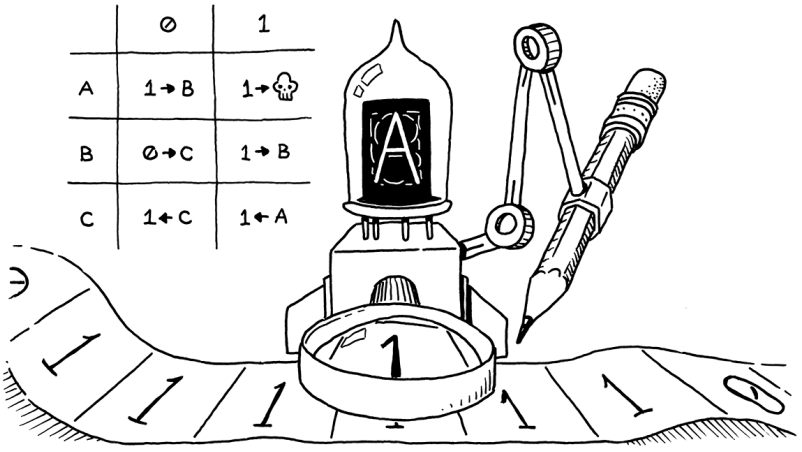
UNIVERSITÀ DEGLI STUDI DI TRIESTE

Marco Sgobino

Dispense del corso di

Complessità e Crittografia



Anno Accademico 2021-2022

Un grazie va a *Matthew Butterick*, ed ai suoi preziosissimi consigli sull'uso corretto (e responsabile) della tipografia.

https://practicaltypography.com/

Chapter 0
Indice

Chapter 1
La macchina di Turing

1.1 Descrizione della macchina

Una **macchina di Turing** è una macchina che è descritta da un insieme di *simboli* $\Gamma = \{\alpha, \beta, \gamma, \delta, \dots\}$ *finito* e da un insieme di *stati* $Q = \{q_1, q_2, \dots, q_n\}$ anch'esso *finito*. La macchina di Turing dispone di un nastro di memoria *potenzialmente illimitato* a destra e a sinistra, avente delle celle contenenti i simboli; essa identifica il simbolo nella posizione dove la *testina* della macchina è collocata sul nastro. Ad ogni iterazione della macchina di Turing viene letto il simbolo, e a seconda dello stato q_i, viene intrapresa un'azione fra le 3 seguenti:

- spostamento della testina a destra;
- spostamento della testina a sinistra;
- riscrittura del simbolo sotto la testina con uno qualsiasi appartenente all'alfabeto di simboli.

Nello specifico, una macchina di Turing è univocamente identificata dalla sua *matrice di transizione* $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{L, R\})$[1], dove i simboli L ed R sono rappresentativi dell'azione di spostarsi, rispettivamente, a sinistra e a destra del nastro di

[1] In talune circostanze, è possibile trovare una definizione differente, cioè

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\};$$

memoria. La matrice di transizione lega, dunque, ciascuno stato al simbolo collocato immediatamente sotto alla testina nel nastro di memoria, stabilendo in maniera univoca l'azione da intraprendere. Si può dire dunque che la matrice di transizione fornisce alla macchina di Turing l'elenco delle possibili azioni da intraprendere, alla lettura di un simbolo sulla cella corrente, a seconda dello stato in cui si trova. L'insieme delle azioni che la macchina di Turing compie è quindi la realizzazione del programma stesso, quello che nei termini del modello RAM si sarebbe detto essere la sequenza di istruzioni elementari. In parole povere, non si scrive un vero e proprio programma come nel caso della macchina a modello RAM, bensì si specifica il passo da compiere alla lettura di un determinato simbolo sul nastro. I passi sono riportati in questa tabella, la matrice di transizione, e lì si descrive il meccanismo della macchina.

Una macchina di Turing può anche essere accompagnata da un alfabeto *ausiliario*, ovverosia da un alfabeto $\mathcal{V}$ comprendente simboli simili a quelli di Γ, ma che vengono utilizzati qualora la macchina di Turing avesse *già processato* la cella in questione (i simboli ausiliari sono "simili" a quelli dell'alfabeto tradizionale, ma hanno una differenza che ne permette la distinzione). Tipicamente, l'utilizzo dell'alfabeto ausiliario è importante nel caso specifico in cui si adoperino procedure per le quali è utile ricordare se una cella sia già stata in precedenza processata dalla macchina di Turing, oppure no — in ogni caso, l'alfabeto ausiliario è meramente un sussidio che permette una semplificazione della procedura o aiuta ad interpretare il comportamento della macchina, non è in nessun modo un qualsivoglia tipo di estensione della macchina di Turing. Come sarà mostrato in seguito, ciascuna altra possibile definizione di macchina di Turing è, dal punto di vista della potenza computazione, del tutto equivalente alla definizione già data sopra. In parole povere, ci facciamo aiutare dall'alfabeto ausiliario per "ricordarci" su quali celle siamo già passati.

Diversamente dal modello RAM, la quantità di memoria destinata ad ogni cella è *limitata*, poiché vi può essere collocato soltanto un numero finito di simboli, quelli appunto dell'insieme Γ. Qua non si possono più inserire numeri grandissimi, bensì bisogna accontentarsi dei simboli che abbiamo inserito nel nostro alfabeto, un numero finito di essi. Ciononostante, la macchina di Turing ha dei vantaggi, ad esempio si presta meglio alla trattazione di stringhe, poiché i simboli possono rappresentare qualsivoglia tipologia di entità astratta, mentre per il modello RAM si avrebbe necessità di una codifica fra numeri reali e simboli da trattare. Non vi è più dunque la limitazio-

in altre parole, è una macchina che *si muove sempre sul nastro,* poiché per ogni stato è sempre definito un movimento. Per questo tipo di macchina, sono necessarie diverse regole d'ingaggio, e i programmi in essa costruiti saranno radicalmente differenti. La differenza fra i due modelli rappresenta un'evidenza della versatilità della macchina di Turing, e della possibilità di introdurla con varie definizioni, dove ciascuna delle quali possiede le proprie peculiarità.

	q_1	q_2	$\cdots$	q_n
α	β/q_2	γ/q_2	$\cdots$	
β	L/q_1	γ/q_3	$\cdots$	
γ	γ/q_3	R/q_2	$\cdots$	
$\vdots$	$\vdots$	$\vdots$	$\ddots$	

Tabella 1.1: Possibile matrice di transizione per una macchina di Turing. Le righe corrispondono ai simboli dell'alfabeto Γ, mentre le colonne sono corrispondenti ai singoli stati dell'insieme Q. Ogni elemento della tabella indica il simbolo da scrivere/stato in cui la macchina dovrà trovarsi al passo successivo,

ne imposta dal fatto che all'interno di una cella possa risiedere esclusivamente un numero naturale, non importa quanto grande sia; nella macchina di Turing le celle possono contenere simboli di qualsiasi natura essi siano. Lo "svantaggio", tuttavia, è che all'interno di ogni cella non può essere contenuta una quantità *arbitraria* di informazione, come invece avveniva per il modello RAM, che faceva uso dei numeri naturali. In ogni caso, le due macchine sono equivalenti, quindi possiamo in qualche modo "scavalcare" questa limitazione che solo in apparenza è limitante.

Tipicamente, assieme all'alfabeto che definisce una macchina di Turing viene definito un sottoinsieme sigma di *simboli di input*, $\Sigma \subset \Gamma$, in concomitanza al quale viene definito un simbolo vuoto, *blank*, $b \in \Gamma - \Sigma$. Il simbolo b incarna dunque l'idea di *cella vuota* — si pensi infatti al valore che una cella di memoria primaria qualsiasi di un computer reale avrebbe, al momento immediatamente successivo all'accensione: essa risulterebbe posta allo zero logico, di fatto non conterrebbe alcun valore d'interesse, dato che essa non è ancora stata "toccata" dall'esecuzione di alcun programma. Il simbolo *blank* sta a significare proprio questo, ed è l'equivalente del valore 'zero' del modello RAM, dove all'avvio del programma ogni cella di memoria fuorché quelle contenenti i valori di ingresso contiene il numero reale 0.

Ricapitolando, ogni macchina di Turing viene univocamente definita da:

- un insieme finito di simboli Γ — essi comprendono sia i simboli di input Σ che il simbolo *blank* b;
- un insieme finito di stati Q;
- una funzione (matrice) di transizione $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{L, R\})$.

Una diversa maniera per definire una macchina di Turing è tramite la *quaterna* o *quadrupla* q_i, s_j, α, q_k, dove q_i è lo *stato in cui si trova la macchina*, s_j è il *simbolo letto dalla testina*, α è il *simbolo scritto nella matrice di transizione* e q_k è lo *stato successivo* in cui la

macchina di Turing si troverà al termine dell'esecuzione di α. In particolare, l'operazione che la macchina di Turing effettua dipende dal simbolo α scritto nella matrice di transizione:

- se $\alpha = s_i$, sostituisci il simbolo s_j con s_i;
- se $\alpha = R$, muovi la testina a destra;
- se $\alpha = L$, muovi la testina a sinistra.

L'insieme di tutte queste quaterne descrive l'insieme dei possibili comportamenti della macchina di Turing, dunque questo elenco è del tutto equivalente ad una matrice di transizione per come la abbiamo definita in precedenza.

In parole povere, anche così definita una macchina di Turing può sovrascrivere un simbolo presente sulla cella con un altro presente nel suo alfabeto, Γ, può spostare la testina di un'unità a destra, e può fare altrettanto a sinistra.

Resta da scogliere il nodo relativo alla terminazione della macchina di Turing. Nel modello RAM, la terminazione avveniva qualora le istruzioni si fossero esaurite, o più precisamente qualora l'indice dell'istruzione successiva fosse quello di un'istruzione assente nella sequenza che definisce la procedura. In assenza del concetto di "istruzione", secondo quale regole dovrebbe terminare una macchina di Turing?

Lo *stop* della computazione di una macchina di Turing avviene qualora la coppia q_i, s_j **non** sia presente nella matrice di transizione. Nel caso di una coppia simbolo—stato non presente nella matrice di transizione, la macchina di Turing avrà terminazione, e vi sarà il riconoscimento del valore finale di computazione, espresso similmente al caso del modello RAM con una convenzione che permetta di identificare il valore finale del risultato della computazione. Dunque, una macchina di Turing si ferma quando non sa più che fare.

