND è specificato da un insieme di quintuple $q_i x_j q_k x_l z_m$ tra le quali ce ne possono essere più di una aventi gli stessi simboli $q_i x_j$. Ad esempio le seguenti quintuple

$$q_0 a q_1 a D$$
$$q_0 a q_2 b S$$
$$q_0 a H \# I$$

descrivono una MTND che, trovandosi in uno stato q_0, in corrispondenza dell'input a, avvia tre passi computazionali in modo parallelo.

La relazione δ_N può anche essere vista come una funzione che ad ogni coppia (q_i, x_j) associa un insieme di terne (q_k, x_l, z_m) ossia:

$$\delta_N : Q \times \Sigma \mapsto 2^{(Q \cup F) \times \Sigma \times \{D,S,I\}}$$

dove $2^{(Q \cup F) \times \Sigma \times \{D,S,I\}}$ è l'insieme di tutti i sottoinsiemi costituiti dalle terne (q_k, x_l, z_m) con $q_k \in Q \cup F, x_l \in \Sigma, z_m \in \{D, S, I\}$. Se tale insieme è vuoto (l'insieme vuoto è contenuto in $2^{(Q \cup F) \times \Sigma \times \{D,S,I\}}$) assumiamo che la MTND si arresti immediatamente entrando nello stato H (quindi $\delta_N(q_i, x_j) = \emptyset$ è equivalente a $\delta_N(q_i, x_j) = (H, x_j, I)$). Possiamo scrivere in maniera più concisa la sequenza di quintuple precedente: $\delta_N(q_0, a) = \{(q_1, a, D), (q_2, b, S), (H, \#, I)\}$.

In accordo con la δ_N, per ogni configurazione istantanea la MTND può eseguire contemporaneamente più transizioni, dando luogo a differenti configurazioni successive.

La computazione di una MTND è realizzata mediante un insieme di computazioni deterministiche che procedono in parallelo.

L'insieme delle configurazioni deterministiche assunte da una MTND durante una computazione può essere rappresentato mediante un albero (non necessariamente finito) in cui:

- la radice è associata alla configurazione iniziale;
- i nodi sono associati a configurazioni deterministiche della macchina;
- gli archi indicano le transizioni fra configurazioni;
- ogni cammino rappresenta una particolare computazione deterministica;
- una configurazione non deterministica assunta dalla MTND è associata ad un livello;
- tutti i nodi a distanza $k \geq 1$ dalla radice sono associati alle configurazioni che si ottengono non deterministicamente dopo k transizioni.

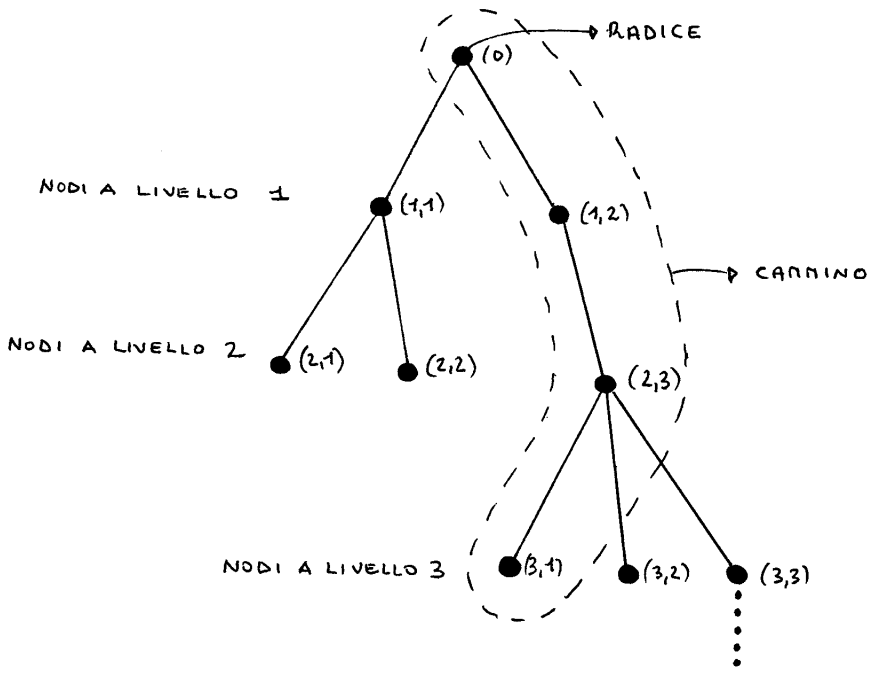


Figura 4.5: Albero delle computazioni di una MTND

Ad ogni nodo corrisponde una configurazione. L'arco tra due nodi sta ad indicare che la funzione δ_N determina una transizione tra una configurazione e l'altra. Un nodo con più archi uscenti denota il carattere non deterministico della funzione δ_N. La computazione $\{0, (1,2), (2,3), (3,3), \ldots\}$ non termina.

ESEMPIO

La MTND $\mathcal{M}$, rappresentata in Figura 4.6, rileva, all'interno della stringa ricevuta in input, la presenza della sottostringa *bac*. Quando si trova nello stato q_0, $\mathcal{M}$ cerca un simbolo b candidato ad essere il primo carattere della sottostringa *bac*. Pertanto, per ogni simbolo a oppure c letto, $\mathcal{M}$ rimane in q_0. Quando invece, dallo stato q_0, viene letto un simbolo b, $\mathcal{M}$ avvia, nondeterministicamente, due computazioni in parallelo. La prima computazione assume che la b letta *non sia* l'inizio della sottostringa cercata e quindi non muta stato continuando a cercare; la seconda, invece, suppone che la b letta *sia* l'inizio di *bac* e dunque, transitando in q_1, inizia a verificare che i due simboli che seguono immediatamente b siano *ac*. Se tale verifica va a buon fine, la computazione accetta la stringa, altrimenti la rifiuta smettendo di cercare. Si noti che non è necessario, nel caso di verifica fallimentare, "tornare indietro" e riprendere la ricerca: infatti la ricerca viene continuata dalla prima computazione avviata in parallelo.

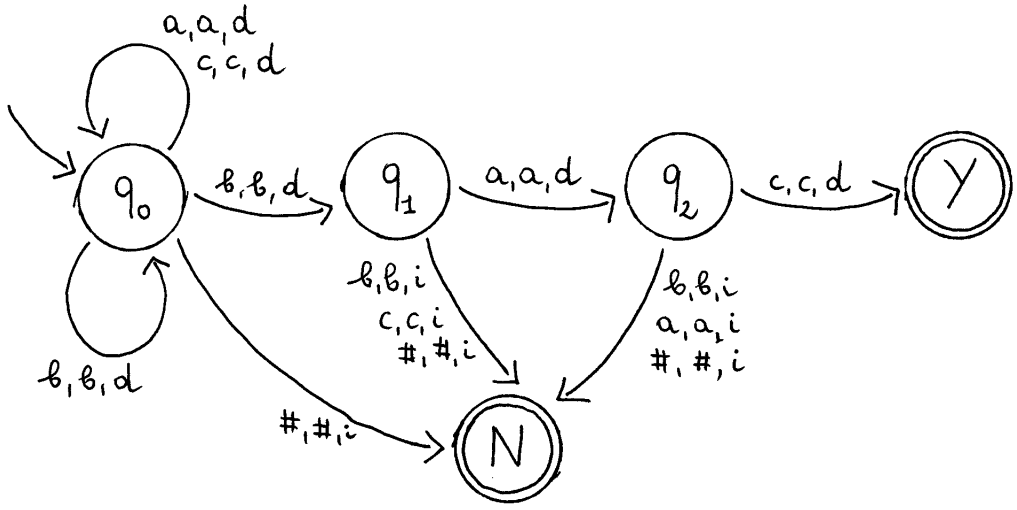


Figura 4.6: Esempio di MTND

OSSERVAZIONE

La computazione di una macchina di Turing deterministica dà luogo a una catena (finita o infinita), come quella rappresentata in Figura 4.7.

La sequenza dei nodi $0, 1, 2, 3, \ldots$ denota la computazione $c_0 \vdash c_1 \vdash c_2 \vdash c_3 \ldots$.

Definizione 33. *Si dice grado di non determinismo di una MTND $\mathcal{M}$, e si indica con $\nu(\mathcal{M})$, il massimo numero di terne $(q_k x_l z_m)$ associate alla coppia $(q_i x_j)$ dalla funzione δ_N, o, equivalentemente, il massimo numero di quintuple che hanno gli stessi simboli iniziali $q_i x_j$.*

Ad esempio sia δ_N la funzione di transizione della MTND $\mathcal{M}$ definita come:

$$\delta_N(q_0, a) = \{(q_0, a, D), (q_1, a, S)\}$$
$$\delta_N(q_1, b) = \{(q_1, b, S), (q_1, c, S)\}$$
$$\delta_N(q_2, c) = \{(q_1, b, D)\}$$

Risulta: $\nu(\mathcal{M}) = 2$

Poiché una MTND potrebbe produrre due differenti output da uno stesso input, diventa necessario determinare un criterio per stabilire quale deve essere considerato il risultato effettivo della computazione non deterministica della macchina. Tale problema si risolve considerando la MTND solamente come dispositivo riconoscitore.

Definizione 34. *Una computazione $c_0, \ldots, c_n$ è accettante se c_0 è una configurazione iniziale e c_n è una configurazione finale tale che $c_n = \alpha Y \beta$, con $\alpha\beta \in \Sigma^*$.*

Definizione 35. *Una computazione $c_0, \ldots, c_n$ è rifiutante se: c_0 è una configurazione iniziale e c_n è una configurazione finale tale che $c_n = \alpha N \beta$ oppure $c_n = \alpha H \beta$, con $\alpha\beta \in \Sigma^*$.*

Figura 4.7: Catena delle computazioni di una MT

Si noti che, poiché la MTND non viene usata per calcolare funzioni, potremmo eliminare lo stato di arresto "generico" H ed avere solo Y ed N come stati finali. Tuttavia preferiamo tenere H per uniformità al caso deterministico.

Le configurazioni non terminanti denotano assenza di responso.

Definizione 36. *Una stringa $x \in \Sigma^*$ è accettata dalla MTND $\mathcal{M}$ se esiste una computazione accettante $c_0, \ldots, c_n$ di $\mathcal{M}$, con $c_0 = \{q_0 x\}$.*

Definizione 37. *Una stringa $x \in \Sigma^*$ è rifiutata dalla MTND $\mathcal{M}$ se ogni computazione di $\mathcal{M}$ a partire dalle configurazione $\{q_0 x\}$, è o rifiutante oppure infinita.*

Definizione 38. *L'insieme $L(\mathcal{M})$ di tutte le stringhe accettate da una MTND $\mathcal{M}$ è detto linguaggio accettato da $\mathcal{M}$.*

4.5 Confronto tra grammatiche e macchine di Turing

Gli automi a stati finiti (più in generale gli accettori a stati finiti) sono stati definiti come modelli capaci di riconoscere una particolare categoria di linguaggi, quelli generati da grammatiche regolari.

Le grammatiche generative (non regolari) sono in grado di generare linguaggi per i quali non esiste l'automa riconoscitore. Abbiamo fatto vedere che per il linguaggio $L = \{a^n b^n \mid n \geq 1\}$ generato dalla grammatica $G = \langle \{a, b\}, \{A, B\}, P, S \rangle$ con P dato da:

$$S \mapsto aAb$$
$$aA \mapsto aaAb$$
$$A \mapsto \epsilon$$

non esiste l'automa riconoscitore. Ci chiediamo adesso se esiste una MT in grado di riconoscere tale linguaggio. La risposta affermativa è nell'esempio che segue.

ESEMPIO

Utilizziamo una MTM con due nastri. L'insieme degli stati Q è $\{q_0, q_1, q_2\}$: q_0 è lo stato iniziale e q_3 è lo stato finale. L'alfabeto è $\{a, b, \$, Z_0\}$.

La funzione di transizione è la seguente:

$$\delta(q_0, a, Z_0) = (q_1, a, Z_0, D, D)$$
$$\delta(q_0, b, Z_0) = (N, b, Z_0, I, I)$$
$$\delta(q_0, \$, Z_0) = (N, \$, Z_0, I, I)$$
$$\delta(q_1, a, \#) = (q_1, a, \$, D, D)$$
$$\delta(q_1, b, \#) = (q_2, b, \#, D, S)$$
$$\delta(q_1, \$, \#) = (N, \$, \#, I, I)$$
$$\delta(q_2, b, \$) = (q_2, b, \$, D, S)$$
$$\delta(q_2, b, Z_0) = (N, b, Z_0, I, I)$$
$$\delta(q_2, \#, \$) = (N, b, Z_0, I, I)$$
$$\delta(q_2, \#, Z_0) = (Y, b, Z_0, I, I)$$

Data una stringa in input alla MTM, se la computazione della macchina termina nello stato finale Y allora tale stringa appartiene al linguaggio L.

Le MT sono modelli in grado di riconoscere una classe di linguaggi generati da grammatiche più ampia di quella che gli automi possono riconoscere. C'è di più: le MT sono modelli equivalenti alle grammatiche formali, vale a dire che per ogni linguaggio $L(G)$ generato da una grammatica formale G esiste una macchina di Turing che accetta L(G) e, viceversa, per ogni linguaggio L accettato da una MT esiste una grammatica G tale che $L(G) = L$.

Teorema 39. *L è generato da una grammatica formale se e soltanto se esiste una macchina di Turing che accetta L.*

DIMOSTRAZIONE. Sia $\mathcal{G} = \langle V_T, V_N, P, S\rangle$ e sia $L = L(\mathcal{G})$. La MTND $\mathcal{M}$ che accetta L agisce come segue:

- la configurazione iniziale è $\langle q_0\alpha\rangle$ ($\alpha \in V_T^*$);
- la MT trasforma la configurazione iniziale in $\langle q\$\alpha\$S\$\rangle$ e tenta di applicare tutte le possibili produzioni a partire da S ottenendo le configurazioni $\$\alpha\$\beta q\$$ per ogni $\beta \in V^*$ tale che $S \Rightarrow^* \beta$.
- Se $\beta \in V_T^*$ essa viene confrontata con α: se $\alpha = \beta$ allora $\alpha \in L(\mathcal{G})$ e quindi viene accettata, altrimenti la MT prosegue cercando altri possibili modi di generare α.

  Chiaramente se $\alpha \in L(\mathcal{G})$ la macchina si arresta. Altrimenti continua a lavorare, continuando nella ricerca o ciclando.

  La parte "solo se" della dimostrazione è lasciata per esercizio al lettore.