Se lo stop della computazione è stato chiarito, ora resta da definire il metodo con cui andremo a recuperare il valore finale della computazione. Se per una macchina modello RAM potevamo recuperare il valore alla cella 0, con una macchina di Turing ciò non è più possibile (come possiamo, infatti, immagazzinare in una singola cella un numero arbitrariamente grande? È evidente che con questa limitazione già citata nei paragrafi precedenti dobbiamo escogitare un nuovo stratagemma per ovviare al problema). Convenzionalmente, l'esito di una macchina di Turing è una particolare configurazione di memoria dello stato finale. Si è infatti scelto che il risultato $f(x)$ sia da leggersi come il numero totale di occorrenze di simboli 1 (meno un'occorrenza) sul nastro nella configurazione iniziale, se la computazione è andata a convergenza, mentre il risultato sarà da considerarsi indefinito altrimenti. La sequenza di occorrenze del

simbolo 1 verrà delimitata dal carattere *blank*. Quindi vi saranno $f(x) + 1$ simboli $'1'$, e pertanto sarà da contare un simbolo $'1'$ in meno (lo $'0'$ sarà indicato con la presenza di un singolo simbolo $'1'$, per il risultato $'1'$ saranno necessari invece 2 simboli $'1'$di fila, e così via). Per quanto invece riguarda i risultati di tipo *vettoriale*, cioè del tipo $f(x_1, x_2, \ldots, x_n)$, ebbene sarà sufficiente costruire n "quadrati" in cui racchiudere gli $x + 1$ simboli 1, ciascuno delimitato dal simbolo b. In altre parole, la situazione è quella descritta dalla Figura 1.1.

Per farla breve, preleveremo il risultato x secondo la formula

$$x = \#\text{ di simboli 1 delimitato da due caratteri blank, sottratto uno.}$$

Un difetto evidente della macchina di Turing è la sua difficoltà di trattazione. Per via dell'utilizzo tramite la matrice di transizione è particolarmente difficile scrivere programmi per una macchina di Turing – questo è principalmente dovuto al fatto che la macchina di Turing è una macchina 'a stati', dove non vi è un insieme di istruzioni da applicare direttamente, ma è necessario determinare prima di tutto la matrice di transizione relativa a ciò che bisogna calcolare, stato per stato e simbolo per simbolo. Il programma non è più esprimibile come una successione di passi "dall'alto verso il basso", ma va scritto sotto forma di una criptica matrice di transizione dove i passi da definire sono quelli compiuti da una macchina di Turing a fronte di un determinato simbolo.

Una macchina di Turing, non importa come sia stata definita, può essere adoperata sostanzialmente per compiere 3 operazioni:

1. per il *calcolo* di una funzione — in questo caso la macchina di Turing è intesa come *calcolatore*, e lo scopo è quello banalmente di calcolare una funzione $f : \mathbb{N}^n \to \mathbb{N}$. In questo caso, la macchina di Turing risulta essere meno efficiente del modello RAM, per via dell'assenza del comodo sistema di istruzio-

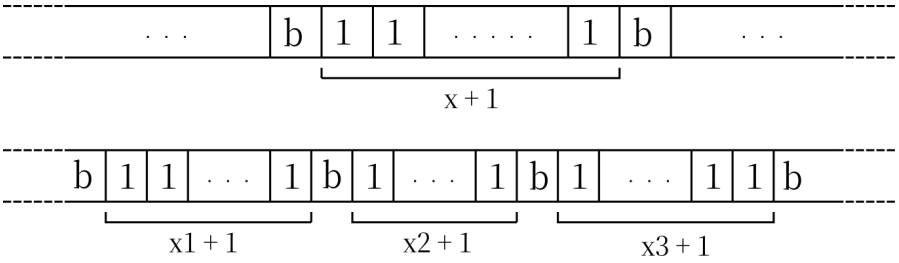


Figura 1.1: Risultato singolo (sopra) e *vettoriale* (sotto) di una macchina di Turing.

ni presente in quest ultimo. Cionondimeno, essa può compiere ugualmente il calcolo con la medesima capacità;

2. per il *riconoscimento* di una stringa — la macchina di Turing è intesa come *accettore*. In questo contesto la macchina di Turing è di gran lunga più efficiente del modello RAM, poiché non è richiesta la codifica da numeri naturali a simboli, necessaria invece se si ha a che fare esclusivamente con numeri naturali;
3. per la *decisione* di un predicato — la macchina di Turing è intesa come *decisore*.

1.1.1 Equivalenza fra macchina di Turing e $\mathcal{R}$

Un importante primo teorema definisce l'equivalenza della macchina di Turing (avente insieme delle funzioni computabili $\mathcal{TC}$ all'insieme delle funzioni parziali ricorsive $\mathcal{R}$, e lo lega indissolubilmente all'insieme delle funzioni computabili dal modello RAM C.

Teorema 1 *dell'equivalenza della macchina di Turing all'insieme $\mathcal{R}$ delle funzioni parziali ricorsive*

Per esso vale che

$$\mathcal{R} \equiv \mathcal{TC} \equiv C,$$

ovverosia gli insiemi delle funzioni Turing–computabili $\mathcal{TC}$, delle funzioni parziali ricorsive $\mathcal{R}$ e delle funzioni computabili dal modello RAM C sono equivalenti.

DIMOSTRAZIONE— Un possibile spunto di dimostrazione di $\mathcal{TC} \subseteq \mathcal{R}$ si ha grazie al fatto che la configurazione e lo stato della MdT durante la computazione possono essere codificati da un numero naturale; le operazioni sulla macchina sono rappresentate da funzioni ricorsive su questi numeri. Il viceversa è invece mostrabile tenendo conto che si può verificare che $\mathcal{TC}$ contiene le funzioni di base ed è chiusa rispetto a sostituzione, ricorsione e minimazione illimitata.

■

1.1.2 Macchina di Turing per il calcolo della somma

Supponiamo di voler fare la somma fra due numeri interi naturali, x ed y. In questo caso, la macchina di Turing dovrebbe calcolare la funzione $f(x, y) = x + y$. Per fare

ciò, costruiremo gli elementi fondamentali della macchina di Turing. In particolare, avremo che l'alfabeto di simboli $\Gamma = \{0, 1, b\}$, cioè avremo bisogno esclusivamente di 2 simboli eccezion fatta per il simbolo *blank*, mentre invece faremo uso di 3 stati $Q = \{q_1, q_2, q_3\}$. Lo stato iniziale è lo stato q_1, mentre lo stato finale è q_3. Resta ora da definire la matrice di transizione. Uno fra i tanti modi di definirla è il seguente (faremo uso delle quadruple),

$$\begin{array}{cccc} q_1 & 1 & b & q_1 \\ q_1 & b & R & q_2 \\ q_2 & 1 & b & q_3 \\ q_2 & b & R & q_2 \end{array}$$

Tabella 1.2: Matrice di transizione per la macchina di Turing che calcola $f(x, y) = x + y$, espressa a quadruple.

L'idea è quella di togliere due simboli 1, di modo che gli 1 rimanenti corrispondano al valore del risultato finale. Infatti, provando a calcolare $f(2, 1) = 3$ avremo che

$$\begin{array}{cccccccc} q_1 & & & & & & & \\ 1 & 1 & 1 & b & 1 & 1 & b & b \\ q_1 & & & & & & & \\ b & 1 & 1 & b & 1 & 1 & b & b \\ & q_2 & & & & & & \\ b & 1 & 1 & b & 1 & 1 & b & b \\ & & q_3 & & & & & \\ b & b & 1 & b & 1 & 1 & b & b \end{array}$$

ed il numero finale di simboli 1 corrisponde proprio al valore della somma, $2 + 1 = 3$.

Possiamo anche costruire un grafo della macchina di cui sopra, mostrato in Figura 1.2.

1.2 Altre versioni della macchina di Turing

1.2.1 Estensioni e menomazioni

Una macchina di Turing può essere apparentemente potenziata mediante l'estensione di essa tramite l'uso di *nastri multitraccia*. In altre parole, anziché adoperare un singolo nastro, si adoperano più nastri simultaneamente. La macchina di Turing viene dunque espansa tramite l'aggiunta di uno *stato della memoria suppletiva*, che ci indica semplicemente il numero del nastro dove la macchina di Turing sta agendo in quel momento. Dunque, ci possiamo immaginare una macchina di Turing con tanti nastri e tante testine che lavorano contemporaneamente, come illustrato in Figura 1.3.

La domanda ora è se l'introduzione della multitraccia consenta di generare una nuova macchina, con capacità di calcolo superiori a quelle della macchina di Turing. La risposta è negativa: dal punto di vista della capacità computazione, una macchina di Turing multinastro non aumenta né diminuisce le capacità. Una macchina di Turing multinastro può essere implementata con un *sistema multitraccia* — in altre parole, vengono adoperate *tante testine quante sono i nastri*. Ci si può facilmente ricondurre alla macchina di Turing convenzionale semplicemente eliminando l'appena introdotto sistema multitraccia e facendo agire la macchina su un nastro alla volta, o per meglio dire, una macchina di Turing multitraccia può essere **simulata** da una macchina di Turing convenzionale: essa dunque, non produce alcun tipo di miglioramento dal punto di vista della computazione, cioè le due macchine hanno **la stessa** potenza com-

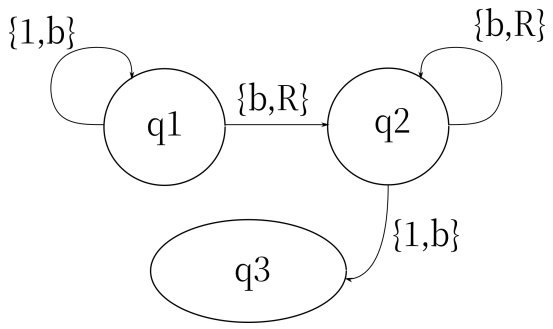


Figura 1.2: Grafo della macchina di Turing per le somme.

putazionale (Figura 1.4). In parole povere, una macchina di Turing capace di agire su N nastri contemporaneamente può essere simulata (e quindi il suo funzionamento sarà lo stesso) da N macchine di Turing, ciascuna che agisce per conto proprio su uno degli N nastri.