□

4.6 Tesi di Church

Le MT hanno la medesima potenza di calcolo dei più familiari programmi sviluppabili nei consueti linguaggi di programmazione ad alto livello. Sebbene i dettagli risulterebbero enormemente comlessi, si potrebbe costruire una MT che simuli l'esecuzione di un programma Pascal: analogamente a quanto fatto per simulare una RAM, sarebbe sufficiente organizzare i nastri della MT al fine di memorizzare i dati su cui il programma opera e simulare l'esecuzione di ogni istruzione Pascal attraverso una opportuna sequenza di passi della MT. Viceversa, non è difficile costruire un programma Pascal che simuli una MT, a patto di poter ignorare le dimensioni finite della memoria del calcolatore reale su si esegue il programma.

Tutti i formalismi noti per modellare i dispositivi di calcolo discreti hanno, al più, il medesimo potere computazionale delle MT. Una profonda analisi delle capacità delle MT e della computazione meccanica in generale, ha condotto Alonso Church a formulare la sua tesi.

TESI DI CHURCH: PRIMA PARTE

Non esiste alcun formalismo, per modellare una determinata computazione *algoritmica*, che sia più potente delle MT e dei formalismi ad essa equivalenti.

La tesi di Church è intrinsecamente non dimostrabile, poiché è basata sulla nozione intuitiva di *calcolo algoritmico*: essa è sostenuta dall'esperienza (dal 1936 ad oggi non è stato individuato nessun formalismo più potente delle MT) e dall'evidenza intuitiva.

TESI DI CHURCH: SECONDA PARTE

Ogni algoritmo può essere codificato in termini di una MT (o di un formalismo equivalente)

Se si riesce a descrivere un algoritmo per risolvere un problema in modo preciso, seppur informale, allora si è in grado anche di progettare una MT che risolve lo stesso problema, indipendentemente dalla lunghezza e tediosità di un simile procedimento. I termini "algoritmico", "meccanico", "effettivo" e "computabile" sono sinonimi.

L'insieme delle funzioni T-calcolabili coincide con l'insieme delle funzioni calcolabili. Per dimostrare che una funzione f è calcolabile non è necessario esibire una MT che calcoli f; è sufficiente, per la tesi di Church, fornire una procedura effettiva scritta in pseudo-codice o in un linguaggio di programmazione.

4.7 Funzioni calcolabili e funzioni non calcolabili

Diamo ora alcuni esempi di funzioni calcolabili e funzioni non calcolabili.

- (Halting Problem) La funzione $g : \mathbf{N} \times \mathbf{N} \mapsto \{0, 1\}$ definita come:

$$g(x,y) = \begin{cases} 1 & \text{se } \varphi_y(x) \text{ è definita} \\ 0 & \text{altrimenti} \end{cases}$$

non è calcolabile. Infatti, supponiamo per assurdo che g sia calcolabile. Definiamo la funzione h come:

$$h(x) = \begin{cases} 1 & \text{se } g(x,x) = 0 \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

h risulta computabile. Infatti, basta apportare semplici modifiche alla macchina $\mathcal{M}$ che calcola g in maniera tale che il risultato sia 0 quando $\mathcal{M}$ restituisce 1 e che la computazione non termini nel caso in cui $\mathcal{M}$ restituisca 1. Sia x_0 il numero di Gödel di una MT $\mathcal{M}_{x_0}$ che calcola h: ossia $\varphi_{x_0} = h$. Diamo in pasto ad h il numero x_0. Può accadere o che $h(x_0) = 1$ oppure che $h(x_0)$ è indefinita. Entrambe queste possibilità generano una contraddizione. Infatti, come abbiamo detto, risulta $h = \varphi_{x_0}$ ma allo stesso tempo h è legata a g. Allora:

- se $h(x_0) = 1$, si ha $\varphi_{x_0}(x_0) = 1$, e quindi φ_{x_0} è definita nel punto x_0; ma allo stesso tempo, se $h(x_0) = 1$, si ha $g(x_0, x_0) = 0$ e quindi dalla definizione di g segue che φ_{x_0} non è definita nel punto x_0;
- se, d'altro canto, $h(x_0)$ è indefinita, essendo $h = \varphi_{x_0}$, si ha che φ_{x_0} è indefinita nel punto x_0; ma allo stesso tempo, se $h(x_0)$ è indefinita, si ha $g(x_0, x_0) = 1$ e quindi dalla definizione di g segue che φ_{x_0} è definita nel punto x_0;

Poiché in entrambi i casi si ottiene una contraddizione, dobbiamo concludere che l'ipotesi di partenza, ovvero la calcolabilità di g, è falsa.

- Analogamente si dimostra che la funzione

$$g(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ è definita} \\ 0 & \text{altrimenti} \end{cases}$$

non è calcolabile.

Consideriamo invece la seguente funzione:

$$g'(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ è definita} \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Tale funzione, pur essendo ottenuta da g con una semplice modifica, risulta calcolabile. Una MT $\mathcal{M}'$ che calcoli g' è molto simile alla MT universale $\mathcal{M}_u$ studiata nella sezione 4.1.8, e opera come segue. $\mathcal{M}'$ dal numero x individua la x-esima MT nella enumerazione delle MT, ovvero la MT $\mathcal{M}_x$ che calcola φ_x. A questo punto, con la stessa tecnica usata da $\mathcal{M}_u$, simula la computazione di $\mathcal{M}_x$ con ingresso x. Se tale computazione

dà come output un valore, allora $\mathcal{M}'$ restituisce 1; se viceversa tale computazione non termina, allora anche $\mathcal{M}'$, per effetto della simulazione, non si arresta e $g'(x)$ risulta indefinita.

- La funzione f definita come

$$f(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è totale} \\ 0 & \text{se } \varphi_x \text{ è parziale} \end{cases}$$

non è calcolabile.

Infatti supponiamo per assurdo che f sia calcolabile. Per definizione f è anche totale e la funzione $g : \mathbf{N} \mapsto \mathbf{N}$, tale che $g(y) = w$ dove w è il numero di Gödel della y-esima MT che calcola una funzione totale risulta a sua volta totale e computabile. Infatti, la MT $\mathcal{M}$ che calcola g con ingresso y, individua qual'è l'y-esima MT che calcola una funzione totale calcolando i valori che assume f successivamente sui numeri da 0 in poi. Viene definita pertanto una enumerazione delle funzioni totali:

N	0	1	2	3	$\cdots$
numeri di Gödel delle funzioni totali	$g(0)$	$g(1)$	$g(2)$	$g(3)$	$\cdots$

La funzione inversa di g, $g^{-1} : \mathbf{N} \mapsto \mathbf{N}$ è una funzione parziale. Essa prende in input un numero naturale e, se questo è un numero di Gödel di qualche f totale, restituisce il numero d'ordine che occupa tale funzione nella enumerazione delle funzioni totali. La funzione g^{-1} è calcolabile: questo fatto deriva dalla calcolabilità di g e dal fatto che g è totale e strettamente crescente.

Consideriamo adesso la funzione: $h(x) = \varphi_{g(x)}(x) + 1$. La funzione h prende in input un numero naturale x e restituisce il valore $\varphi_{g(x)}(x)$ sommato di 1, dove $\varphi_{g(x)}(x)$ è la x-esima funzione totale nella enumerazione ed il suo numero di Gödel è $g(x)$. La funzione h è una funzione totale e calcolabile. Quindi è inclusa nella enumerazione. Sia ω_0 il numero di Gödel di h: $x_0 = g^{-1}(\omega_0)$ è il numero d'ordine di h nella enumerazione delle funzioni totali.

Vediamo cosa succede si diamo in pasto ad h il suo numero d'ordine. Per la definizione di h risulta: $h(x_0) = \varphi_{g(x_0)}(x_0) + 1$. D'altra parte $\varphi_{g(x_0)}(x_0)$ è un altro modo di scrivere $h(x_0)$. Infatti $g(x_0) = \omega_0$, cioè il numero di Gödel di h. Ne deriva la seguente contraddizione: $h(x_0) = \varphi_{g(x_0)}(x_0) + 1 = \varphi_{\omega_0}(x_0) + 1 = h(x_0) + 1$

OSSERVAZIONE

I risultati fin qui ottenuti possono essere riformulati in maniera più generale. Abbiamo appurato che nessun algoritmo può decidere se un dato programma si fermerà per ogni possibile valore di input (ovvero se la funzione associata al programma è totale). Risulta inoltre

impossibile decidere meccanicamente se un dato algoritmo si fermerà prima o poi per un dato valore di ingresso: questo è il problema della terminazione delle MT.

- Definiamo la funzione h come:

$$h(x) = \begin{cases} 1 & \text{se } \varphi_{\bar{y}}(\bar{x}) \text{ è definita} \\ 0 & \text{se } \varphi_{\bar{y}}(\bar{x}) \text{ non è definita} \end{cases}$$

  h è la funzione costante 1 oppure è la funzione costante 0, a seconda del valore di verità della frase "$\varphi_{\bar{y}}(\bar{x})$ è definita" . Di conseguenza la funzione h è computabile poiché può consistere solo nella funzione costante 1 o nella funzione costante 0, ciascuna delle quali è computabile. Non è comunque noto come calcolare h, anche se è noto che è computabile.

- Consideriamo la seguente funzione:

$$h(x) = \begin{cases} 1 & \text{se Dio esiste} \\ 0 & \text{se Dio non esiste} \end{cases}$$

  h è certamente computabile, poiché corrisponde alla funzione costante 1 oppure alla funzione costante 0, entrambe computabili. Non è noto di quale delle due funzioni si tratti, quindi non si è in grado di calcolarla, pur sapendo che è calcolabile.

Questo risultato ha una portata molto generale. Ogni volta che il problema ha una risposta sempre (cioè indipendentemente dall'input) positiva o negativa, non può essere indecidibile, poiché è equivalente al calcolo della funzione costante 1 o della funzione costante 0. Ciò non implica, tuttavia, la possibilità di trovare effettivamente una soluzione. Il problema della irrisolubilità algoritmica può presentarsi solo quando si devono prendere in esame infiniti casi (argomenti di una funzione, stringhe di linguaggi).

- Si consideri la funzione $g : \mathbf{N} \mapsto \{0, 1, \ldots, 9\}$ definita come $g(x) = x$-esima cifra nella rappresentazione decimale del numero reale π. Chiediamoci se g è una funzione calcolabile. A questo proposito l'analisi numerica fornisce algoritmi che calcolano una approssimazione decimale di π, in modo tale che la x-esima cifra di π sia estratta, per ogni intero x. La funzione g risulta, pertanto, computabile.

- Esaminiamo ora la funzione

$$f(x) = \begin{cases} 1 & \text{se esiste una sequenza di esattamente } x \text{ "5"} \\ & \text{consecutivi nell'espansione decimale di } \pi \\ 0 & \text{altrimenti} \end{cases}$$

  In questo caso l'approccio di prima non funziona: possiamo calcolare un numero n comunque grande di cifre di π e possono verificarsi solo due situazioni: o si è trovata

una sequenza di esattamente x 5 consecutivi oppure non la si è trovata. Nel primo caso si può concludere che $f(x) = 1$ ma nell'altro caso non si può trarre nessuna conclusione.

Si noti che la definizione di f è perfettamente rigorosa dal punto di vista matematico. Soltanto questa definizione non è algoritmica e quindi non si può invocare la tesi di Church per dire che f è calcolabile. Ciò non ci consente comunque di concludere che f sia non computabile. Può accadere che venga dimostrato un teorema secondo cui ogni sequenza finita di cifre apparirebbe almeno una volta nell'espansione di π. In tal caso avremmo $f(x) = 1$ per ogni x e quindi $f(x)$ sarebbe computabile. Molto tempo fa lo stesso tipo di incertezza esisteva per la funzione h definita da:

$$h(x) = \begin{cases} 1 & \text{se esiste una sequenza di esattamente } x \text{ “5” consecutivi nell'espansione decimale di } \frac{741}{2339} \\ 0 & \text{altrimenti} \end{cases}$$

che oggi è noto essere computabile.

- Consideriamo un ultimo esempio. Sia h la funzione definita come:

$$h(x) = \begin{cases} 1 & \text{se esiste una sequenza di almeno } x \text{ “5” consecutivi nell'espansione decimale di } \pi \\ 0 & \text{altrimenti} \end{cases}$$

Dimostriamo subito che $h(x)$ è calcolabile. Infatti nella espansione decimale di π ci sarà almeno una sequenza di 5 consecutivi. Abbiamo due casi: o esiste una sequenza di 5 infinita, oppure ogni sequenza di 5 è finita. Nel primo caso $h(x)$ è una funzione del tipo: $h(x) = 1$. Nel secondo caso, invece, sia k la lunghezza massima di una sequenza di 5: h è la funzione

$$h(x) = \begin{cases} 1 & \text{se } x < k \\ 0 & \text{se } x \geq k \end{cases}$$

In entrambi i casi $h(x)$ è chiaramente calcolabile, qualunque sia k. Tuttavia si noti che non si sa quale dei due casi sussiste, né, qualora sussista il secondo, quale sia il valore giusto di k.

4.8 Decidibilità di un insieme

4.8.1 Insiemi decidibili e semidecidibili

In questa sezione vedremo come sia possibile applicare alla teoria degli insiemi i risultati ottenuti per le funzioni calcolabili. Focalizziamo la nostra attenzione sui sottoinsiemi di $\mathbf{N}$. Sia I un sottoinsieme di $\mathbf{N}$.

Definizione 40. *Un insieme I è detto decidibile (o ricorsivo) se e solo se la sua funzione caratteristica c_I è totale e calcolabile. La funzione $c_I : \mathbf{N} \mapsto \{0, 1\}$ è definita come:*

$$c_I(x) = \begin{cases} 1 & \textit{se } x \in I \\ 0 & \textit{altrimenti} \end{cases}$$

Definizione 41. *Un insieme I è detto semidecidibile (o ricorsivamente enumerabile) se e solo se è vuoto oppure è il codominio di una funzione totale e calcolabile. Cioè se $I = \emptyset$ oppure se esiste una $g(x)$ totale e calcolabile tale che $I = C(g) = \{z \mid z = g(x), x \in \mathbf{N}\}$ ($C(g)$ denota il codominio di g)*

Per un insieme decidibile I è possibile decidere algoritmicamente se un numero naturale x appartiene o meno a I. Infatti una MT che sia in grado di implementare la funzione caratteristica di I, fornirà una risposta alla domanda $x \in I$, per ogni x. Un tipico problema di appartenenza consiste nello stabilire se una determinata stringa appartenga o meno ad un certo linguaggio.

Nel caso di un insieme semidecidibile I è possibile soltanto enumerare i suoi elementi per mezzo di un algoritmo. Possiamo dare in ingresso alla funzione g (il cui codominio è appunto I), in successione, i numeri naturali partendo da 0 e esaminare uno alla volta i risultati ottenuti. Se $x \in I$ allora x prima o poi comparirà nella enumerazione: basta "attendere" un tempo sufficientemente lungo per vedere comparire x nella enumerazione e concludere che $x \in I$. Se invece $x \notin I$, x non comparirà mai nella enumerazione: potremmo "attendere" un tempo t arbitrariamente lungo ma qualora x non comparisse non saremmo autorizzati a dire che x non apparirà più (potrebbe infatti saltar fuori in un istante t' successivo a t).

Esaminiamo ora alcune proprietà caratteristiche degli insiemi decidibili e semidecidibili.

- Un insieme ricorsivo è ricorsivamente enumerabile. Infatti, sia c_I la funzione caratteristica di un insieme ricorsivo I. Se I è vuoto allora è ricorsivamente enumerabile; altrimenti I contiene un elemento k; definiamo una funzione g nella maniera seguente:

  $$g(x) = \begin{cases} x & \text{se } c_I(x) = 1 \\ k & \text{altrimenti} \end{cases}$$

  Risulta che g è totale e calcolabile (perché c_I è calcolabile) e inoltre $C(g) = I$, quindi I è ricorsivamente enumerabile.

- Un insieme I è ricorsivo se e solo se sia I sia il suo complemento $\bar{I} = \mathbf{N} \setminus I$ sono ricorsivamente enumerabili.

  Dimostriamo la parte del "se". Partiamo dall'ipotesi che I e $\bar{I}$ siano ricorsivamente enumerabili. Questo vuol dire che esistono g e $\bar{g}$ totali e calcolabili tali che $C(g) = I$ e $C(\bar{g}) = \bar{I}$. Definiamo una funzione $f(x)$ nel modo seguente: dato un valore $x \in \mathbf{N}$ $f(x)$ genera i valori: $g(0), \bar{g}(0), g(1), \bar{g}(1), \ldots, x$ finché il valore x non appare nella sequenza. Ciò accadrà prima o poi, in quanto $C(g) \cup C(\bar{g}) = \mathbf{N}$. Allora se x è di posto pari nella sequenza, poniamo $f(x) = 0$ e se di posto dispari poniamo $f(x) = 1$. Quindi $f(x)$ è totale e calcolabile ed è la funzione caratteristica di I. Perciò I è ricorsivo. Dimostriamo la parte del "solo se". Se I è un insieme ricorsivo anche $\bar{I}$ è un insieme ricorsivo. Infatti la funzione caratteristica $c_{\bar{I}}$ definita come:

$$c_{\bar{I}}(x) = \begin{cases} 1 & \text{se } c_I(x) = 0 \\ 0 & \text{se } c_I(x) = 1 \end{cases}$$

è totale e calcolabile.

Segue, dall'affermazione precedentemente dimostrata, che sia I che $\bar{I}$ sono ricorsivamente enumerabili.

4.8.2 Esempi di insiemi decidibili, semidecidibili, nonsemidecidibili

Gli insiemi più comuni come quello dei numeri pari, quello dei numeri dispari, ecc., sono decidibili.

Gli insiemi finiti, come pure gli insiemi il cui complemento è finito, sono decidibili.

Dalla definizione di insieme semidecidibile deduciamo che la cardinalità della famiglia F_I degli insiemi semidecidibili è minore o uguale alla cardinalità dell'insieme F_C delle funzioni calcolabili. Possiamo infatti associare ad ogni insieme I la funzione totale e calcolabile f tale che $C(f) = I$ individuando una corrispondenza (univoca) tra F_I e F_C: di conseguenza vale $|F_I| \leq |F_C|$). Sappiamo inoltre che $|F_C| < |2^{\mathbf{N}}|$, ovvero il numero delle funzioni calcolabili è inferiore al numero delle funzioni da $\mathbf{N}$ in $\mathbf{N}$ che è pari a $|2^{\mathbf{N}}|$. Ma $|2^{\mathbf{N}}|$ è anche la cardinalità della famiglia dei sottoinsiemi di $\mathbf{N}$. Pertanto, concludiamo che gli insiemi ricorsivamente enumerabili sono un sottoinsieme dell'insieme delle parti di $\mathbf{N}$. Esistono quindi insiemi che non sono nemmeno semidecidibili. Anzi sono la quasi totalità.

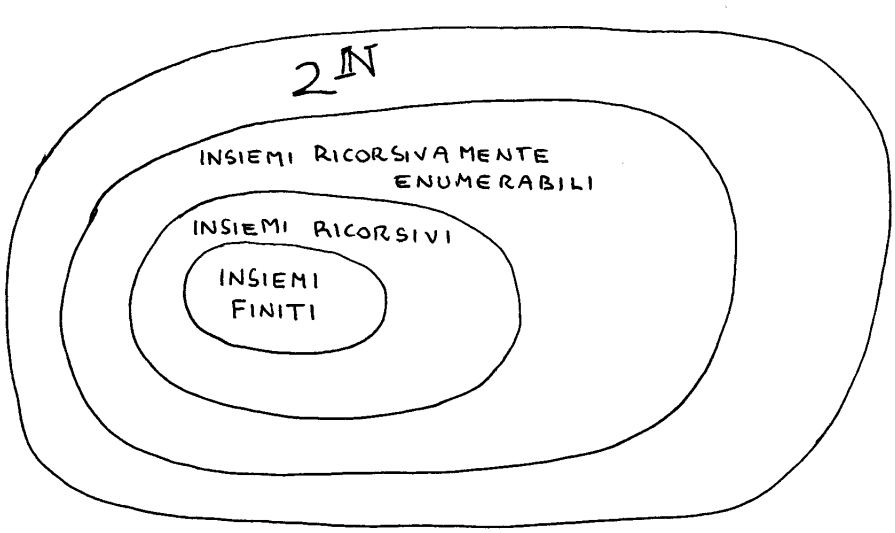


Figura 4.8: Insiemi di numeri naturali

4.9 Teoremi di Kleene e Rice

In questa sezione vengono presentati due teoremi fondamentali della teoria della calcolabilità: il teorema di Kleene ed il teorema di Rice.

Il teorema di Kleene ci dice che, data una qualunque funzione totale e computabile t, è sempre possibile trovare un intero p tale che $\varphi_p = \varphi_{t(p)}$ La funzione φ_p è chiamata punto fisso di t.

Si noti che $\varphi_p = \varphi_{t(p)}$ non implica $p = t(p)$. Il caso $p = t(p)$ rappresenta il caso banale in cui t è la funzione identità che non altera i dati presi in input e li restituisce tali e quali. La scrittura $\varphi_p = \varphi_{t(p)}$ va interpretata così: t è un algoritmo (una MT, ovvero è una funzione φ_k della enumerazione delle MT, per qualche k) che opera una trasformazione dei dati in ingresso. C'è almeno un input p per t tale che il risultato della computazione di t su p è un indice $\bar{p}$ della enumerazione delle MT tale che $\varphi_p = \varphi_{\bar{p}}$. L'indice p è detto punto fisso per t; p è un intero e corrisponde ad una MT della enumerazione, vale a dire ad un algoritmo, ad un programma. Il teorema di Kleene afferma che le funzioni totali e calcolabili ammettono sempre un programma invariante, cioè per ogni trasformazione effettiva t di programmi in programmi esiste sempre un programma p tale che $t(p)$ è equivalente a p nel senso che entrambi calcolano la stessa funzione parziale.

Teorema 42 (Kleene). *Sia t una qualunque funzione totale e computabile. Allora è sempre possibile trovare un intero p tale che $\varphi_p = \varphi_{t(p)}$*

DIMOSTRAZIONE. Sia u un intero (cioè un qualsiasi indice di MT). Partendo da u costruiamo una MT di indice $g(u)$ che implementi il seguente algoritmo applicato ad un valore x dato:

- si calcoli $z = \varphi_u(u)$
- se la computazione di $\varphi_u(u)$ termina, si calcoli $\varphi_z(x)$

Poiché la procedura precedente è effettiva, esiste, per la tesi di Church, una MT in grado di implementarla; perciò $g(u)$ è una funzione totale (esiste una MT opportuna per ogni u, anche nel caso in cui $\varphi_u(u)$ non è definita) e calcolabile (si può costruire effettivamente). Riassumendo:

$$\varphi_{g(u)} = \begin{cases} \varphi_{\varphi_u(u)}(x) & \text{se } \varphi_u(u) \text{ è definita} \\ \text{indefinita} & \text{se } \varphi_u(u) \text{ non è definita} \end{cases}$$

Si noti che g è totale, mentre $\varphi_{g(u)}$ non lo è necessariamente. Ora, sia t una qualsiasi funzione calcolabile e totale. Allora la funzione $t \circ g$ ottenuta per composizione è anch'essa totale e calcolabile. Sia allora v un indice di MT che calcola $t \circ g$ e si ponga $u = v$ nella costruzione precedente. Si noti che $\varphi_v(v) = t \circ g(v)$ è definita essendo $t \circ g$ totale. Perciò abbiamo $\varphi_{g(v)} = \varphi_{\varphi_v(v)} = \varphi_{t \circ g(v)}$ e quindi $p = g(v)$ è un punto fisso di t. □

Teorema 43 (Rice). *Sia A un qualsiasi insieme di funzioni computabili. L'insieme $I = \{x \mid \varphi_x \in A\}$ è ricorsivo se e solo se A è vuoto o A è l'insieme di tutte le funzioni computabili.*

DIMOSTRAZIONE. Sia data una qualunque enumerazione effettiva delle MT. Supponiamo per assurdo che l'insieme I sia ricorsivo e che A non coincida con l'insieme vuoto né con l'insieme di tutte le funzioni calcolabili. Esiste allora un modo per decidere effettivamente per ogni x se questo appartiene a I oppure no. Si può quindi trovare, scorrendo la lista degli indici delle MT, il primo $i \in I$ per cui (α): $\varphi_i \in A$ e il primo $j \notin I$ per cui (β): $\varphi_i \notin A$. Si consideri la funzione caratteristica g di I:

$$g(x) = \begin{cases} 0 & \text{se } \varphi_x \notin A \\ 1 & \text{se } \varphi_x \in A \end{cases} \qquad (\gamma)$$

Essa è per ipotesi totale e calcolabile. Anche la funzione h, definita come segue, è totale e calcolabile:

$$h(x) = \begin{cases} i & \text{se } \varphi_x \notin A \\ j & \text{se } \varphi_x \in A \end{cases}$$

Ora, per il teorema di Kleene, esiste un $\bar{x}$ tale che: (δ): $\varphi_{h(\bar{x})} = \varphi_{\bar{x}}$. Supponiamo ora $h(\bar{x}) = i$. Abbiamo allora, per (γ), $\varphi_{\bar{x}} \notin A$, mentre abbiamo per (δ) $\varphi_{\bar{x}} = \varphi_j$ e per (α), $\varphi_j \in A$: una contraddizione. Si supponga invece che $h(\bar{x}) = j$; allora per (δ) $\varphi_{\bar{x}} \in A$. Tuttavia per (δ) $\varphi_{\bar{x}} = \varphi_j$ e per (β) $\varphi_j \notin A$; di nuovo una contraddizione. □

Il teorema di Rice rappresenta un fortissimo risultato negativo. Esso afferma sostanzialmente che non saremo mai in grado di caratterizzare insiemi di programmi partendo da proprietà delle funzioni calcolate, a meno che tale proprietà non sia veramente banale. Quindi ad esempio il teorema di Rice implica che non si può mai in generale decidere se un programma Pascal calcola o meno una certa funzione, se due generici programmi sono equivalenti, se esistono particolari valori dei dati iniziali che mandano in ciclo un generico programma.

Capitolo 5

Elementi di Teoria della Complessità Computazionale

Dalla teoria della calcolabilità abbiamo appreso che se un problema può essere risolto da una MT allora è possibile scrivere un programma per calcolatore che lo risolva.

Tuttavia, un problema teoricamente decidibile può essere praticamente intrattabile se non è possibile ottenere la soluzione in un tempo e/o spazio di memoria "ragionevole".

L'intrattabilità costituisce una limitazione notevole per le applicazioni pratiche. Esistono molti problemi interessanti che pur essendo decidibili si rivelano in pratica intrattabili.

Il concetto di intrattabilità è strettamente correlato con quello di complessità: un problema è intrattabile quando la sua complessità è troppo elevata. Misurare la complessità di un algoritmo significa valutare il suo costo di esecuzione.

Prima di affrontare gli argomenti della teoria della complessità introduciamo alcune nozioni sui grafi, in quanto la maggior parte degli esempi sarà costituita da problemi sui grafi, e le notazioni O, Ω e Θ che sono gli strumenti base dell'analisi asintotica delle misure di complessità.

5.1 Preliminari

5.1.1 Nozioni sui grafi

Un grafo è una rappresentazione grafica di una relazione binaria.

Una relazione binaria $\mathcal{R}$ su un insieme A è un sottoinsieme del prodotto cartesiano $A \times A$: $\mathcal{R} \subseteq A \times A$. Una relazione binaria $\mathcal{R}$ può essere definita in maniera implicita, mediante un predicato P che definisce le coppie di elementi che sono in relazione tra loro, oppure in maniera esplicita elencando gli elementi di $\mathcal{R}$. Ad esempio, sia A l'insieme {Anna, Marcello, Franco, Teresa, Luigi} ed $\mathcal{R}_1$ la relazione definita dal predicato P_1: "a, b elementi di A sono in relazione ($(a, b) \in \mathcal{R}_1$) se e solo se a è più giovane di b". Sapendo che Anna e Marcello hanno 24 anni, Franco ne ha 25, Teresa e Luigi ne hanno 72 la relazione $\mathcal{R}_1$ è così composta:

$\mathcal{R}_1$ = {(Anna, Franco), (Anna, Teresa), (Anna, Luigi), (Marcello, Franco), (Marcello, Teresa), (Marcello, Luigi), (Franco, Teresa), (Franco, Luigi)}.

Sia P_2 il predicato: "a, b elementi di A sono in relazione ($(a, b) \in \mathcal{R}_2$) se e solo se a e b hanno la stessa età". La relazione $\mathcal{R}_2$ definita dal predicato P_2 è la seguente: $\mathcal{R}_2 = \{$(Anna, Marcello), (Marcello, Anna), (Luigi, Teresa), (Teresa, Luigi)$\}$.

Sia $\mathcal{R}_3$ la relazione sull'insieme A data in maniera esplicita come segue: $\mathcal{R}_3 = \{$(Anna, Teresa),(Teresa, Anna)$\}$. Si provi ad immaginare un predicato che definisce in maniera implicita $\mathcal{R}_3$.