Tale rappresentazione, tuttavia, può avere il vantaggio di presentare una maggiore somiglianza con il tipo di computazione svolto all'interno di un computer moderno. Si pensi infatti alla memoria RAM, alla memoria cache, al disco rigido e così via; una macchina multinastro può ricalcare le varie memorie di un computer moderno. una macchina di Turing può dunque "simulare" qualsiasi computer moderno[2] semplicemente introducendo tanti nastri e tante tracce quante sono quelle dei dispositivi fisici adoperati dal calcolatore moderno. Nella fattispecie, si avrà un nastro ed una traccia per la memoria RAM, un altro nastro ed un'altra traccia per la memoria a disco rigido, e così via. In realtà, si tratta esclusivamente di un artificio che ci consente di tracciare un collegamento fra il calcolatore moderno e la macchina di Turing, poiché una macchina multinastro, sia essa multitraccia, può essere *emulata* da una macchina di Turing a nastro singolo. Quindi, anche se una macchina a nastro singolo è più che sufficiente per calcolare qualsiasi funzione che è anche calcolabile da un nostro computer moderno, possiamo adoperare l'astrazione del multinastro per facilitarci nella comprensione e rendere il tutto più simile ad un computer. Anche se poi la macchina di Turing lo surclassa comunque, essendo dotata di *memoria infinita* (mentre i nostri computer, no).

Il medesimo discorso si applica anche al primo tentativo di *menomare* la macchina di Turing (una sorta di sabotaggio), nel senso che potremmo pensare di rendere il nastro semi-infinito, cioè illimitato solo a destra o solo a sinistra. Ebbene, nonostante il sabotaggio venga messo in atto, la macchina di Turing menomata avrà di fatto la medesima capacità computazionale di quella "standard", perché possiamo sempre avere a disposizione una quantità illimitata di memoria da un lato (un po' come per il modello RAM), o far uso di trucchi come quello dell'alfabeto ausiliario $\mathcal{V}$ che comunque faciliterebbero (di molto) le computazioni in una situazione simile. Comunque la si veda, una macchina di Turing non si può né potenziare né depotenziare, a meno di non effettuare operazioni che *sconvolgano* il suo funzionamento minandone le fondamenta, oppure rimuovendo l'ipotesi della memoria illimitata da entrambi i lati (in quel caso si che si incapperebbe in una limitazione pesante).

[2] Un computer può a sua volta simulare da una macchina di Turing, dal momento che possono essere applicati potenzialmente infiniti banchi di memoria al computer — nella pratica, tuttavia, sappiamo che ciò non è possibile, e i banchi di memoria non saranno mai del tutto *illmitati*. Sarebbe bello altrimenti!

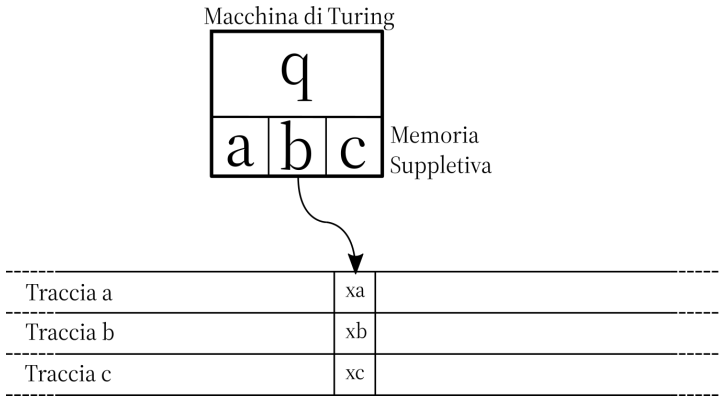


Figura 1.3: Macchina di Turing avente memoria con nastro multitraccia. La testina può collocarsi, una alla volta, su ciascuna delle nastre. La memoria suppletiva rende possibile tenere traccia di quale nastro si sta adoperando per l'esecuzione delle operazioni.

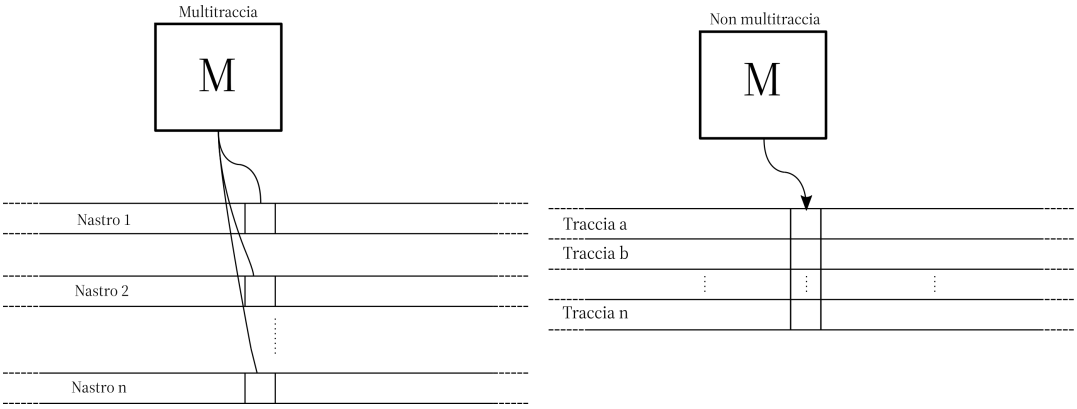


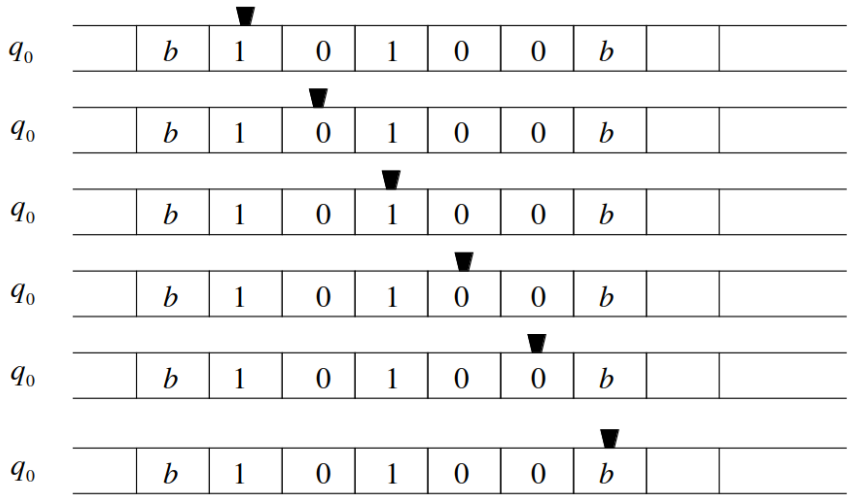
Figura 1.4: Equivalenza fra una macchina di Turing a multitraccia e una macchina di Turing a singola testina. Non importa il numero di testine: la macchina a singola testina potrà sempre percorrere (pazientemente) un nastro dopo l'altro, simulando la macchina multitraccia.

1.2.2 Macchina RAM per *accettare* stringhe

Uno dei possibili utilizzi per una macchina di Turing (o più in generale, per una macchina Turing-equivalente) è quello di *accettore* di stringhe: la macchina riceve in ingresso un simbolo, una *stringa*; essa si dirà *accettata* qualora la computazione risultante terminasse nello stato di ACCETTAZIONE, altrimenti si dirà *rifiutata* qualora la computazione terminasse invece in uno stato di RIFIUTO. Per farla breve, essa ha come "input" (stato iniziale) una stringa, e l'evoluzione di questo input potrà portare o ad uno stato in cui la stringa è considerata accettata, o in uno stato opposto in cui essa è rifiutata. Si osservi che, in ogni caso, non c'è garanzia che una macchina di Turing non possa ciclare all'infinito; in quel caso saremmo di fronte ad una divergenza.

Nel modello RAM, per accettare una stringa è necessario operare una codifica. In particolare, la stringa viene codificata in un numero naturale e la macchina risponde con "1" o "0" a seconda che la stringa venga o meno accettata. In questo caso la codifica avviene sia per i valori di input, da stringa a numeri naturali, che per i valori finali della computazione, da numeri naturali ad accettato–rifiutato. Con la macchina di Turing, invece, si può far intervenire direttamente uno *stato di accettazione* ACCETTAZIONE q_Y o uno *stato di rifiuto* RIFIUTO q_N. La computazione terminerà qualora uno fra questi due stati venisse raggiunto dalla macchina — con conseguente accettazione o rifiuto della stringa a seconda dello stato finale. Un'altra possibilità è quella di accontentarci di *semi-decidere* riguardo l'accettazione di una stringa, cioè di dotarsi di una macchina in grado di riconoscere sì la stringa in questione, ma di *non poterla rifiutare*, poiché in tal caso vi sarebbe un'infinita computazione, una divergenza.

Lo stato iniziale viene di solito denotato con q_0, e potrebbero esserci degli stati ulteriori "intermedi" fra quello iniziale e quelli terminanti. Un esempio di accettazione è dato dalla macchina illustrata in Figura 1.5. La macchina riconosce il linguaggio dato dalle stringhe con due "zeri" nelle ultime due posizioni a destra, in particolare essa riconosce tutte le stringhe che terminano con "00".



	0	1	b
q_0	$(q_0,0,R)$	$(q_0,1,R)$	$(q_1,0,L)$
q_1	$(q_2,0,L)$	$(q_3,0,L)$	$(q_N,0,L)$
q_2	$(q_Y,0,L)$	$(q_N,0,L)$	$(q_N,0,L)$
q_3	$(q_N,0,L)$	$(q_N,0,L)$	$(q_N,0,L)$

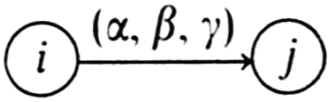
Figura 1.5: Macchina che riconosce il linguaggio dato dalle stringhe con due 0 nelle ultime due posizioni a destra, e suo funzionamento.

1.2.3 Macchina di Turing definita a grafo

Una macchina di Turing può anche essere "definita a grafo". In questo modello di definizione, la macchina di Turing,

- ha un *nastro semi-illimitato* a destra, diviso in celle;
- ha un alfabeto Σ accoppiato da un alfabeto *ausiliario* $\mathcal{V}$;
- ha un simbolo di spaziatura Δ, equivalente al simbolo *blank* b;
- un puntatore, del tutto equivalente alla testina;
- un *programma*, definito come **grafo finito orientato**, con i vertici definiti come *stato*. Vi è uno stato di inizio, indicato con INIZIO, e un sottoinsieme eventualmente vuoto di stati di arresto, indicati con ACCETTAZIONE. I nodi del grafo sono collegati da *archi*.

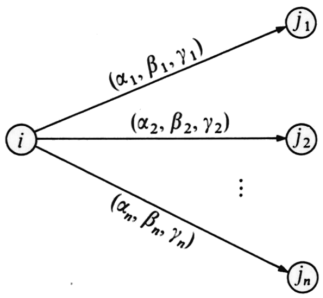
Ciascun arco è della forma



dove

$$\alpha \in \Sigma \cup \mathcal{V} \cup \{\Delta\}, \beta \in \Sigma \cup \mathcal{V} \cup \{\Delta\} \text{ e } \gamma \in \{L, R\}.$$

Dunque, siamo nello stato i; la macchina legge α, scrive β al posto di α, e infine va a destra oppure a sinistra a seconda che il simbolo γ sia pari ad R o ad L. Una proprietà importante è che tutti gli archi che partono da un medesimo vertice devono avere α diversi. Questo perché la macchina non può tollerare delle ambiguità: leggendo un simbolo, può compiere univocamente una sola azione, descritta dal grafo.



Se così non fosse, si avrebbero più cammini possibili per uno stesso simbolo – un'ipotesi che come vedremo in seguito sarà violata assumendo che una macchina di Turing possa non essere di tipo *deterministico*.

Si possono disegnare le macchine di Turing direttamente con i grafi, per poi all'occorrenza convertirle nella forma a matrici di transizione. Il problema espresso in Figura 1.6 è il problema del riconoscimento di una stringa avente forma $a^n b^n | n \geq 0$, ed è molto noto nella teoria della computabilità, poiché è un tipico esempio di problema che è risolubile da una macchina di Turing, ma **non risolubile** mediante una *macchina a stati finiti*. Tali macchine, infatti, non presentano alcun tipo di *memoria*, e dunque per questa particolare mancanza non sono in grado di risolvere il problema del riconoscimento. La macchina di Turing, invece, è in grado di risolverlo in virtù della sua superiore potenza di computazione (e di memoria).

Un ulteriore esempio dell'applicazione della macchina di Turing a grafo è mostrato in Figura 1.7, dove la macchina in questione è in grado di concatenare due stringhe a due lettere a e b.

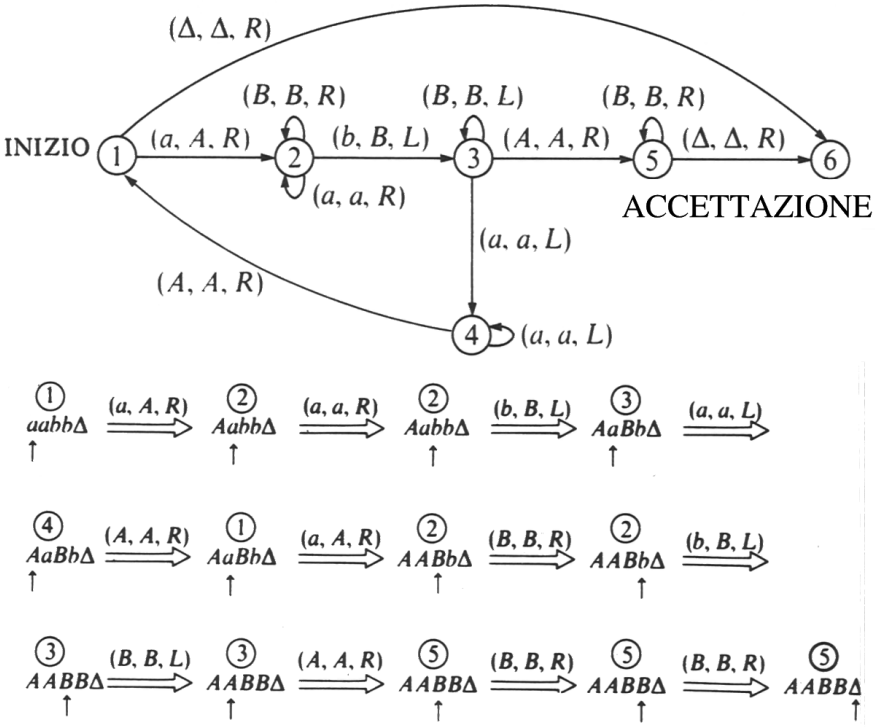
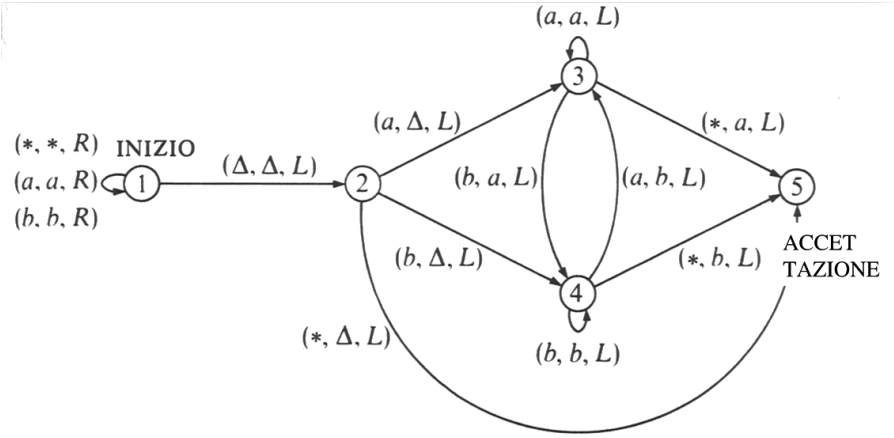


Figura 1.6: Macchina di Turing in grado di riconoscere una stringa della forma $a^n b^n | n \geq 0$.



La macchina di Turing M_7 su $\{a, b, *\}$, descritta nella figura 1.10 calcola la funzione di concatenazione su $\Sigma = \{a, b\}$: essa prende una coppia di parole (w_1, w_2) come entrata e produce la parola $w_1 w_2$

Nastro d'entrata	a	b	a	a	*	b	b	a	b	Δ	Δ	Δ	···
Nastro d'uscita	a	b	a	a	b	b	a	b	Δ	Δ	Δ	Δ	···

Figura 1.7: Macchina di Turing in grado di concatenare due stringhe w_1 e w_2.

Chapter 2
Le macchine non deterministiche

2.1 Gerarchie delle potenze di calcolo

Oltre alla macchina di Turing nelle sue varie versioni, sono possibili altre tipologie di macchine, con vari livelli di gerarchia fra potenze di calcolo che ne determinano le possibilità. Il seguente elenco ne illustra alcune, da minor capacità di memoria a maggiore:

macchine a *stati finiti* in questo caso, hanno 0 *memorie push-down* — le macchine a stati finiti sono le meno potenti in assoluto dal punto di vista computazionale, e non sono in grado di risolvere il problema del riconoscimento di stringhe $a^n b^n$, espresso in Figura 1.6;

macchine a *1 memoria push-down* dotate di una memoria push-down che implementa una pila FIFO (*first in—-first out*), hanno una maggiore potenza di calcolo rispetto alla macchina a stati finiti;

macchine a *2 memorie push-down* dotate di 2 memorie push-down, esse sono equivalenti alla macchina di Turing.

macchine *di Post* sono uno speciale tipo di macchine a memorie push-down, aventi una memoria di tipo LIFO (*last in—-first out*). Esse sono equivalenti alle macchine a 2 memorie push-down e alle macchine di Turing;

macchine con 3 o più pile push-down non forniscono vantaggi dal livello della potenza computazionale, e sono tutte Turing-equivalenti — il vantaggio è, semmai, nella semplificazione di calcoli fornita dall'introduzione della pila aggiuntiva.

Perciò, la gerarchia delle potenze di calcolo è la seguente, dall'alto verso il basso:

1. Macchine non deterministiche di Turing, con due memorie push-down — Macchine di Turing, Modello RAM, Macchine di Post, macchine finite con due memorie push-down (sono in grado di accettare $\{a^n b^n a^n | n \geq 0\}$). Si potrebbe dimostrare che le macchine di Turing deterministiche e non deterministiche hanno la medesima potenza di calcolo;
2. Macchine finite non deterministiche con una memoria push-down, sono in grado di riconoscere $\{ww^R | w \in \{a, b\}^*\}$[1];
3. Macchine finite con una memoria push-down, riconoscono $\{a^n b^n | n \geq 0\}$;
4. Macchine finite non deterministiche senza memorie push-down — macchine finite senza memorie push-down — automi finiti.

2.1.1 Alcune definizioni sulle stringhe

Sia dato l'alfabeto Σ^*. Faremo uso di tre fondamentali funzioni definite su Σ^*:

- l'operazione `testa(x)`, che fornisce la "testa" di una stringa, ovverosia la lettera di estrema sinistra della parola x;
- l'operazione `coda(x)`, la quale invece fornisce la "coda" di una stringa, o la stringa privata della sua testa x;
- l'operazione $\sigma \cdot x$, che *concatena* la lettera σ e la parola x, per formare un'unica parola nuova.

Per esempio, se $\Sigma = \{a, b\}$, allora `testa(abb) = a`, `coda(abb) = bb` e inoltre $a \cdot bb = abb$.

[1] Il *nodo di decisione* che introduce il non determinismo serve per gestire la situazione data dall'incapacità di riconoscere il punto di rottura ww^R, cioè il punto in cui finisce w ed inizia w^R. Con il non determinismo, ogni qualvolta si arriva al nodo di decisione si tengono valide entrambe le possibilità — dunque, per questa ragione essa è più potente della macchina ad una memoria push-down deterministica.