Sia $\mathcal{R}$ una relazione binaria su un insieme A. Per rappresentare graficamente $\mathcal{R}$ si associ ad ogni elemento di A un punto del piano ed ad ogni coppia $(a, b) \in \mathcal{R}$ un segmento orientato che parte dal punto a e arriva nel punto b. Ad esempio, il seguente grafo rappresenta la relazione $\mathcal{R} = \{$(Marcello, Anna), (Marcello, Luigi), (Teresa, Franco), (Franco, Teresa)$\}$.

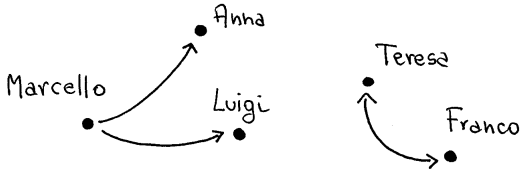


Figura 5.1: esempio di relazione

Diamo la definizione formale di grafo:

Definizione 44. *Un grafo è una coppia ordinata $G = (V, E)$ dove V è un insieme finito di punti ed E è un insieme di segmenti orientati aventi estremi in V.*

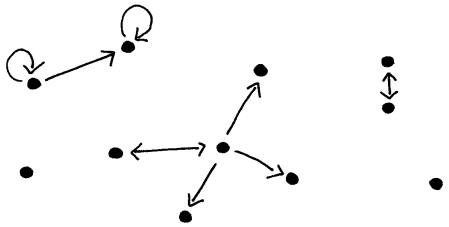


Figura 5.2: esempio di grafo

TERMINOLOGIA E NOTAZIONI

Dato un grafo $G = (V, E)$, l'insieme dei punti, denotato con $V(G)$, è detto anche insieme dei vertici o dei nodi. L'insieme dei segmenti orientati E, denotato con $E(G)$, è detto anche insieme delle linee o degli archi o spigoli. Nel seguito utilizzeremo prevalentemente i termini nodi e archi. Un arco dal nodo u al nodo v viene denotato con uv.

Un grafo $G = (V, E)$ si dice:

- riflessivo se $\forall\ u \in\ V(G),\ uu \in\ E(G)$
- simmetrico se $\forall\ u, v \in\ V(G),\ uv \in\ E(G) \leftrightarrow\ vu \in\ E(G)$
- transitivo se $\forall\ u, v, z \in\ V(G),\ uv, vz \in\ E(G) \rightarrow\ uz \in\ E(G)$

Un arco del tipo uu viene detto autoanello (o cappio).

Figura 5.3: un autoanello

Se il grafo è simmetrico gli archi non sono orientati.

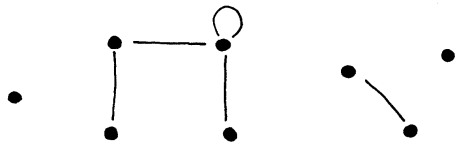


Figura 5.4: grafo simmetrico

Nel seguito avremo a che fare con grafi simmetrici privi di autoanelli.
Due nodi $u, v \in\ V(G)$ si dicono adiacenti se l'arco $uv \in\ E(G)$.
Due archi e e f si dicono adiacenti se hanno un nodo in comune.

ISOMORFISMO

Una relazione binaria individuata da un particolare grafo può essere rappresentata graficamente in molti modi diversi variando la disposizione dei nodi del grafo di partenza. Quando due grafi "apparentemente" diversi denotano la stessa relazione si dicono *isomorfi*.

Diamo la definizione formale di isomorfismo tra grafi.

Definizione 45. *Due grafi G_1 e G_2 si dicono isomorfi se esiste una biiezione $f : V(G_1) \mapsto V(G_2)$ tale che $uv \in\ E(G_1)$ se e solo se $f(u)f(v) \in\ E(G_2)$.*

GRAFO COMPLEMENTARE, COMPLETO E VUOTO

Sia $G = (V, E)$ un grafo. Il grafo $\bar{G} = (V, (V \times V) \setminus E)$ si dice grafo complementare di G.

Il grafo $G = (V, V \times V)$ si dice grafo completo mentre il suo complementare $\bar{G} = (V, \emptyset)$ si dice grafo vuoto.

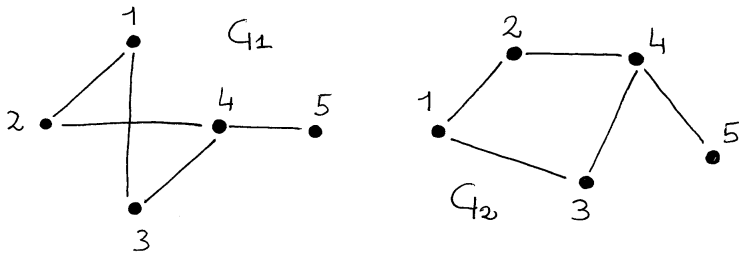


Figura 5.5: grafi isomorfi

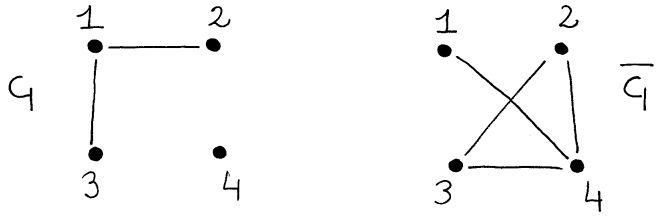


Figura 5.6: grafo complementare

I grafi completi (così come i grafi vuoti) con n nodi sono tutti isomorfi tra loro. Indichiamo con K_n un qualsiasi grafo completo con n nodi e con $\bar{K}_n$ un qualsiasi grafo vuoto con n nodi.

Diremo *nullo* il grafo tale che $V(G) = \emptyset$ (e di conseguenza $E(G) = \emptyset$).

ESERCIZIO

I grafi K_n e $\bar{K}_n$ sono entrambi simmetrici e transitivi. Si dimostri tale affermazione.

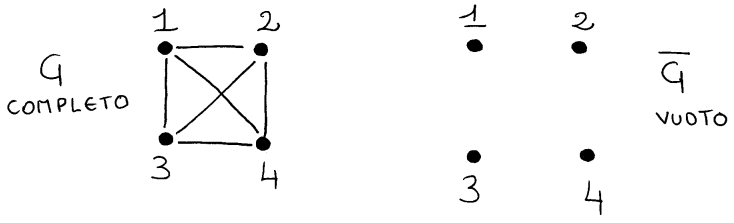


Figura 5.7: grafo completo e grafo vuoto

SOTTOGRAFI

Sia G un grafo e S un sottoinsieme di $V(G)$. Sia F un sottoinsieme di $E(G)$ tale che ogni arco in F ha entrambi gli estremi in S. Il grafo $H = (S, F)$ viene detto sottografo di G.

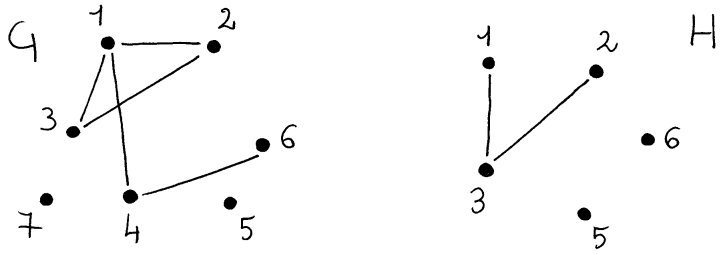


Figura 5.8: sottografo

Sia G un grafo e S un sottoinsieme di vertici di G. Il sottografo H indotto da S ha come nodi l'insieme S e come archi tutti gli archi di G che hanno entrambi estremi in S.

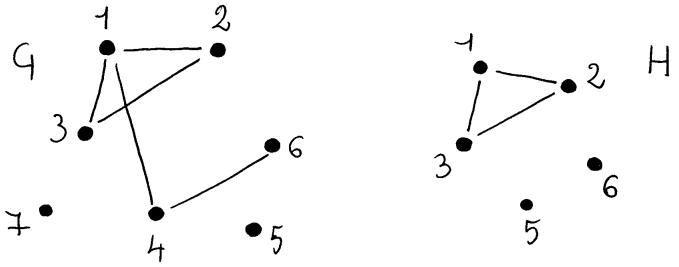


Figura 5.9: sottografo indotto

Sia G un grafo e F un sottoinsieme di archi di G. Il sottografo H indotto da F ha come archi l'insieme F e come nodi i nodi di G che sono estremi di qualche arco in F.

CONNESSIONE

Un grafo G si dice connesso se ogni coppia $u, v \in V(G)$ è "collegata" tramite archi.

I sottografi di G che godono della proprietà di connessione e che sono massimali rispetto a tale proprietà si dicono *componenti connesse*.

In maniera informale potremmo dire che un grafo G è connesso se è composto da un'unica componente connessa.

CLIQUE, INSIEME STABILE, VERTEX COVER

Sia G un grafo. Un sottoinsieme Q di $V(G)$ si dice *clique* se Q induce un sottografo

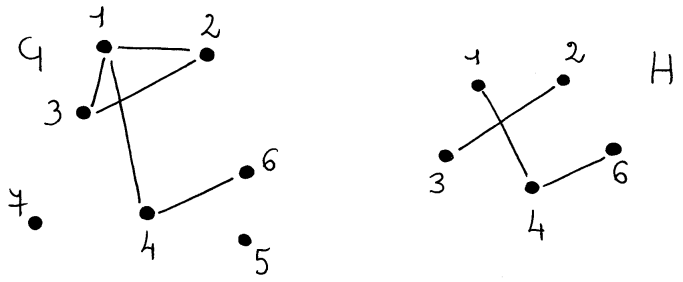


Figura 5.10: sottografo indotto da un insieme di archi

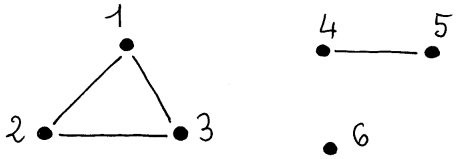


Figura 5.11: grafo non connesso

completo di G. Un sottoinsieme S di $V(G)$ si dice *insieme stabile* (o *indipendente*) se S induce un sottografo vuoto di G.

Si noti che l'insieme $S = \{3, 5, 6\}$ nel grafo in figura 5.14 è un insieme stabile massimale: se aggiungiamo un altro nodo non abbiamo più un insieme stabile. S è anche massimo: ogni altro insieme stabile avrà al più lo stesso numero di elementi di S.

L'insieme $Q = \{1, 2\}$ non è massimale (e quindi neanche massimo): possiamo aggiungere l'elemento 3 e ottenere una clique più grande.

Un *vertex cover* è un insieme di nodi X tale che per ogni arco $uv \in E(G)$ si ha che $u \in X$ oppure $v \in X$. In altre parole, X è una copertura degli archi di G. Nel grafo rappresentato in figura 5.15, l'insieme di nodi $X = \{1, 3, 4, 5\}$ è un vertex cover.

Teorema 46. *L'insieme dei nodi di qualunque grafo G può essere partizionato in due sottoinsiemi X e S tale che X è un vertex cover e S è un insieme stabile.*

DIMOSTRAZIONE. Sia X un vertex cover di G e supponiamo per assurdo che $S = V(G) \setminus X$

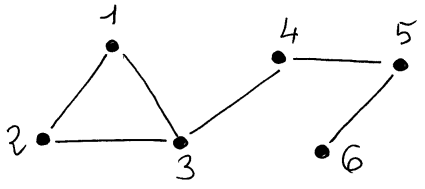


Figura 5.12: grafo connesso

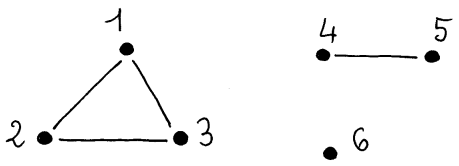


Figura 5.13: grafo con 3 componenti connesse

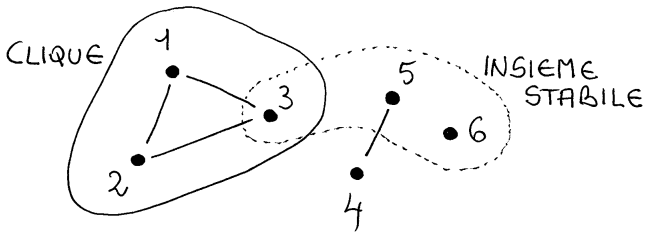


Figura 5.14: una clique e un insieme stabile

non sia stabile. Ma allora S contiene due nodi u e v adiacenti, e poiché u e v non appartengono a X sia ha che l'arco uv non è coperto da alcun nodo di X. Dunque X non copre gli archi di G. Ciò contraddice l'ipotesi che X sia un vertex cover. □

Si può dimostrare facilmente che per ogni grafo $G = (V, E)$ ed un sottoinsieme di nodi $V' \subseteq V$, le seguenti affermazioni sono equivalenti:

- V' è una clique per G
- V' è un insieme stabile per $\bar{G}$
- $V \setminus V'$ è un vertex cover per $\bar{G}$

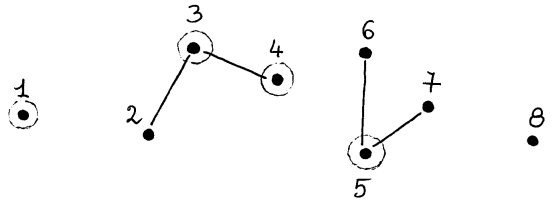


Figura 5.15: un vertex cover

5.1.2 Notazioni O,Ω,Θ

Introduciamo tre notazioni che vengono usate nello sviluppo della teoria della complessità (e ancor più nello studio degli algoritmi) allo scopo di caratterizzare l'andamento *asintotico* (ovvero l'andamento per valori di ingresso molto grandi) di una funzione rispetto ad un altra funzione.

Definizione 47. *Sia $f : \mathbf{N} \mapsto \mathbf{N}$. Definiamo l'insieme di funzioni*

$$O(f(n)) = \{g : \mathbf{N} \mapsto \mathbf{N} \mid \exists c_1 \in \boldsymbol{R}^+, n_0 \in \mathbf{N} \wedge \forall n \geq n_0,\ g(n) \leq c_1 f(n)\}.$$

L'insieme $O(f(n))$ è dunque l'insieme delle funzioni che risultano essere "asintoticamente più piccole" della funzione f, dove con "asintoticamente più piccole" intendiamo essenzialmente 2 concetti:

1. le funzioni dell'insieme $O(f(n))$ sono minori di $f(n)$ a meno di una costante moltiplicativa c, cioè una funzione $g \in O(f(n))$ può anche assumere valori maggiori di quelli assunti da f, purchè si possa scegliere una costante che moltiplicata per i valori assunti da f, renda tali valori maggiori dei corrispondenti valori in g

2. le funzioni dell'insieme $O(f(n))$ sono minori di $f(n)$ (a meno di una costante moltiplicativa c) non per tutti i valori di n, ma solo per valori maggiori di un certo punto n_0; in altre parole sono minori solo "da un certo punto in poi".