2.2 La macchina di Turing non deterministica

La macchina di Turing effettiva è di tipo *deterministico*. Nella fattispecie, una macchina deterministica definita a grafo vede ciascun arco che parte dallo stesso vertice avere *simboli diversi*. Se ciò non fosse vero, leggendo un unico simbolo α_i e trovandosi nello stato q_j la macchina di Turing non potrebbe "scegliere" il percorso, poiché ne esisterebbe più di uno. Viceversa, una macchina *non deterministica* rompe questa assunzione, e permette alla macchina di compiere una scelta, sia essa arbitraria o del tutto casuale, riguardo quale percorso seguire fra i vari disponibili.

Il *non determinismo* viene introdotto per due ragioni:

- si desidera osservare se una macchina di Turing non deterministica sia più *potente* oppure no di una macchina di Turing deterministica;
- si cerca di valutare se possano esistere differenti paradigmi di computazione (ad esempio, la computazione parallela è possibile?).

Una *macchina di Turing non deterministica* può avere nel suo diagramma di flusso dei *nodi di decisione* dove a fronte di un medesimo simbolo vi sono *più archi possibili* — il non determinismo viene dunque introdotto da questo fenomeno: una macchina di Turing non deterministica può scegliere il suo percorso con una legge non deterministica, ma semmai dettata dal caso o da una *scelta arbitraria*. È un po' come se la nostra macchina di Turing fosse in grado di percorrere *simultaneamente* tutti i percorsi possibili descritti da un certo simbolo α_q, poiché essa indiscriminatamente accetta più archi alla lettura del medesimo simbolo α_q, e può perciò muoversi verso più nodi simultaneamente.

Una macchina non deterministica accetta l'idea che vi siano più possibilità di sviluppo di una computazione a fronte di un medesimo simbolo identificato sul nastro e associato ad un determinato stato; sono dunque possibili diversi percorsi di computazione. Questo tipo di macchina, ad esempio, potrebbe percorrere *tutti i percorsi simultaneamente*, concependo il **parallelismo** nella computazione, nel senso che più possibilità e percorsi nel calcolo sono possibili.

Questa nuova potenzialità, apparentemente di gran lunga migliorativa, **non aumenta** la potenza di calcolo della macchina di Turing. Questo perché una macchina di Turing deterministica è in grado, in linea di principio, di simulare una macchina non deterministica. Per convincersi di ciò è sufficiente compiere una ricerca per ogni diramazione

dell'albero dei nodi, con un aumento esponenziale[2] del numero di operazioni da effettuare, ma pur sempre un'operazione realizzabile e computabile da una macchina di Turing. In altre parole, armata di una certa pazienza, la macchina di Turing deterministica può percorrere uno alla volta ciascuno dei sentieri che la sua controparte non deterministica avrebbe percorso tutti contemporaneamente. Una macchina non deterministica ha dunque il potenziale vantaggio di poter risolvere problemi in *tempo lineare* che dalla macchinia deterministica sarebbero invece risolti in in tempo esponenziale. Una grandissima spinta della velocità di calcolo, ma che non garantisce di calcolare nulla di più di quanto si potrebbe fare con una macchina di Turing deterministica. Resta da domandarsi se ciò renda in qualche modo possibile rendere più efficienti i nostri computer. Ebbene, una macchina di Turing non deterministica può essere vista come una macchina di Turing deterministica ove, ad ogni diramazione, *lancia la computazione di nuove macchine di Turing.* Se una macchina di Turing deterministica deve seguire 3 nuovi percorsi a partire da un certo nodo, essa è come se accendesse 3 nuove macchine di Turing, ciascuna su uno dei 3 percorsi e che calcola per conto proprio. È chiaro che questo procedimento di lancio di nuove macchine può essere, nella nostra immaginazione, arbitrariamente effettuato senza poter incappare in qualche limitazione, accendendo di volta in volta sempre nuove macchine senza mai porre un limite al loro numero. Nella realtà materiale (purtroppo) ciò equivarrebbe a lanciare *un nuovo computer* ad ogni diramazione. Questo è realizzabile fino ad un certo punto (ammesso di disporre di tanti computer), ma è chiaro che il procedimento non può essere arbitrariamente replicato. Siccome i nostri computer sono dotati di core, possono raggiungere (da soli) buone capacità di parallelizzazione, ma infinitamente inferiori a quelle richieste per una vera parallelizzazione e per simulare il non determinismo dal punto di vista della velocità di calcolo (gli infiniti core non esistono). Se ciò fosse invece possibile nel mondo materiale, potremmo effettivamente risolvere problemi difficilmente trattabili in tempo polinomiale, rendendoli di fatto trattabili.

Il parallelismo è insito nella macchina non deterministica; supponendo di voler risolvere un problema di accettazione, la stringa si considera *accettata* qualora uno qualunque fra i percorsi possibili incappasse nello stato di ACCETTAZIONE. Più nello specifico, una parola $w \in \Sigma^*$ si dice *accettata* da una macchina di Turing non deterministica se esiste una computazione della macchina M che cominci con entrata $x = w$, e che termini ad un arresto con lo stato di ACCETTAZIONE. Se w non viene accettata e lo stato di arresto è quello del RIFIUTO, allora si dice che w è *rifiutata*.

[2] L'aumento esponenziale è dovuto al fatto che, ad ogni diramazione effettuata da una macchina di Turing non deterministica, la relativa macchina deterministica dovrà (almeno) sdoppiarsi su 2 percorsi.

In alternativa, siamo di fronte ad un ciclo infinito $w \in ciclo(M)$, cioè dinanzi ad una divergenza.

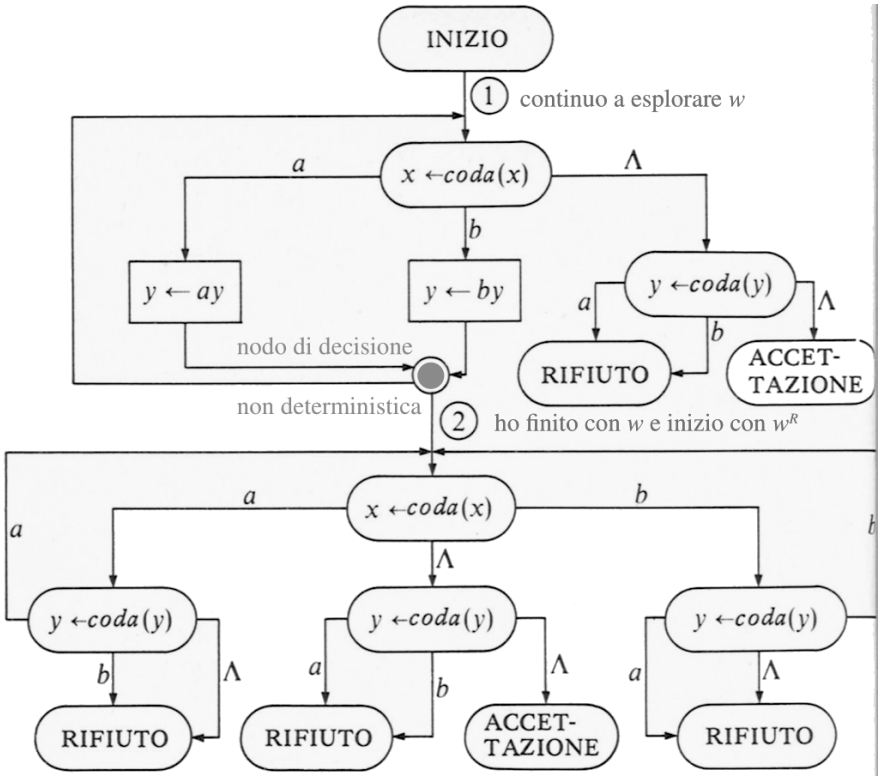


Figura 2.1: Macchina finita non deterministica che accetta le stringhe del tipo $\{ww^R | w \in \{a,b\}^*\}$. Questa macchina accetta tutte le stringhe di quel tipo, e rifiuta tutte le altre Σ^* – accettazione(M_6). Esempi di stringhe accettate sono la stringa vuota Δ, aa, $babbab$, $bbbabbabbb$ e così via.

Possiamo quindi pensare ad una macchina di Turing non deterministica come ad una macchina avente per ogni cella, nella matrice di transizione, un *insieme* $\delta(q,s)$ di triple q_i, s_i, α_i, dove $\alpha_i \in \{L,R\}$ e ad ogni elemento dell'insieme corrisponde una possibile scelta da compiere arbitrariamente o casualmente; dunque da lì è ottenuto il non determinismo della macchina. Ogni possibile scelta genererà un diverso ramo nell'albero della computazione; per l'accettazione di un simbolo è sufficiente che uno *qualsiasi* fra i rami di computazione termini nello stato di ACCETTAZIONE. Dunque, benché una macchina di Turing non deterministica non **aumenti** la potenza di calcolo intrinseca della macchina, essa consente la **parallelizzazione** dei possibili percorsi della computazione, rendendo di fatto possibile risolvere in tempo lineare problemi che sarebbero risolubili comunque, ma in tempo esponenziale, per una macchina deterministica.

2.2.1 Equivalenza fra macchine non deterministiche e macchine multinastro

Teorema 2 *Ogni macchina di Turing non deterministica ha un equivalente macchina di Turing deterministica multinastro.*

DIMOSTRAZIONE— Una traccia della dimostrazione è una ricerca nell'albero di computazione. Con almeno due nastri, si simula con uno la macchina non deterministica mentre con l'altro si collezionano tutti i possibili k "prossimi passi" della tabella di transizione. La macchina di Turing deterministica allora controllerà tutte le configurazioni stato–simbolo, livello per livello, dell'albero di computazione. Lo stato finale di ACCETTAZIONE terminerà la computazione complessiva.

■

L'idea è quella, dunque, di *simulare* il non determinismo compiendo un numero crescente in modo esponenziale di passi, uno per ogni ramo dell'albero non deterministico di computazione.

Teorema 3 *Ogni macchina di Turing non deterministica $T_M(n)$ ha un equivalente macchina di Turing deterministica con ordine $2^{O(T_M(n))}$.*

DIMOSTRAZIONE— Una traccia può essere che il numero massimo di foglie è $O(b^{T_M(n)})$, dove b è il numero di figli. Il tempo per viaggiare dalla radice lungo ogni ramo è, per una macchina multinastro,

$$O(T_M(n)b^{T_M(n)}) = 2^{O(T_M(n))},$$

mentre per una macchina a nastro singolo

$$(2^{O(T_M(n))})^2 = 2^{O(2T_M(n))} = 2^{O(T_M(n))},$$

dunque l'ordine è lo stesso.

■

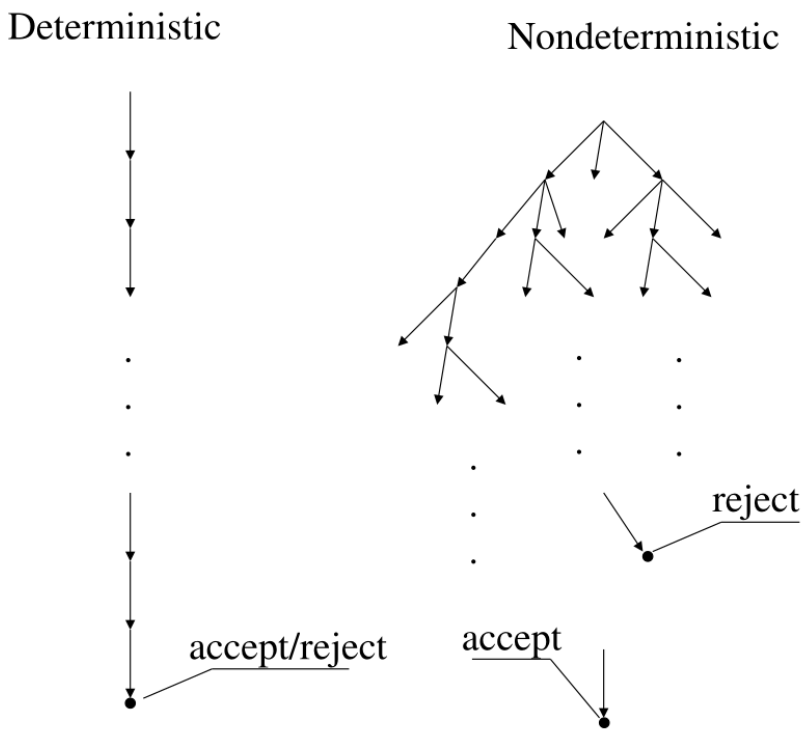


Figura 2.2: Esempio di computazione "in parallelo" effettuata dalla macchina non deterministica (a destra), e simulazione da parte della macchina deterministica (a sinistra).

Chapter 3
Le reti neuronali di Hopfield

Le **reti neuronali** incarnano un diverso paradigma di computazione rispetto a quello della computazione procedurale algoritmica. Storicamente esse prendono spunto dalla natura, in particolare dal concetto di *neurone*, inteso come singola unità di calcolo, dotata di input, di output e di una *funzione caratteristica*. Diversamente dall'idea della computazione procedurale, dove la *complessità* è definita tramite la complessità di tipo computazionale (o temporale), le reti neuronali esprimono la loro complessità attraverso la **complessità strutturale**, o **complessità circuitale**. In altre parole, per risolvere un problema l'idea è quella di aumentare la complessità di tipo circuitale della rete, ad esempio aggiungendo neuroni, facendo variare le conduttanze, adoperando diverse funzioni di attivazione. La computazione è di tipo *collettivo*, ed emerge come proprietà dell'*evoluzione dinamica* della rete. Ogni decisore locale, detto *neurone*, non ha visibilità della computazione globale, e svolge il proprio compito localmente — ogni decisore locale concorre alla soluzione globale in modo sfumato, con il proprio valore in stato alto o basso: la rete è **robusta** rispetto a malfunzionamenti locali, poiché ciascun neurone singolo non è critico e non può pregiudicare l'intera rete da solo. Spesso è addirittura possibile eliminare qualche neurone senza che la rete ne risenta in maniera rilevante.

Dal punto di vista strettamente logico, il paradigma di computazione a reti neuronali è l'unico paradigma radicalmente differente da quello procedurale, mentre il paradigma

a DNA[1] è una tipologia di calcolo che, nella realtà, non differisce sostanzialmente dal metodo procedurale.

Esistono due filoni di reti neuronali; il primo associato ai **perceptron**, macchine a strati che fungono da riconoscitori di pattern e configurazioni. Le variabili di ingresso rappresentano un ente, una configurazione del sistema esterno, e il perceptron è in grado di fornire in output una risposta che consente di riconoscere tale configurazione o pattern in base a come essa sia stata configurata e ai suoi parametri. I cosiddetti *multilayer perceptron* sono particolari casi di perceptron, aventi i neuroni organizzati in *layer*. Vi sono due layer "principali" di input e di output, e dei layer "nascosti" (in gergo *hidden layer*) dove avvengono ulteriori passaggi intermedi. Tipicamente, i layer sono costituiti da neuroni aventi particolari *funzioni di attivazione*. Le funzioni di attivazione sono ciò che determina il comportamento dei singoli neuroni — in particolare, esse determinano la maniera in cui un neurone debba attivarsi o rimanere nello stato di quiete. I perceptron hanno la caratteristica fondamentale di essere *feed-forward*, cioè di avere una *direzionalità* nella rete: i neuroni non possono essere connessi in cicli, cioè il corrispondente grafo è aciclico. L'infrastruttura dei perceptron ha un comportamento di tipo euristico, cioè non esiste alcun modello di computazione associato alla struttura, non vi sono teoremi e, di fatto, il tutto è carente di un solido un apparato matematico. Per modellare un perceptron, solitamente, si adottano tecniche euristiche di *machine learning*, con apprendimenti automatici.

Il secondo filone, invece, è quello della **rete di Hopfield**; tale filone sarebbe in grado, in linea di principio, di *risolvere problemi* di natura matematica. Mediante una rete di Hopfield è possibile risolvere ad esempio il *travelling salesman problem*, un problema presumibilmente intrattabile[2]. Le reti di Hopfield, diversamente dai perceptron, possono essere modellate con grafi ciclici, dunque esistono percorsi ciclici fra neuroni.

Ambedue i modelli, tuttavia, non possono essere considerati dei veri e propri modelli alternativi a quello della macchina di Turing; in primis per l'assenza dell'apparato matematico, e in secondo luogo poiché è stato dimostrato che nella sostanza e sotto opportune ipotesi una rete neurale ha la stessa potenza di calcolo del modello RAM. Le reti neurali quindi, non sono un modello di computazione alternativo, semmai sono un *paradigma* differente.

Le reti neuronali hanno avuto grande successo in 3 periodi storici:

[1] Trattasi di un paradigma di calcolo dove si va a cercare lo spazio delle soluzioni tramite una vera e propria ricerca esauriente, e si catturano quelle più adatte. La potenza di questo metodo è quello di potersi permettere una ricerca esauriente della soluzione, poiché ciascun DNA è infinitesimamente piccolo, e dunque può essere esaminato in parallelo.

[2] "Presumibilmente" si riferisce al fatto che, fino ad ora, nessuno è riuscito a dimostrare né che tale problema può essere risolto in tempo polinomiale, né che non può esserlo.

- all'inizio degli anni 40 — nel 1948 fu fornito il primo modello di neurone e rete neuronale. L'idea fu quella di avvalersi di una macchina fisica per simulare il comportamento dei neuroni naturali, presenti nel nostro cervello (Figura 3.1. All'epoca uscirono anche articoli su memorie a breve e lungo termine adoperando reti neuronali;
- durante gli anni 80 — nei primi anni 80, Hopfield individuò un modello di reti neuronali che associava le reti neuronali ad un particolare modello matematico; qualità che prima era assente. Si tratta di una caratteristica fondamentale: quando si cerca di risolvere problemi con le reti neuronali, l'assenza di teoremi (ad esempio, quelli asintotici) non ci permette di stabilire la *qualità* di una soluzione, oppure se la computazione andrà in qualche maniera a buon fine. Dunque, l'assenza di un buon apparato matematico che faccia da base alle reti neuronali è il principale svantaggio di tale paradigma. Hopfield riuscì a fornire la soluzione di alcuni fra questi problemi matematici, suggerendo la possibilità di risolvere problemi, come ad esempio quello sopra citato del *travelling salesman problem*. Ci fu allora una corsa dal punto di vista scientifico riguardo la possibilità di risolvere in maniera efficiente i problemi presumibilmente intrattabili. Si può dimostrare, tuttavia, che la potenza di computazione di una macchina di Hopfield **non è superiore** alla capacità di computazione della macchina di Turing. Con una macchina di Hopfield non è dunque possibile risolvere in tempo polinomiale problemi che non sono risolubili in tale maniera già dalla macchina di Turing — vi è dunque l'*equivalenza* fra le due macchine. Le reti di Hopfield caddero pertanto ben presto in disuso;
- il terzo ed ultimo periodo d'oro delle reti neuronali è il giorno d'oggi, dove la potenza di calcolo superiore delle macchine moderne apre la strada ad un ampio uso delle reti neuronali in applicazioni concrete, che si avvalgono della pura potenza computazionale di gran lunga maggiore rispetto al passato per produrre risultati in tempo apprezzabile. Esse si vedono applicate in problemi di machine learning quali il riconoscimento di pattern in immagine, la classificazione di immagini, l'estrazione di feature (per il loro utilizzo con ulteriori tecniche di machine learning).