Illustriamo questa definizione considerando per semplicità due rette. La funzione $g(n) = 4n+100$ risulta maggiore, per ogni valore di n della funzione $f(n) = 2n$. Eppure $g \in O(f(n))$. Infatti, se considero la costante $c_1 = 5$ e il punto $n = 100$, trovo $g(100) = 500$ e $c_1 f(100) = 1000$; e come si può verificare, per tutti gli $n > 100$, risulta sempre $g(n) \leq c_1 f(n)$. Notate che per valori di n piccoli, ad esempio $n = 2$, si ha $g(n) > c_1 f(n)$; ma questo non importa, perché come detto ci interessa quello che succede per valori grandi di n, da un punto n_0 in poi.

D'altra parte, si consideri la funzione $h(n) = \frac{1}{100}n^2$. Tale funzione non appartiene all'insieme $O(f(n))$. Per $n = 10$ risulta $h(10) < f(10)$. Per $n = 100$, di nuovo $h(100) < f(100)$. Ma per $n \geq 1000$, risulta decisamente $h(n) \geq f(n)$. Certo, possiamo sempre utilizzare la costante c_1 per aumentare il valore di f, così come abbiamo fatto quando abbiamo confrontato g con f. Se considero $c_1 = 10$, risulta $h(1000) < c_1 f(1000)$. Ma la costante c_1 può solo ritardare la sorte del confronto: infatti $h(10000) > c_1 f(10000)$. Per quanto grande scelga c_1, da un certo punto in poi h supererà il prodotto $c_1 f(n)$.

In maniera analoga definiamo l'insieme delle funzioni "asintoticamente più grandi" e quello delle funzioni "asintoticamente equivalenti" rispetto alla funzione f.

Definizione 48. *Sia $f : \mathbf{N} \mapsto \mathbf{N}$. Definiamo l'insieme di funzioni*

$$\Omega(f(n)) = \{g : \mathbf{N} \mapsto \mathbf{N} \mid \exists c_2 \in \boldsymbol{R}^+, n_0 \in \mathbf{N} \wedge \forall n \geq n_0,\ g(n) \geq c_2 f(n)\}.$$

Definizione 49. *Sia $f : \mathbf{N} \mapsto \mathbf{N}$. Definiamo l'insieme di funzioni*

$$\Theta(f(n)) = \{g : \mathbf{N} \mapsto \mathbf{N} \mid \exists c_1, c_2 \in \boldsymbol{R}^+, n_0 \in \mathbf{N} \wedge \forall n \geq n_0,\ c_2 f(n) \leq g(n) \leq c_1 f(n)\}.$$

5.2 Concetti di base

5.2.1 Problemi e linguaggi

I problemi di calcolo che sorgono nell'ambito delle varie discipline scientifiche sono alquanto diversi tra loro. Per studiare la complessità di un determinato problema (o meglio di un algoritmo che risolva un determinato problema) abbiamo bisogno di una misura di complessità. Il primo passo è quello di stabilire una misura di complessità comune a problemi aventi caratteristiche diverse: ciò viene fatto stabilendo una corrispondenza tra problemi e linguaggi.

Un problema (del tipo di quelli studiati in questo corso) può essere definito come un insieme di istanze, e un'istanza è un esempio specifico di un particolare problema. Le istanze stanno ai problemi come gli oggetti stanno alle classi. Una classe definisce quali sono le caratteristiche degli oggetti ad essa appartenenti; gli oggetti costituiscono la rappresentazione "concreta" di quanto è definito nella classe. Ad esempio, la classe dei *grafi aciclici* è definita come l'insieme di tutti quei grafi i cui archi non formano cicli. Alcune istanze di tale classe sono le seguenti:

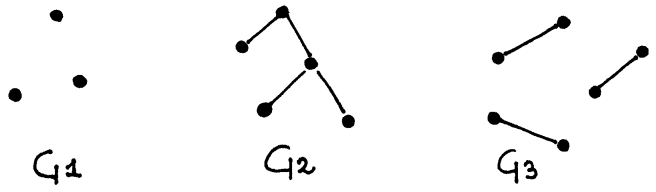


Figura 5.16: grafi aciclici

Consideriamo una classe particolare di problemi: quella dei problemi decisionali. Un problema decisionale (detto anche *in forma di riconoscimento*) ammette due soli tipi di risposta: SI (esiste una soluzione effettiva) o NO (non esiste una soluzione effettiva).

Se un'istanza di un problema decisionale ammette risposta SI, diremo che è un'istanza affermativa; in caso contrario, diremo che è un'istanza negativa.

ESEMPIO: BISACCIA

Dato un insieme di oggetti aventi ognuno un costo ed un profitto prefissati, il problema consiste nel determinare, se esiste, un sottoinsieme di questi oggetti il cui profitto sia al di sopra di un dato valore ed il cui costo sia al di sotto di una quantità fissata.

L'istanza di BISACCIA è costituita da un insieme finito U, due funzioni $c, p : U \mapsto \mathbf{N}$ e due costanti $k, b \in \mathbf{N}$. Il problema consiste nel determinare un sottoinsieme $U_1 \subseteq U$ tale che $\Sigma_{u \in U_1} p(u) \geq k$ e $\Sigma_{u \in U_1} c(u) \leq b$.

Il problema della bisaccia è di tipo decisionale: esso ammette una risposta affermativa oppure una risposta negativa.

Risulta naturale far corrispondere ad un problema decisionale il problema dell'appartenenza

di una parola ad un dato linguaggio, codificando ogni istanza del problema tramite una parola tale che l'istanza ha risposta SI se e solo se la parola appartiene al linguaggio corrispondente.

ESEMPIO: codifica del problema BISACCIA.

Una possibile codifica del problema BISACCIA si può realizzare codificando in binario il numero di elementi di U, il loro costo, il loro profitto e le costanti k e b.

L'intera istanza può essere codificata in una parola costruita sull'alfabeto $\{0, 1\}$ utilizzando la seguente tecnica:

- ad ogni cifra di ogni numero che rappresenta un dato viene associata la codifica in binario in cui i simboli 0 e 1 vengono duplicati. Ad esempio al numero 2 viene associata la stringa 1100, al numero 3 viene associata la stringa 1111;
- ogni codifica di ciascuna cifra è seguita dalla stringa 01. Il numero 23 viene codificato come segue: 110001111101;
- si usa il simbolo # per separare la codifica dei vari dati.

Con questa tecnica si può realizzare una codifica di un'istanza del problema BISACCIA data da $\langle 3, (2, 5, 7), (5, 5, 2), 7, 6\rangle$ dove 3 è il numero di elementi di U, $(2, 5, 7)$ è il vettore dei costi associati a tali elementi, $(5, 5, 2)$ è il vettore dei profitti, 7 è il profitto minimo e 6 è il costo massimo. La codifica di tale istanza è la seguente:

111101#11000111001101111111101#11001101110011011110001#11111101#11110001

In base alla corrispondenza esistente tra problemi decisionali e linguaggi ha senso parlare di problemi decisionali decidibili, indicando con questo termine i problemi decisionali che possono essere codificati in linguaggi decidibili.

Abbiamo già sottolineato il fatto che la teoria della complessità computazionale tratta problemi decidibili. All'interno di tale classe distinguiamo i problemi "facili" da quelli "difficili".

I problemi decisionali hanno un ruolo molto importante nella teoria della complessità computazionale in quanto costituiscono gli elementi di base di altri problemi più interessanti e complessi chiamati problemi di ottimizzazione.

In termini intuitivi i problemi di ottimizzazione danno per scontato che l'istanza ammette una soluzione ed hanno per obiettivo quello di individuare la soluzione "ottima" ossia la soluzione avente costo ottimo rispetto a una misura prefissata.

ESEMPIO

Il problema di ottimizzazione corrispondente al problema decisionale BISACCIA consiste nel massimizzare il profitto mantenendo il costo al di sotto di una quantità fissata. Tale problema consiste quindi nel trovare un sottoinsieme $U_1 \subseteq U$ che massimizzi la quantità $\Sigma_{u\in\ U_1} p(u)$ e soddisfa il vincolo $\Sigma_{u\in\ U_1} c(u) \leq\ b$.

Com'è ovvio la complessità di un problema di ottimizzazione non può essere inferiore a quella del problema decisionale corrispondente: in effetti il saper calcolare l'ottimo ci consente di verificare se esiste una soluzione avente una misura prefissata. Si potrebbe a questo punto obiettare che non tutti i problemi sono di tipo decisionale: esistono, ad esempio, problemi numerici fondamentali del tipo "dato x, calcolare $f(x)$". Abbiamo visto nella parte di teoria della calcolabilità che a tale problema si può associare il problema del riconoscimento di un linguaggio. In effetti, si può definire un problema decisionale corrispondente al calcolo di f del tipo "dati x e y, è vero che $f(x) = y$?". La complessità di un algoritmo che risolva tale problema non sembra essere inferiore a quella dell'algoritmo che calcola f.

Torniamo ai problemi in forma decisionale e in forma di ottimizzazione.

ESEMPIO: CAMMINO MINIMO

- Forma di ottimizzazione: dato un grafo $G = (N, A, w)$, orientato, pesato sugli archi con valori non negativi e due nodi s e t quanto vale la lunghezza del cammino minimo da s a t?

  Un'istanza di CAMMINO MINIMO è dunque descritta completamente da un grafo, dai suoi pesi, e dalla indicazione di due suoi nodi. L'output di un algoritmo che risolva tale problema sarà costituito dal *valore* della lunghezza minima, z^*.

- Forma decisionale: dato un grafo $G = (N, A, w)$, orientato, pesato sugli archi con valori non negativi, due nodi s e t e un intero positivo k, esiste un cammino di lunghezza $\leq k$?

A prima vista le due versioni dello stesso problema possono sembrare non equivalenti. Se conosciamo il valore z^* della lunghezza del cammino minimo, è evidente che sapremo immediatamente rispondere al problema in forma di riconoscimento, in quanto, se $k \geq z^*$, la risposta sarà SI, e altrimenti sarà NO.

Ma è vero anche il viceversa. Infatti, supponiamo di saper risolvere il problema in forma di riconoscimento, e vogliamo la risposta a quello in forma di ottimizzazione. Consideriamo allora un upper bound U al valore della soluzione ottima (tale upper bound può essere estremamente banale, ad esempio se stiamo cercando il cammino minimo su un grafo possiamo scegliere come U la somma dei pesi di tutti gli archi del grafo). Si può dunque inizialmente risolvere il problema in forma di riconoscimento impostando k al valore $U/2$. Se la risposta è SI, vuol dire che il percorso minimo ha lunghezza non superiore a $U/2$, e dunque torneremo a risolvere un problema di riconoscimento con $k = U/4$. Se la risposta è NO, al passo successivo imposteremo invece $k = (3/4)U$. E così via, sempre dimezzando l'intervallo di valori entro i quali può cadere z^*. Dopo al più un numero di passi pari a $log_2 U$, avremo individuato il valore ottimo z^*. Dunque, se disponiamo di una procedura per la soluzione del problema in forma di riconoscimento, possiamo, attraverso una semplice ricerca binaria, pervenire alla soluzione del problema di ottimizzazione. Quel che è importante, è che se si sa risolvere il problema di riconoscimento in modo *efficiente*, con la stessa efficienza sarà possibile risolvere quello in forma di ottimizzazione. Per questo motivo, di qui in avanti faremo sempre riferimento a problemi in forma di riconoscimento.

5.2.2 Misure di complessità

Ci proponiamo di formulare la misura di complessità di un algoritmo al fine di poter valutare la complessità di uno specifico problema, poiché quest'ultima dipende in effetti dall'esistenza o meno di un algoritmo efficiente (rispetto alla misura di complessità usata) che lo risolve.

Non esiste un'unica misura di complessità: ci sono vari criteri che possono essere presi in considerazione. In generale distinguiamo tra misure di complessità statiche e dinamiche.

Una misura di complessità statica è basata sulle caratteristiche dell'algoritmo e prescinde dai valori dell'input su cui esso opera. Un esempio tipico di misura statica è la dimensione di un algoritmo. Altri esempi sono il massimo numero di cicli e il massimo numero di definizioni ricorsive annidate.

Una misura dinamica tiene conto sia delle caratteristiche dell'algoritmo che dell'input su cui esso opera e dipende quindi dall'andamento di una computazione. Le misure dinamiche esprimono il costo degli algoritmi, al variare della dimensione dei dati, in termini di risorse (tempo e/o spazio) richieste dalla loro esecuzione. Un esempio di misura dinamica è il numero di passi elementari eseguiti durante una computazione: si noti che la definizione precisa di passo elementare dipende dalle caratteristiche del modello di calcolo usato.

Altre misure dinamiche sono il numero di operazioni di un determinato tipo (operazioni aritmetiche, confronti) e spazio di memoria occupato.

Nel seguito faremo riferimento soltanto alle misure di tipo dinamico, tempo e spazio.

ESEMPIO

Il calcolo del fattoriale

- Algoritmo iterativo:

  FACT1(n):
  $k := 1$;
  `if` $n > 0$ `then`
  `for` $i := 1$ `to` n `do` $k := k * i$;
  FACT1 :=k.

- Algoritmo ricorsivo:

  FACT2(n):
  `if` $n = 0$ `then` FACT2 := 1
  `else` FACT2 := FACT2$(n-1) * n$.

- Da un punto di vista statico l'algoritmo ricorsivo è leggermente più vantaggioso di quello iterativo essendo il corpo dell'algoritmo più corto e semplice;

- Da un punto di vista dinamico entrambi gli algoritmi eseguono n operazioni di moltiplicazione, se n è l'intero in input;

- Tuttavia, sebbene secondo le precedenti osservazioni potremmo concludere che i due algoritmi abbiano le stesse caratteristiche di efficienza, analizzando le esigenze di memoria si arriva ad una diversa conclusione:
  - entrambi richiedono una memoria di dimensione pari a $\log_2 n!$ per contenere la rappresentazione binaria del risultato;
  - FACT1 usa solo 2 variabili i e k di dimensione rispettivamente $\log_2 n$ e $\log_2 n!$;
  - FACT2 richiede una memoria maggiore: $\sum_{j=1}^{n} \log_2 j!$ quindi è meno efficiente di FACT1.