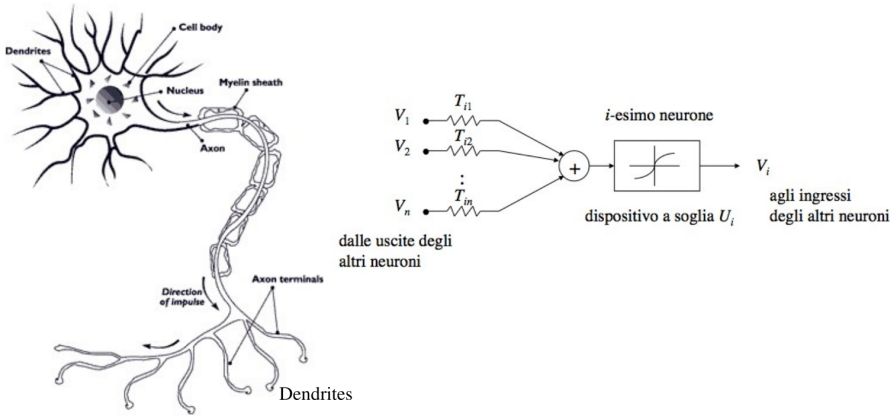


Figura 3.1: Paragone fra neurone naturale e neurone di Hopfield. Si osservino innanzitutto le somiglianze fra le due strutture. La somiglianza principale è il fatto che anche i neuroni naturali presentano vari input e vari output, collegati con diversi altri neuroni. Tipicamente, anche un neurone naturale presenta una vera e propria funzione di attivazione, che ne determina il comportamento "a soglia". La stimolazione del neurone naturale è stimolata da segnali continui, sebbene il *firing* (eccitazione del neurone) avvenga in maniera discreta.

3.1 Il neurone reale ed il neurone simulato

Una macchina di Hopfield simula un neurone reale. Vi sono all'incirca 10^{11} neuroni (cento miliardi) nel cervello, i quali formano una rete neuronale di una complessità strabiliante. Il cervello è in grado di svolgere una quantità di compiti enorme, in modo molto efficiente — il cervello è infatti la struttura più complessa che esista nell'universo noto, non paragonabile ad alcun altro tipo di struttura, sia essa già esistente in natura o creata artificialmente. I segnali elettrici nel cervello si sviluppano nell'ambito dei *segnali continui*, tuttavia il funzionamento di un neurone si svolge, in realtà, propriamente nell'ambito dei *segnali discreti*. Vi sono infatti i cosiddetti *spike*: un neurone si "eccita" o si "rilassa", manifestando dunque un comportamento discreto secondo questo punto di vista, anche se il sottostante segnale è, di fatto, un segnale elettrochimico di natura continua.

3.2 La rete discreta di Hopfield

La **rete discreta di Hopfield** è composta da singoli neuroni, modellati tramite le tensioni elettriche. Ciascun i-esimo neurone ha la forma illustrata in Figura 3.1: esso è dotato di n ingressi V_j collegati ad altri neuroni, delle "conduttanze" dal valore $T_{i,j}$ che simulano i contatti fra neuroni[3]; ciascuno degli ingressi si sommerà e la somma verrà valutata dalla *funzione di attivazione*, collocata nella parte centrale. Il neurone di Hopfield è un dispositivo dal comportamento *a soglia* (sigmoide): la funzione di attivazione pesa la somma degli ingressi, producendo un output V_i **pressoché discreto** da dirigere agli ingressi di altri neuroni. Nel caso discreto, la soglia si dice essere *rigida*: stabilita la soglia U_i del neurone, avremo due possibilità,

- o $V_i = 1$ se $\sum_{j \neq i} T_{ij} V_j > U_i$;
- oppure $V_i = 0$ se $\sum_{j \neq i} T_{ij} V_j < U_i$;

cioè la soglia determina il comportamento che l'uscita del neurone presenta a seconda dei valori dell'ingresso (o più precisamente, a seconda di quanto vale la loro somma pesata dalle conduttanze T_{ij}).

In un certo senso quindi, si può affermare che un neurone di Hopfield si comporta "discretamente", nel senso che il suo valore di uscita può assumere valori o prossimi allo 0, o prossimi all'1, con possibili valori intermedi dipendenti esclusivamente dalla forma che la funzione di attivazione presenta (si assume che essa difficilmente possa avere la forma di un perfetto gradino, e che esistano porzioni di essa in cui il valore è compreso fra 0 ed 1 con valori continui. Ciononostante, è sufficiente interpretare il segnale continuo in senso discreto, cioè valutando con un'opportuna soglia il valore dell'output di un neurone, un po' come avviene nei circuiti digitali). Nella fattispecie, consideriamo un comportamento a soglia di tipo ideale, cioè nel quale la funzione di attivazione è un gradino ideale.

Per quanto concerne le reti di Hopfield, è importante fare due osservazioni. La prima è che se $V_i = 1$, in ogni caso (sia che fosse già in 1 o che fosse in 0) si ha che $\Delta V_i \geq 0$. Viceversa, se $V_i = 0$, in ogni caso si avrà che $\Delta V_i \leq 0$: questa cruciale osservazione ci permette di stabilire che

[3] Il valore di tali conduttanze *varia* in funzione del tempo, in particolare in base a quanto frequentemente il neurone viene sollecitato — questo modello artificiale è basato su quello reale, dove i dendriti variano a seconda della frequenza con cui il neurone viene sollecitato.

$$\Delta E = -\Delta V_i(\sum_{j \neq i} T_{ij} V_j - U_i) \leq 0,$$

cioè la variazione di energia, ovvero la potenza, *è sempre minore di zero*. La **variazione dell'energia della rete**, dunque, **è sempre negativa**. Questo fatto è di fondamentale interesse, poiché ci definisce la maniera in cui la rete tende ad evolversi, riducendo ad ogni passo la quantità di energia totale.

La seconda importante osservazione è che nel caso in cui si assuma una simmetria della rete (cioè quando il neurome i-esimo incide sul neurone j-esimo tanto quanto il neurone j-esimo incide su quello i-esimo), l'energia E_i del neurone i-esimo sarà pari a

$$E_i = -V_i(\sum_{j \neq i} T_{ij} V_j - U_i),$$

e sotto ipotesi $T_{ij} = T_{ji}$, si ha infine che l'**energia della rete di Hopfield** E è pari a

$$E = -\frac{1}{2} \sum_{ij, j \neq i} T_{ij} V_i V_j + \sum_i V_i U_i, \tag{3.1}$$

un termine corrispondente ad una *forma quadratica*.

Dunque, per come è stata impostata la rete e per le due osservazioni effettuate sopra, l'energia della rete è associata ad un funzionale che decresce sempre: tenendo conto di tale risultato fondamentale, si può costruire una rete di Hopfield.

I neuroni possono, almeno secondo una particolare configurazione iniziale, non presentare un valore dell'uscita adeguato ai valori presenti all'ingresso. In ogni istante discreto, ciascun neurone ha la medesima probabilità di essere eccitato (firing), cioè si può verificare la compatibilità del suo valore di uscita sulla base della somma pesata degli ingressi. Potrebbe darsi che la somma degli ingressi sia maggiore della soglia U_i, ma che il valore d'uscita sia basso, o viceversa che la somma degli ingressi presenti un valore basso, con uscita alta — in tal caso, avremo che il neurone *non è soddisfatto*, cioè è in una condizione di disagio. Esso vorrebbe vedere la propria coerenza soddisfatta: tanto più i neuroni di una rete di Hopfield sono soddisfatti, tanto minore sarà l'energia associata allo stato che produrrà tale soddisfacimento.

Il modello che Hopfield suggerisce per superare tale difficoltà è quello di *verificare*, neurone per neurone, la coerenza fra i valori ai suoi ingressi e il suo valore d'uscita, con una verifica che avviene in tempo discreto. Qualora non ci fosse coerenza, il neu-

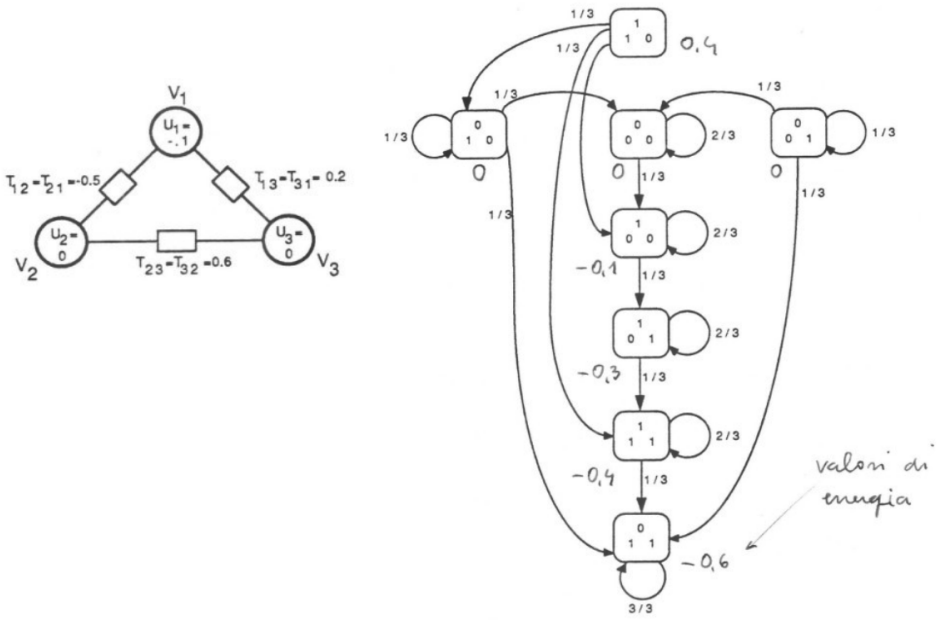


Figura 3.2: Rete di Hopfield costituita da 3 neuroni, collegati ad anello. Ciascun neurone è collegato ad ogni altro mediante una particolare conduttanza T_{ij}, che soddisfa la relazione di simmetria $T_{ij} = T_{ji}$. Si osserva, sulla destra, l'evoluzione probabilistica della rete, con decremento dell'energia. Non sono ammessi incrementi energetici.

rone andrà modificato, ottenendo un nuovo stato della rete. In ogni caso, il fenomeno a cui si assiste è quello per cui ad ogni variazione l'energia della rete cala complessivamente. Il procedimento di verifica proposto da Hopfield è del tutto aleatorio, e consiste nell'aggiustamento progressivo dei pesi, calando di volta in volta di energia. Secondo la legge dei grandi numeri, la rete si evolverà percorrendo i vari passi, fino a terminare in una configurazione a minima energia: la configurazione di *minimo locale* dell'energia della rete. Una volta raggiunta la configurazione di minimo locale dell'energia, non si può più uscire dalla configurazione. Il minimo dell'energia della rete, dunque, dovrà corrispondere in qualche maniera alla soluzione cercata.