Distinguiamo due misure di complessità dinamiche: il costo uniforme e il costo logaritmico. L'approccio basato sul costo uniforme è il più semplice. La misura della complessità temporale consiste nel determinare il numero di istruzioni elementari eseguite nella computazione, e la misura della complessità spaziale consiste nel determinare il numero di variabili usate nella computazione. Questo tipo di misura è semplice da effettuare; tuttavia non è adeguato perché considera unitario il costo di un'operazione aritmetica su due interi indipendentemente dalla loro dimensione.

Quando gli operandi sono interi le cui rappresentazioni binarie sono costituite da centinaia di bit, essi non possono essere memorizzati in un'unica parola macchina (16 o 32 bit). Pertanto gli operandi devono essere memorizzati in più parole macchina e l'esecuzione dell'operazione aritmetica corrisponde all'esecuzione di un programma che realizza l'operazione su interi di grandi dimensioni mediante più operazioni su interi aventi la dimensione di una parola macchina.

Pertanto la misura del costo uniforme è significativa nel caso in cui si possa assumere che durante l'intera computazione tutte le variabili contengono valori la cui rappresentazione binaria è contenuta in una parola macchina.

Un'anomalia caratteristica relativa all'utilizzo della misura del costo uniforme è la seguente. Siano k e $a \geq 2$ due interi positivi la cui dimensione della rappresentazione binaria non eccede la dimensione della parola macchina. Vogliamo calcolare il numero a^{2^k}.

Calcoliamo:

$$a^2 = a \cdot a,\ a^4 = a^2 \cdot a^2,\ \ldots,\ a^{2^k} = a^{2^{k-1}} \cdot a^{2^{k-1}}$$

La complessità spaziale del costo uniforme è 3 perché una variabile addizionale è sufficiente per realizzare la computazione seguente:

$$\text{for } i = 1 \text{ to } k \text{ do } a := a * a$$

La complessità temporale del costo uniforme è $O(k)$ perché vengono eseguite esattamente k moltiplicazioni. In realtà c'è bisogno di almeno 2^k bit per rappresentare il risultato a^{2^k} e per scrivere 2^k bit la macchina deve fare $\Omega(2^k)$ operazioni concernenti parole macchina. C'è pertanto un divario esponenziale tra la complessità temporale e spaziale del costo uniforme e quelle reali.

In una situazione del genere, in cui i valori delle variabili crescono illimitatamente, è più appropriato utilizzare la misura del costo logaritmico secondo la quale il costo di ogni operazione elementare è la somma delle dimensioni delle rappresentazioni binarie degli operandi.

La misura del costo logaritmico è generalmente adottata nell'analisi della complessità degli algoritmi.

Nel seguito faremo per lo più riferimento alla complessità temporale delle computazioni, cioè il costo di impiego della risorsa tempo; la complessità spaziale, vale a dire il costo relativo alla quantità di memoria utilizzata nella computazione, verrà considerata marginalmente.

5.2.3 Complessità temporale e spaziale

Sia A un algoritmo. Per ogni input x, $\text{Time}_A(x)$ denota la complessità (secondo il costo logaritmico) della computazione di A sull'input x e $\text{Space}_A(x)$ denota la complessità spaziale (secondo il costo logaritmico) della computazione di A su x.

In realtà non si considerano Time_A e Space_A come le funzioni che abbiamo appena definito: il numero di possibili input può essere considerevolmente grande e valutare la complessità temporale e spaziale per ognuno di essi sarebbe un compito assai arduo. Di conseguenza la complessità viene definita come funzione della dimensione dell'input e si è interessati alla crescita asintotica di tale funzione.

Definizione 50. *Sia A un algoritmo. La complessità temporale di A (nel caso pessimo) è una funzione $Time_A : \mathbf{N} \mapsto \mathbf{N}$ definita come:*

$$Time_A(n) = \max\{Time_A(x), \text{ per ogni possibile input } x \text{ di dimensione } n\}$$

Definizione 51. *La complessità spaziale di A (nel caso pessimo) è una funzione $Space_A : \mathbf{N} \mapsto \mathbf{N}$ definita come:*

$$Space_A(n) = \max\{Space_A(x), \text{ per ogni possibile input } x \text{ di dimensione } n\}$$

L'analisi della complessità del caso pessimo risulta poco adeguata quando un algoritmo si comporta in maniera non uniforme su input di uguale lunghezza. In questo caso si può ricorrere all'analisi del caso medio che consiste nel determinare la complessità media per tutte le istanze di input di lunghezza n.

In questa sede ci limitiamo a considerare l'analisi del caso pessimo: il nostro scopo è quello di definire dei limiti asintotici alla complessità degli algoritmi che risolvono problemi *difficili*.

Definizione 52. *Sia U un problema risolvibile algoritmicamente e siano f, g due funzioni da $\mathbf{N}$ in $\boldsymbol{R}^+$.*

*Diciamo che $O(g(n))$ è una delimitazione superiore (*upper bonud*) alla complessità temporale di U se **esiste** un algoritmo A che risolve U, tale che $Time_A(n) \in O(g(n))$.*

*Diciamo che $\Omega(f(n))$ è una delimitazione inferiore (*lower bound*) alla complessità temporale di U se, **per ogni** algoritmo B che risolve U, risulta $Time_B(n) \in \Omega(f(n))$.*

Un algoritmo C è ottimale per il problema U se $Time_C(n) \in O(g(n))$ e $\Omega(g(n))$ è un lower bound per la complessità temporale di U.

Per stabilire un upper bound per la complessità di un problema U è sufficiente trovare un algoritmo che risolve U.

Stabilire un lower bound non banale alla complessità di U è un compito assai arduo perché significa dimostrare che ogni possibile algoritmo che risolve U, non solo quelli noti ma anche quelli che verranno, ha complessità temporale almeno $\Omega(f(n))$ per qualche f. Si tratta in effetti di una prova di non esistenza poiché bisogna dimostrare che non esiste un algoritmo che risolva U e che abbia una complessità temporale asintotica minore di $f(n)$.

A conferma di ciò si consideri che si è a conoscenza di centinaia di problemi per i quali la complessità del miglior algoritmo conosciuto è esponenziale rispetto alla dimensione dell'input e non è noto alcun lower bound superlineare come ad esempio $\Omega(n \log n)$.

Pertanto si congettura che per molti di questi problemi non esiste alcun algoritmo che li risolva in tempo polinomiale (rispetto alla dimensione dell'input), ma non si è in grado di dimostrare che per risolverli c'è bisogno di una quantità di tempo superiore a $O(n)$.

Ciò che risulta difficile è dimostrare che certi problemi sono *intrinsecamente difficili* e, di conseguenza, che nessuno mai potrà inventare un algoritmo che li risolva in maniera efficiente.

5.3 La classe P

Lo scopo principale della teoria della complessità è classificare i problemi algoritmici a seconda della difficoltà computazionale. Gli obiettivi, pertanto, sono i seguenti:

- trovare una specifica formale della classe di problemi risolvibili praticamente;
- sviluppare metodiche che permettano di classificare i problemi algoritmici a seconda dell'appartenenza o meno a tale classe.

La definizione seguente fornisce una formalizzazione ragionevole della intuitiva nozione di problemi risolvibili praticamente.

Per ogni algoritmo M sia $L(M)$ il linguaggio riconosciuto da M.

Definizione 53. *Definiamo la classe* P *dei linguaggi decidibili in tempo polinomiale come segue:* $\mathrm{P} = \{L = L(M) \mid M$ *è un algoritmo con* $Time_M(n) \in O(n^c)$ *per qualche intero positivo* c*. Un linguaggio (problema decisionale)* L *è detto* trattabile *(risolvibile praticamente) se* $L \in \mathrm{P}$*. Un linguaggio (problema decisionale)* L *è detto* intrattabile *se* $L \notin \mathrm{P}$*.*

La classe P, dei problemi decisionali decidibili in tempo polinomiale viene fatta coincidere, in base alla definizione precedente, con la classe dei problemi risolvibili praticamente. La validità di tale corrispondenza è avallata dai seguenti fatti:

- La definizione della classe P è robusta nel senso che P è invariante in tutti i modelli computazionali. La classe P rimane la stessa indipendentemente dal fatto che essa sia definita in termini di Macchine di Turing tempo-polinomiali, in termini di programmi tempo-polinomiali scritti in un qualsiasi linguaggio di programmazione o in termini di algoritmi tempo-polinomiali derivanti da qualsiasi ragionevole formalizzazione della nozione di computazione (il termine *tempo-polinomiale* rappresenta un modo sintetico per dire *a complessità polinomiale rispetto alla risorsa tempo*).

  Ciò è conseguenza di un risultato fondamentale della teoria della complessità che dice che tutti i modelli computazionali (formalizzazioni della nozione intuitiva di algoritmo) realistici, sono *polinomialmente equivalenti.* Il fatto che due modelli computazionali siano polinomialmente equivalenti vuol dire che, se c'è un algoritmo tempo-polinomiale per un problema algoritmico U in un formalismo, allora c'è un algoritmo tempo-polinomiale

per U nell'altro formalismo e viceversa. La MT, la RAM e tutti i linguaggi di programmazione sono modelli computazionali polinomialmente equivalenti. Pertanto, se è possibile scrivere un algoritmo tempo-polinomiale per U in C++ allora esiste un algoritmo tempo-polinomiale per U in ogni ragionevole formalismo computazionale. D'altro canto, se si dimostra che non esiste una MT tempo-polinomiale che decide un linguaggio L, allora si può essere sicuri che non esiste un programma tempo-polinomiale che decide L.

- La scelta della classe P quale classe dei problemi trattabili ha anche una motivazione che si basa sulla esperienza fatta nel campo della progettazione degli algoritmi. Consideriamo la tabella 5.1 che illustra la crescita di complessità delle funzioni $10n, 2n^2, n^3, 2^n, n!$ per gli input di dimensione $10, 50, 100$ e 300.

	$n = 10$	$n = 50$	$n = 100$	$n = 300$
$f(n) = 10n$	100	500	1000	3000
$f(n) = 2n^2$	200	5000	20000	180000
$f(n) = n^3$	1000	125000	1000000	27000000
$f(n) = 2^n$	1024	(16 cifre)	(31 cifre)	(91 cifre)
$f(n) = n!$	$\approx 3.6 \cdot 10^6$	(65 cifre)	(161 cifre)	(623 cifre)

Tabella 5.1: complessità a confronto

Si osservi che se i valori di $f(n)$ sono troppo grandi, in tabella compare solo il numero di cifre necessario alla rappresentazione decimale di $f(n)$. Assumendo di avere un computer che esegue 10^6 operazioni al secondo, un algoritmo A con $\text{Time}_A(n) = n^3$ viene eseguito in 27 secondi se $n = 300$. Ma se $\text{Time}_A(n) = 2^n$ allora l'esecuzione di A per $n = 50$ impiegherebbe più di 30 anni!! Pertanto un algoritmo con complessità esponenziale non può essere considerato trattabile. Bisogna comunque tener presente che un algoritmo polinomiale con tempo di esecuzione n^{1000} non è certamente utile dal punto di vista pratico. Nonostante ciò diciamo che gli algoritmi con complessità tempo-polinomiale $O(n^c)$ sono trattabili, qualunque sia il valore di c. L'esperienza infatti ha dimostrato la ragionevolezza nel considerare trattabili le computazioni tempo-polinomiali. In quasi tutti i casi, una volta trovato un algoritmo tempo-polinomiale per un problema algoritmico, sebbene non di utilità pratica a causa del grado troppo elevato del polinomio che ne esprime il costo di esecuzione, è sempre stato possibile trovare un altro algoritmo tempo-polinomiale di grado più basso. Ci sono solo poche eccezioni note di problemi non banali in cui il miglior algoritmo tempo-polinomiale non ha utilità pratica.

ESEMPI

- *DIVISIBILITÀ DI UN INTERO PER* 4: dato un intero $n \in \mathbf{N}$, esiste m tale che $n = 4m$? Se consideriamo la rappresentazione binaria di $n \in \mathbf{N}$, poiché un intero è

divisibile per 4 se e solo se le ultime due cifre binarie sono 00, sicuramente esiste un algoritmo polinomiale che risolve il problema, o equivalentemente, una MT che decide, in un tempo polinomiale, il linguaggio associato $L = \{x \mid x \in \{1\} \circ \{0,1\}^* \circ \{00\}\}$

- *CONNESSIONE*: dato un grafo $G = (V, E)$ non orientato, G è connesso?
- *MATCHING BIPARTITO*: dato un grafo $G = (V, E)$ non orientato e bipartito (cioè tale che l'insieme dei nodi V è partizionabile in due insiemi stabili V_1 e V_2), esiste un matching con almeno k archi? (un matching è un insieme di archi a due a due non adiacenti: un matching in un grafo bipartito è composta da archi che hanno un estremo in V_1 e l'altro in V_2)

5.4 La classe NP

Abbiamo definito la classe P come la classe dei problemi trattabili, "facili". Desideriamo adesso disporre di un metodo che ci consenta di classificare i problemi a seconda che essi appartengano o meno a P. Per dimostrare che un problema L appartiene a P è sufficiente esibire un algoritmo tempo-polinomiale per L. Non disponiamo, invece, di un criterio altrettanto semplice che ci consenta di stabilire la non appartenenza a P di L, cioè per dimostrare che L è intrattabile, "difficile". Infatti per dimostrare la non appartenenza di L a P dobbiamo dimostrare che ogni possibile algoritmo che risolve L ha complessità non limitata superiormente da polinomi. E, naturalmente, dimostrare qualcosa che riguarda non uno specifico algoritmo, ma tutti i possibili algoritmi che risolvono L, anche quelli non ancora inventati, è un compito molto difficile.

Per ovviare a questo inconveniente è stato introdotto il concetto di NP-completezza che quanto meno ci fornisce una buona motivazione per credere che un dato problema è difficile, rimanendo in ogni caso nell'impossibilità di dimostrarlo. Definiremo il concetto di NP-completezza nella prossima sezione: qui ci proponiamo di definire la classe NP. A tale proposito tiriamo in ballo la computazione di tipo non deterministico.

Il non determinismo ha poco a che fare con il calcolo computazionale pratico poiché non siamo in grado di realizzarlo efficientemente su calcolatori reali. Ricordiamo che nel non determinismo, ad ogni passo computazionale, la computazione si ramifica e procede in parallelo. Questo vuol dire che un algoritmo non deterministico può avere molte computazioni sullo stesso input x, mentre qualsiasi algoritmo deterministico ha esattamente una computazione per ogni input. La definizione seguente generalizza ad un qualsisi algoritmo non deterministico la definizione di accettabilità introdotta per la MTND.

Definizione 54. *Sia M un algoritmo non deterministico. Diciamo che M accetta un linguaggio L, $L = (M)$, se:*

1. *per ogni $x \in L$, esiste almeno una computazione di M che accetta x;*
2. *per ogni $y \notin L$, tutte le computazioni di M rifiutano y.*

Per ogni input $x \in L$, la complessità temporale $\text{Time}_M(x)$ di M su x è la complessità temporale della più breve computazione accettante di M su x. Per gli input $x \notin L$, invece,

la complessità temporale $\text{Time}_M(x)$ di M su x è la complessità temporale della più lunga computazione di M su x. La complessità temporale di M è la funzione Time_M da $\mathbf{N}$ in $\mathbf{N}$ definita come:

$$\text{Time}_M(n) = \max\{\text{Time}_M(x) \mid x \in L(M) \wedge |x| = n\}$$

Definizione 55. *Definiamo la classe* NP *come la classe dei problemi decisionali decidibili non deterministicamente in tempo-polinomiale.*

$$\text{NP} = \{L(M) \mid M \textit{ è un algorimo non deterministico tempo-polinomiale}\}$$

Si osservi che per un algoritmo non deterministico è sufficiente che uno dei rami della computazione costituisca la strada giusta che provi la soluzione.

Un algoritmo non deterministico polinomiale è solo uno strumento formale per catturare la nozione di "verificabilità" in tempo-polinomiale anziché un modo realistico per risolvere problemi di decisione. Infatti, possiamo considerare una computazione deterministica come una dimostrazione del fatto che una certa stringa x appartenga (o meno) al linguaggio L. In quest'ottica, la complessità di un algoritmo deterministico che decide L è la complessità del produrre una dimostrazione dell'appartenenza di una stringa x ad L. Anziché considerare un algoritmo non deterministico come un insieme di computazioni deterministiche parallele, possiamo immaginarlo come un algoritmo deterministico che, ogniqualvolta deve scegliere tra diversi rami dell'albero delle computazioni, riceve un aiuto "dall'esterno" che lo indirizza sul ramo "migliore" per stabilire se $x \in L$ o meno. Allora, nel caso di una stringa $x \in L$, la complessità di un algoritmo non deterministico per decidere L, con ingresso x, è la complessità di stabilire che $x \in L$ sfruttando gli aiuti dall'esterno. Possiamo vedere questi aiuti dall'esterno come una dimostrazione del fatto che $x \in L$ che ci viene fornita da un nostro amico: eseguendo l'algoritmo non deterministico, noi verifichiamo che i passaggi da lui eseguiti conducano effettivamente a trovare che $x \in L$. Pertanto, la complessità della computazione non deterministica è equivalente alla complessità della verifica deterministica di una data prova del fatto che $x \in L$.

Di conseguenza, la classe NP è identificata come la classe dei problemi che possiedono un certificato polinomiale, vale a dire la classe di quei problemi per i quali esiste un algoritmo tempo-polinomiale che verifica che un'istanza affermativa del problema risulti tale. Si osservi che i problemi della classe P, hanno un certificato polinomiale. Infatti, un problema in P possiede un algoritmo risolutivo A che data un'istanza risponde SI (è un'istanza affermativa) oppure NO (è un'istanza negativa).

Il certificato polinomiale è proprio A: data un'istanza affermativa, A non può fare altro che confermare che l'istanza è affermativa. Di conseguenza, la classe NP comprende al suo interno tutti i problemi che stanno in P: vale a dire $\text{P} \subseteq \text{NP}$.

ESEMPIO DI PROBLEMA DI DECISIONE IN NP

TSP: problema del commesso viaggiatore.

Dato un insieme $C = \{c_1, c_2, \ldots c_m\}$ di città una distanza $d(c_i, c_j) \in \mathbf{N}^+$ per ogni coppia di città ed un bound $B \in \mathbf{N}^+$, esiste un *tour* di tutte le città in C avente lunghezza totale al

più B, ossia esiste una permutazione $\langle c_{i_1}, c_{i_2}, \ldots c_{i_m}\rangle$ di C tale che $d(c_{i_1}, c_{i_2}) + d(c_{i_2}, c_{i_3}) + \ldots + d(c_{i_m}, c_{i_1}) \leq B$?

Non è noto alcun algoritmo polinomiale per risolvere questo problema.

Tuttavia, supponiamo che qualcuno, data una particolare istanza I di questo problema, sappia suggerire che la risposta per quell'istanza è SI. Uno scettico chiederebbe la prova dell'affermazione, ossia la prova dell'esistenza di una permutazione con le proprietà richieste. Data una particolare permutazione, il certificato polinomiale - un algoritmo polinomiale nella lunghezza dell'istanza I - testa se la soluzione fornita è un tour (controlla cioè che ci siano effettivamente gli archi che congiungono i nodi i cui indici sono dati in sequenza nella permutazione) e in tal caso calcola la lunghezza del tour e la confronta con B.

5.5 NP-completezza

Abbiamo definito due classi di linguaggi P e NP. NP è una classe interessante dal punto di vista pratico. Infatti la maggior parte dei problemi decisionali di interesse appartiene a NP.

La domanda se P $\subset$ NP (sottoinsieme proprio) oppure P = NP costituisce un problema aperto, forse il più rilevante di tutta l'informatica teorica. Si congettura che P $\subset$ NP; le ragioni principali a sostegno di tale ipotesi sono le seguenti:

- è opinione diffusa che trovare una prova (algoritmo deterministico) non è facile quanto verificare la correttezza di una prova (algoritmo non deterministico). Questa intuizione matematica avalla l'ipotesi che P $\subset$ NP;
- si è a conoscenza di più di 3000 problemi in NP, molti dei quali studiati per più di 40 anni, per i quali non si è riusciti a trovare un algoritmo risolutivo deterministico tempo-polinomiale ed è improbabile che ciò sia solo conseguenza di una incapacità degli addetti ai lavori.

Queste considerazioni forniscono una nuova idea di come dimostrare la "difficoltà" di alcuni problemi, anche se non abbiamo metodi formali matematici diretti a tale scopo. Proviamo a dimostrare che $L \notin P$ per un $L \in NP$ assumendo che P $\subset$ NP. L'idea è quella di classificare un problema decisionale L come uno dei più difficili tra tutti i problemi in NP. L'appartenenza di L a P avrebbe come conseguenza immediata P = NP. Dal momento che non crediamo che P = NP, questo è un argomento ragionevole per credere che $L \notin P$ cioè che L è difficile. Dal momento che vogliamo evitare difficili prove di non esistenza di algoritmi efficienti, definiamo i problemi più difficili in NP come quei problemi tali che un ipotetico algoritmo efficiente per uno di essi, potrebbe essere trasformato in un algoritmo efficiente per ogni altro problema in NP. La definizione seguente formalizza questa idea.

Definizione 56. *Un problema A si riduce polinomialmente a un problema B se, data un'istanza a di A, è possibile costruire in tempo polinomiale un'istanza b del problema B tale che a è affermativa se e solo se b è affermativa.*

La notazione $A \mapsto B$ indica che A si riduce a B (in genere sottointenderemo polinomialmente). Si noti che se $A \mapsto B$, risolvendo efficientemente B siamo in grado di risolvere

efficientemente anche A. Quindi, possiamo dire che se $A \mapsto B$, il problema B è almeno tanto difficile quanto lo è A.

Osserviamo che se $A \mapsto B$ e $B \in P$ allora $A \in P$ o, equivalentemente, se $A \mapsto B$ e $A \in NP \setminus P$ allora $B \in NP \setminus P$. Un'altra osservazione di rilievo è che la riduzione polinomiale è chiaramente una relazione transitiva ossia $A \mapsto B$ e $B \mapsto C$ implicano $A \mapsto C$.

Diremo che due problemi di decisione A e B sono polinomialmente equivalenti (A e B sono di "pari difficoltà") se $A \mapsto B$ e $B \mapsto A$.

Diamo una descrizione più dettagliata della "operazione" di riduzione polinomiale.

Definizione 57. *Una riduzione (o trasformazione) polinomiale da un linguaggio L_1 su un alfabeto A_1^* ad un linguaggio L_2 su un alfabeto A_2^* è una funzione che soddisfa le seguenti condizioni:*

1. *esiste una MT deterministica che calcola f in tempo polinomiale;*
2. *$\forall x \in A_1^*, x \in L_1$ se e solo se $f(x) \in L_2$.*

ESEMPIO DI TRASFORMAZIONE POLINOMIALE

Diamo un esempio di trasformazione polinomiale: HC $\mapsto$ TSP. TSP è il problema del commesso viaggiatore definito nella sezione precedente. Definiamo HC, il problema del circuito hamiltoniano, come segue: dato un grafo $G = (V, E)$ non orientato con $V = \{v_1, v_2, \dots v_n\}$, esiste una permutazione dei vertici $\langle v_{i_1}, v_{i_2}, \dots v_{i_n} \rangle$ tale che $v_{i_j} v_{i_{j+1}} \in E$ per ogni $j = 1, \dots, n-1$ e $v_{i_n} v_{i_1} \in E$? In altre parole, esiste un circuito - cioè un insieme di archi collegati tra loro - che passa una sola volta per ogni vertice del grafo?

Dimostriamo che $HC \mapsto TSP$. Data un'istanza $G = (V, E)$ di HC, l'istanza $f(G)$ di TSP è definita come segue:

- $C = V$
- $\forall v_i v_j \in C$,
$$d(v_i, v_j) = \begin{cases} 1 & \text{se } v_i v_j \in E \\ 2 & \text{altrimenti} \end{cases}$$
- $B = n$

La funzione f può essere calcolata da un algoritmo polinomiale, in quanto per ognuna delle $\frac{n(n-1)}{2}$ distanze $d(v_i, v_j)$ è necessario verificare solo se $v_i v_j \in E$.

Per completare la prova resta da dimostrare che $G = (V, E)$ è un'istanza positiva di HC se e soltanto se $f(G)$ è un'istanza positiva di TSP:

- Se $G = (V, E)$ è un'istanza positiva di HC significa che esiste una permutazione dei vertici $\langle v_{i_1}, v_{i_2}, \dots, v_{i_n} \rangle$ che costituisce un circuito hamiltoniano per G. Allora $\langle v_{i_1}, v_{i_2}, \dots, v_{i_n} \rangle$ costituisce un tour per $f(G)$ di lunghezza $n = B$;

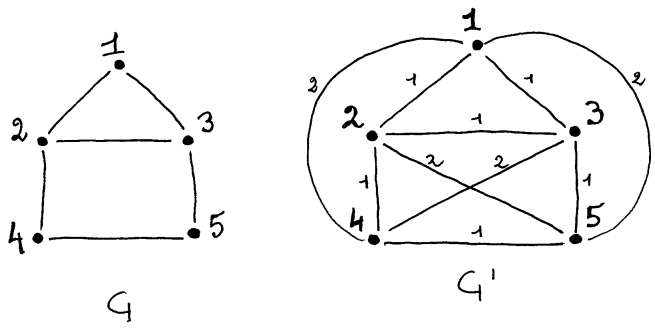


Figura 5.17: trasformazione di un istanza G di HC in un istanza $G' = f(G)$ di TSP

- se $f(G)$ è un'istanza positiva di TSP esiste un tour $\langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$ in $f(G)$ con lunghezza totale minore o uguale a B. Poiché ogni nodo è a distanza 1 o 2 e poichè si devono sommare esattamente n distanze, $B = n$ implica che tutte le distanze sono pari a 1. Dalla definizione di $f(G)$ segue che $v_i v_{i+1}$ per $1 \leq i < n$ e $v_n v_1$ sono archi di G e quindi $\langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$ è un circuito hamiltoniano per G.

Il concetto di riduzione polinomiale ci consente di introdurre una nuova classe di complessità.

Definizione 58. *Un problema A si dice NP-completo se:*

1. *$A \in NP$;*
2. *preso un qualunque problema $B \in NP$, B si riduce ad A.*

Dunque se un problema è NP-completo vuol dire che esso è almeno altrettanto difficile quanto qualunque altro problema in NP, almeno nel senso che diamo qui alla parola "difficoltà" misurata in termini di complessità computazionale. Dunque i problemi NP-completi rappresentano i più difficili problemi in NP.

Si noti che in base alla definizione di problema NP-completo, si ha che se si trovasse un algoritmo polinomiale per uno qualunque di essi avremmo trovato un algoritmo polinomiale per tutti i problemi in NP!

SAT

Consideriamo un importante problema classico della programmazione logica. Com'è noto, una qualunque espressione logica può essere posta in forma normale congiuntiva. Se chiamiamo *letterale* una variabile booleana o una variabile booleana negata, possiamo esprimere una formula in FNC come un insieme di clausole $C_1, C_2, \ldots, C_m$ ognuna delle quali è una disgiunzione di letterali. Un insieme di valori delle variabili booleane prende il nome di assegnamento di verità.

SAT è il seguente problema: data un'espressione booleana in forma normale congiuntiva, formata da m clausole e n variabili booleane $x_1, x_2, \ldots x_n$, esiste un assegnamento di verità che la soddisfa, cioè tale che l'espressione assuma valore vero?

Un contributo fondamentale alla teoria della complessità computazionale fu dato da Stephen Cook nel 1971: egli dimostrò - attraverso strumenti formali che fanno riferimento a modelli di calcolo quale ad esempio le MT - il primo risultato di NP-completezza:

Teorema 59. *SAT è NP-completo.*

Non riportiamo qui la dimostrazione, ma va sottolineato che questo teorema è di importanza fondamentale, per almeno due motivi. Il primo è che ha mostrato che la classe di problemi NP-completi non è vuota. Il secondo è che, a causa della transitività della relazione di riduzione polinomiale, questo risultato può essere utilizzato per dimostrare - in modo molto più diretto - altri risultati di NP-completezza.

Si consideri infatti un nuovo problema, e chiediamoci se esso sia NP-completo o meno. Per dimostrare che lo è secondo la definizione vista sopra, dobbiamo dimostrare anzitutto che questo problema è in NP. Questa parte della dimostrazione è spesso estremamente semplice (ma non sempre, va detto). La parte sostanziale della dimostrazione consiste nel mostrare che qualunque problema in NP si riduce al nostro problema. Dal Teorema di Cook, però, discende che qualunque problema in NP si riduce a SAT. Allora, basta dimostrare che SAT si riduce al nostro problema per completare la dimostrazione. In questo modo è possibile dimostrare, come vedremo tra poco, la NP-completezza di molti altri problemi. Una volta dimostrata la NP-completezza di un problema, questo può essere impiegato per dimostrare la NP-completezza di altri problemi ancora. Infatti, per dimostrare che un problema è NP-completo, basta prendere un qualunque problema già notoriamente NP-completo e ridurlo al problema in esame (se esso è in NP).

5.5.1 Esempi di problemi NP-completi

Vediamo nel seguito di dimostrare la NP-completezza di alcuni problemi di decisione.

- **3-SAT** 3-SAT è SAT con il vincolo che ogni clausola C_i deve essere la disgiunzione di esattamente 3 letterali.

  Dimostriamo che 3-SAT è NP-completo.

  3-SAT è in NP: un algoritmo non deterministico per 3-SAT indovina un assegnamento di valori di verità v e verifica se v soddisfa C in tempo polinomiale.

  SAT si riduce a 3-SAT: bisogna definire una trasformazione polinomiale f tale che per ogni istanza I di SAT I è soddisfacibile se e solo se $f(I)$ è soddisfacibile.

  Sia data una generica istanza di SAT: un insieme di variabili booleane $U = \{u_1, u_2, \ldots, u_n\}$ ed un insieme $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ di clausole su U

  Una istanza di 3-SAT (insieme di variabili U' ed un insieme di clausole $\mathcal{C}'$) può essere costruita come segue:

– per ogni clausola C_i $(i = 1, \ldots, m)$ definiamo un insieme di clausole $\mathcal{C}'_i$ di esattamente 3 letterali su un insieme di variabili booleane in $U \cup U'_i$, dove U'_i è un insieme di variabili aggiuntive il cui uso è limitato alle clausole in $\mathcal{C}'_i$
Gli m insiemi di clausole $\mathcal{C}'_i$ ed insiemi di variabili booleane aggiuntive U'_i possono essere combinati ponendo:
– $U' = U \cup (\bigcup_{i=1}^{m} U'_i)$
– $\mathcal{C}' = \bigcup_{i=1}^{m} \mathcal{C}'_i$

Mostriamo come si costruiscono $\mathcal{C}'_i$ e U'_i a partire dalla generica clausola C_i di SAT.

Sia $C_i = z_1 \vee z_2 \vee \ldots \vee z_k$, dove ogni z_j $(j = 1, \ldots, k)$ rappresenta un letterale su U.

Il modo in cui si costruiscono $\mathcal{C}'_i$ e U'_i a partire da C_i dipende dal valore di k:

1. $k = 1$ $(C_i = z_1)$: Sia $U'_i = \{x_{i_1}, x_{i_2}\}$. Allora $\mathcal{C}'_i = \{z_1 \vee x_{i_1} \vee x_{i_2},\ z_1 \vee \overline{x}_{i_1} \vee x_{i_2},\ z_1 \vee x_{i_1} \vee \overline{x}_{i_2},\ z_1 \vee \overline{x}_{i_1} \vee \overline{x}_{i_2}\}$
2. $k = 2$ $(C_i = z_1 \vee z_2)$: Sia $U'_i = \{x_{i_1}\}$. Allora: $\mathcal{C}'_i = \{z_1 \vee z_2 \vee x_{i_1},\ z_1 \vee z_2 \vee \overline{x}_{i_1}\}$
3. $k = 3$ $(C_i = z_1 \vee z_2 \vee z_3)$: Sia $U'_i = \emptyset$ e $\mathcal{C}'_i = \{C_i\}$
4. $k > 3$: Sia $U'_i = \{x_{i_1}, \ldots x_{i_{k-3}}\}$ e $\mathcal{C}'_i = \{z_1 \vee z_2 \vee x_{i_1},\ \overline{x}_{i_1} \vee z_3 \vee x_{i_2}, \ldots, \overline{x}_{i_{l-2}} \vee z_l \vee x_{i_{l-1}},\ \overline{x}_{i_{k-4}} \vee z_{k-2} \vee x_{i_{k-3}},\ \overline{x}_{i_{k-3}} \vee z_{k-1} \vee z_k\}$

Per provare che si tratta effettivamente di una trasformazione bisogna dimostrare che l'insieme di clausole $\mathcal{C}'$ è soddisfacibile se e solo se l'insieme $\mathcal{C}$ è soddisfacibile.

Se esiste un assegnamento di verità $\nu : U \mapsto \{T, F\}$ che soddisfa l'insieme $\mathcal{C} = \{C_1, \ldots, C_m\}$ di clausole su U, ν può essere estesa ad un assegnamento $\nu' : U' \mapsto \{T, F\}$ che soddisfa $\mathcal{C}'$.

In particolare, ν soddisfa separatamente tutte le clausole $C_i = z_1 \vee \ldots \vee z_k$ in $\mathcal{C}$. Dunque:

– nei primi tre casi $(k = 1, 2, 3)$ ν soddisfa l'insieme $\mathcal{C}'_i$ per cui ν può essere estesa a U'_i arbitrariamente.
– se $k > 3$, sia l un indice tale che $\nu(z_l) = T$:
  * se $l = 1$ o $l = 2$ allora poniamo $\nu'(x_{i_j}) = F$ per ogni variabile $j = 1, \ldots, k-3$;
  * se $l = k-1$ o $l = k$ allora poniamo $\nu'(x_{i_j}) = T$ per ogni variabile $j = 1, \ldots, k-3$;
  * altrimenti, poniamo $\nu'(x_{i_j}) = T$ per ogni variabile $j = 1, \ldots, l-2$ e $\nu'(x_{i_j}) = F$ per ogni variabile $j = l-1, \ldots, k-3$.

È immediato verificare che ν' soddisfa $\mathcal{C}'$.

D'altra parte, se esiste un assegnamento di verità ν' che soddisfa $\mathcal{C}'$ è immediato verificare che la restrizione di ν' a U è un assegnamento di verità che soddisfa $\mathcal{C}$.

- **CLIQUE** Dato un grafo $G = (V, E)$ non orientato e un intero k, esiste un sottografo completo (clique) di G con almeno k nodi?

  Dimostriamo che CLIQUE è NP-completo.

CLIQUE è chiaramente in NP, dal momento che la presenza di una clique con almeno k nodi può essere certificata dall'indicazione dei suoi nodi, e si tratta quindi solo di verificare che, presi comunque due nodi di questo insieme, c'è un arco che li collega.

Riduciamo ora SAT a CLIQUE. Data un'istanza di SAT, si costruisca un'istanza di CLIQUE come segue. Il grafo G avrà m gruppi di nodi, uno cioè per ogni clausola dell'istanza di SAT. In ciascun gruppo, avremo un nodo per ogni letterale della corrispondente clausola. Indichiamo allora con (j, l_i) il nodo del gruppo corrispondente alla clausola j e al letterale l_i (il quale, ricordiamo, può essere uguale a x_i o a $\bar{x_i}$). L'insieme degli archi è definito invece come segue. Esiste un arco tra due nodi (j, l_i) e (r, l_s) se $j \neq r$ e $l_i \neq \bar{l_s}$. In altre parole, gli archi uniscono nodi di gruppi diversi corrispondenti a letterali che non sono l'uno la negazione dell'altro. A questo punto ci chiediamo se G contiene una clique con m nodi, ossia definiamo $k = m$. (Più di m non può averne in quanto in tal caso almeno due nodi dovrebbero far parte dello stesso gruppo, mentre i nodi dello stesso gruppo non sono collegati da archi). Se G contiene una tale clique, i letterali corrispondenti ai nodi dei vari gruppi che fanno parte della clique sono tutti "compatibili", nel senso che se tra essi vi compare il letterale l_i, di certo non vi compare $\bar{l_i}$. Dunque, ponendo al valore *true* tutti questi letterali, si ottiene un assegnamento di verità che soddisfa tutte le clausole, e dunque l'istanza di SAT è affermativa. D'altro canto, se una tale clique in G non esiste, vuol dire che non è possibile selezionare m letterali tutti mutuamente compatibili, uno da ogni clausola, e dunque l'istanza di SAT è negativa. Osserviamo infine che l'istanza di G può essere costruita in tempo polinomiale, dal momento che, indicando con m e n il numero di clausole e di variabili booleane in SAT, i nodi di G sono $O(mn)$ e gli archi sono $O(m^2n^2)$.

- **INSIEME STABILE** Dato un grafo $G = (V, E)$ non orientato e un intero k, esiste un insieme stabile di G con almeno k elementi?

  Dimostriamo che INSIEME STABILE è NP-completo.

  Chiaramente INSIEME STABILE è in NP. Mostriamo ora che CLIQUE $\mapsto$ INSIEME STABILE. Data un'istanza di CLIQUE, costituita da un grafo G' e un intero k', costruiamo un'istanza di INSIEME STABILE nel seguente modo: G ha lo stesso insieme di nodi di G', e in G è presente un arco (i, j) se e solo se quell'arco non è presente in G'. In altre parole, definiamo G come il grafo complemento di G'. Si ponga poi $k = k'$. È evidente che in G è presente un insieme stabile di cardinalità k se e solo se in G' è presente una clique di cardinalità k'. L'osservazione che G può essere costruito in tempo $O(|V|^2)$ completa la dimostrazione.

- **VERTEX COVER** Dato un grafo $G = (V, E)$ non orientato e un intero k, esiste una copertura degli archi di G con al più k nodi?

  Dimostriamo che VERTEX COVER è NP-completo.

  Al solito, VERTEX COVER è in NP, in quanto, dato l'elenco dei nodi che formano un vertex cover, basta verificare che ciascun arco ha almeno un estremo un tale insieme. Riduciamo ora INSIEME STABILE a VERTEX COVER. Tale riduzione è immediata se si fa la seguente osservazione: dato un grafo G, consideriamo un insieme di nodi S su

tale grafo. Se tale insieme di nodi copre tutti gli archi ciò vuol dire che non c'è nessun arco che ha ambedue gli estremi fuori di S, ossia nell'insieme $V \setminus S$. In altre parole, i nodi di $V \setminus S$ formano un insieme stabile. Ma allora chiedersi se esiste un insieme stabile di cardinalità almeno k' eqivale a chiedersi se, sullo stesso grafo, esiste un vertex cover di cardinalità al più pari a $|V| - k'$. Formalmente, data un'istanza $\langle G' = (V', E'), k' \rangle$ di INSIEME STABILE, l'istanza di VERTEX COVER si ottiene semplicemente ponendo $G = G'$ e $k = |V| - k'$.

- **HITTING SET (HS)** Data una collezione $C = \{S_1, S_2, \ldots, S_m\}$ di sottoinsiemi di S e $k \in \mathbf{N}^+$, esiste un sottoinsieme $S' \subset S$ tale che $|S'| \leq k$ e $S' \cap S_i \neq \emptyset$ per ogni $i = 1, \ldots, m$?

  Dimostriamo che HS è NP-completo.

  È banale verificare che HS è in NP.

  Dimostriamo che HS è NP-completo con una tecnica di restrizione. Consideriamo il problema 2-HS, ossia la restrizione di HS al caso in cui ogni S_i ha cardinalità pari a 2. Tale problema è proprio il VERTEX COVER, per cui 2-HS è in NP. Ovviamente, se 2-HS è NP-completo allora anche la sua generalizzazione deve essere NP-completa.

- **PARTITION** Dato un insieme finito A e una dimensione $d(a)$ per ogni elemento di A, esiste un sottoinsieme $A' \subset A$ tale che $\sum_{a \in A'} d(a) = \sum_{a \in A \setminus A'} d(a)$?

  Omettiamo la dimostrazione; diamo invece un esempio di istanza positiva per PARTITION:

  - $A = \{a_1, a_2, \ldots, a_5\}$
  - $d(a_1) = 14, d(a_2) = 5, d(a_3) = 3, d(a_4) = 2, d(a_5) = 8$

  Esempio di istanza affermativa: $A' = \{a_1, a_4\}$.

- **KNAPSACK** Dato un insieme finito A, una dimensione $d(a)$ ed un valore $v(a)$ per ogni a in A, una capienza $C \in \mathbf{N}^+$ ed un profitto $P \in \mathbf{N}^+$, esiste un sottoinsieme $A' \subset A$ tale che $\sum_{a \in A'} d(a) \leq C$ e $\sum_{a \in A'} v(a) \geq P$?

  La restrizione di KNAPSACK al caso in cui

  - per ogni $a \in A$, $d(a) = v(a)$;
  - $C = P = \frac{1}{2} \sum_{a \in A} d(a)$

  è proprio PARTITION, quindi KNAPSACK è NP-completo.

- **MULTIPROCESSOR SCHEDULING (MS)** Dato un insieme finito A di task, una durata $d(a) \in \mathbf{N}^+$ per ogni $a \in A$ un numero m di processori una durata massima $D \in \mathbf{N}^+$, esiste una partizione $A_1, A_2, \ldots A_m$ di A in m sottoinsiemi (disgiunti) tale che $\max_{1 \leq i \leq m} \sum_{a \in A_i} d(a) \leq D$? Ovvero è possibile ripartire i task tra i processori in modo che ogni processore termini il proprio lavoro entro un tempo D?

  La restrizione di MS al caso in cui

- $m = 2$
- $D = \frac{1}{2} \sum_{a \in A} d(a)$

è PARTITION, quindi anche MS è NP-completo.

Bibliografia

[1] *Artificial Intelligence: a modern approach*, S.Russell, P.Norvig, Prentice-Hall

[2] *Automata Theory: Machines and Languages*, R.Y.Kain, McGraw-Hill

[3] *Computability and Complexity Theory*, S.Homer, A.L.Selman, Springer Verlag, ISBN 0-387-95055-9

[4] *Computational Complexity*, C.H.Papadimitriou, Addison-Wesley

[5] *Elements of the theory of computation*, H.R.Lewis, C.H.Papadimitriou, Prentice-Hall

[6] *Fondamenti di Informatica*, A.V.Aho, J.D.Ullman, Zanichelli

[7] *Gödel, Escher, Bach, un' eterna ghirlanda brillante*, D.R. Hofstadter, Adelphi

[8] *Informatica Teorica*, C.Ghezzi, D.Mandrioli, Cittàstudi

[9] *Introduction to automata theory, languages and computation*, J.E.Hopcroft, J.D.Ullman, Addison-Wesley

[10] *Mathematical logic for computer science*, M.Ben-Ari, Prentice-Hall

[11] *Logica a Informatica*, A.Asperti, A.Ciabattoni, McGraw-Hill

[12] *Teoria della Complessità Computazionale*, D.P.Bovet, P.Crescenzi, Franco Angeli

[13] *Teoria della Computabilità, Logica, Teoria dei Linguaggi Formali*, Aiello, Albano, Attardi, Montanari, Materiali Didattici ETS