Risolvere un problema con una rete di Hopfield, dunque, necessita di un passo iniziale di *codifica*, cioè che il procedimento di minimazione dell'energia della rete, cioè la minimazione di una forma quadratica, abbia un corrispettivo con la soluzione del problema desiderato. In linea di principio, avendo a disposizione una valida codifica, è possibile risolvere il problema semplicemente minimizzando l'energia della rete corrispondente, con il procedimento di verifica in tempi discreti visibile in Figura 3.2.

3.2.1 Gli stati stabili

Specialmente per le macchina a topologia più complessa, possono esistere configurazioni di minimo locale dell'energia della rete che però non sono configurazioni di *minimo globale*.

A volte potrebbe risultare comodo imporre la stabilità di uno stato, per esempio per l'applicazione delle reti al concetto di *memoria indirizzabile*. Per fare ciò, si impongono delle equazioni alla rete, di modo da eleggere determinati neuroni della rete come stabili. Un esempio di ciò è mostrato in Figura 3.3.

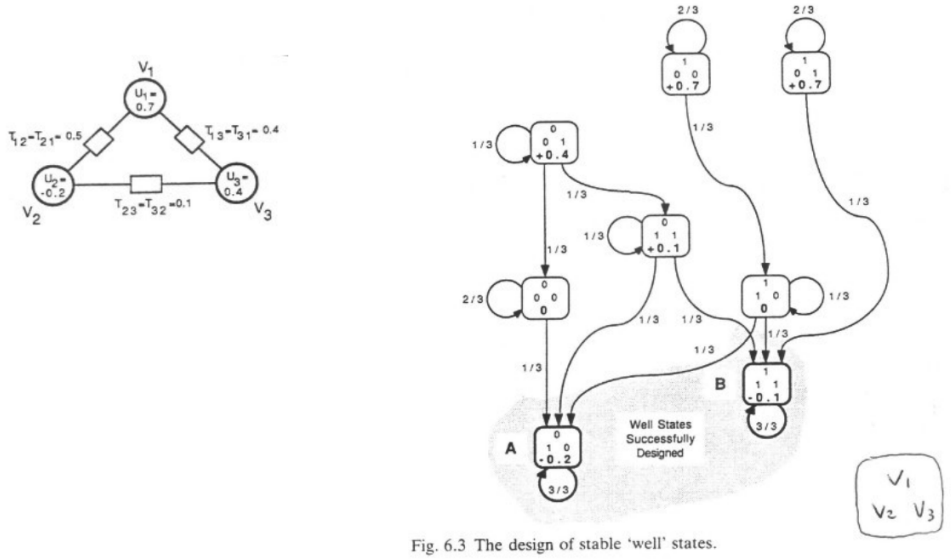


Figura 3.3: Reti di Hopfield, dove i neuroni al livello di energia più basso sono eletti *stati stabili*.

Tipicamente, per reti aventi n neuroni è possibile eleggere $\log(n)$ stati come stabili, esibendo dunque una crescita di tipo logaritmico.

Se si vuole imporre uno stato stabile, è necessario determinarlo con l'opportuna equazione. Supponiamo di voler imporre lo stato stabile $V_1 = 0$; per fare ciò, bisogna imporre

$$T_{12}V_2 + T_{13}V_3 - U_1 < 0,$$

relativo al sistema di equazioni

$$\begin{cases} T_{12} - U_1 < 0 \\ U_2 < 0 \\ T_{23} - U_3 < 0 \end{cases},$$

che corrisponde all'imposizione dello stato stabile A in Figura 3.3. Sempre facendo riferimento alla stessa rete, imporre lo stato stabile B significherebbe vedere soddisfatto il sistema di equazioni

$$\begin{cases} T_{12} + T_{13} - U_1 < 0 \\ T_{12} + T_{23} - U_2 > 0 \\ T_{23} + T_{13} - U_3 > 0 \end{cases}.$$

Infatti, avendo definito le quantità nella seguente maniera,

$$\begin{aligned} T_{12} = T_{21} = 0,5 \quad U_1 = +0,7 \\ T_{13} = T_{31} = 0,4 \quad U_1 = -0,2 \\ T_{23} = T_{32} = 0,1 \quad U_1 = +0,4 \end{aligned}$$

si ha che, per l'equazione di stato A (a sinistra), e l'equazione di stato B (a destra)

$$\begin{cases} 0,5 - 0,7 < 0 \quad \leftrightarrow \quad \text{Sì} \quad 0,5 + 0,4 - 0,7 > 0 \quad \leftrightarrow \quad \text{Sì} \\ -0,2 < 0 \quad \leftrightarrow \quad \text{Sì} \quad 0,5 + 0,1 + 0,2 > 0 \quad \leftrightarrow \quad \text{Sì} \\ -0,1 - 0,4 < 0 \quad \leftrightarrow \quad \text{Sì} \quad 0,1 + 0,4 - 0,4 > 0 \quad \leftrightarrow \quad \text{Sì} \end{cases}$$

ed ambedue gli stati sono, di conseguenza, soddisfatti. In Figura 3.3 essi corrispondono agli stati stabili 010 e 111.

3.2.2 La macchina di Boltzmann

Può capitare che per una data rete di Hopfield vi siano alcuni minimi locali, detti anche *falsi minimi*; come ad esempio mostrato in Figura 3.4; la soluzione corrispondente

al minimo globale potrebbe quindi non corrispondere a quella trovata mediante l'evoluzione, se la rete è incappata in un minimo locale. Ciascun minimo locale è, di fatto, uno stato stabile, dal quale nel caso vi si incappasse, non si potrebbe uscire, poiché il livello di energia non può mai aumentare.

Di solito, si desidera che l'evoluzione converga verso gli stati stabili scelti, oppure verso un minimo globale — i minimi locali, tuttavia, risulterebbero essere alla stregua di "falsi" stati stabili, che andrebbero a minare il comportamento desiderato della rete, per il quale sarebbe desiderabile ottenere una soluzione unica, e corrispondente al minimo globale (o comunque a stati stabili dal valore sufficientemente basso di energia).

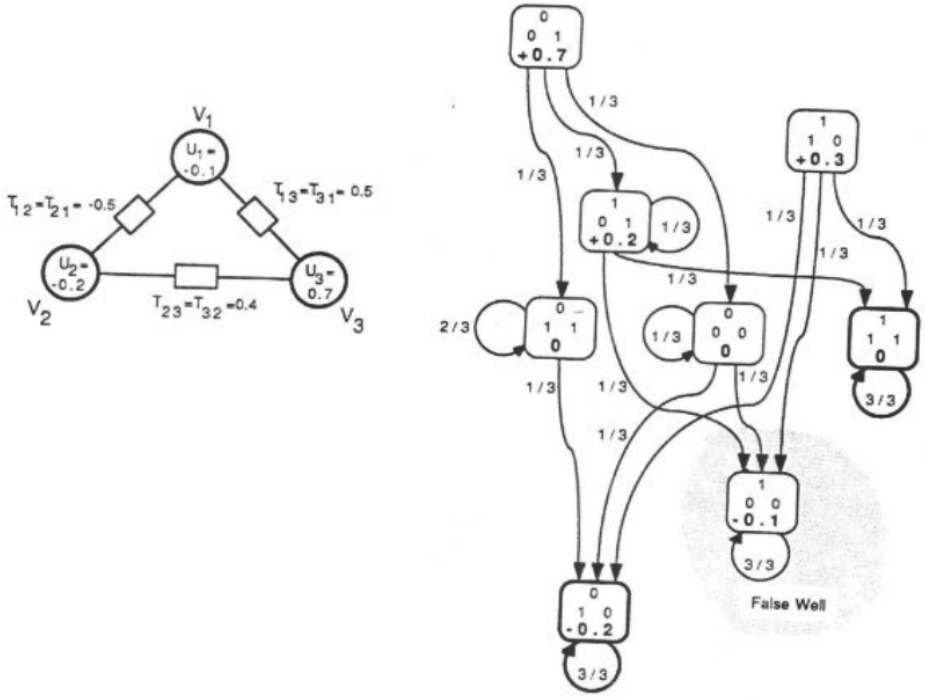


Figura 3.4: Il problema dei *falsi minimi*, anche detti minimi locali.

Con un po' di fortuna, i minimi locali possono avere un valore prossimo al minimo assoluto; tuttavia ciò non è per nulla garantito, e non esistono criteri matematici per stabilire la qualità della soluzione ottenuta. In altre parole, *non possiamo sapere a priori se la qualità della soluzione ottenuta*, incappando in uno stato stabile artificiale o di minimo locale, *è sufficientemente buona oppure non lo è.*

Ciò rappresenta in ultima analisi un evidente difetto delle reti neurali di Hopfield, poiché diversamente al paradigma della computazione procedurale dove è sempre e comunque possibile ricondursi in qualche maniera al modello di Turing o al modello RAM, e dunque ai teoremi studiati in tale ambito, nel paradigma di computazione delle reti di Hopfield non c'è alcun criterio matematico in grado di stabilire **a priori** la qualità di una determinata soluzione.

Una soluzione al problema dei minimi locali imprevisti arrivò a cavallo degli anni 80, quando si cercò di superare il problema evocando la cosiddetta **Boltzmann machine**, un modello della rete di Hopfield dove uno stato può vedere *aumentata la propria energia secondo una legge probabilistica*, violando dunque l'assunzione di base tale per cui l'energia complessiva della rete può soltanto diminuire. Ciò viene attuato riscaldando in maniera *probabilistica* la rete, introducendo cioè una funzione *sigmoidale* dipendente da un parametro T, detto *temperatura del neurone*; le funzioni sigmoidali della macchina di Boltzmann sono progettate in modo tale che una temperatura più elevata corrisponda a sigmoidi meno rigide, e temperature che tendono allo 0 termico corrispondono a sigmoidi molto simili al tradizionale gradino.

Dunque, con il modello della macchina di Boltzmann, temperature della rete molto basse e prossime allo zero termico produrrebbero un comportamento della rete di Hopfield del tutto simile a quanto incontrato sino ad ora, cioè ad un'evoluzione nella quale non sono ammessi aumenti del livello dell'energia — viceversa, ad alte temperature del neurone si assiste mano a mano a potenziali fenomeni in cui, al passaggio successivo, la rete si è evoluta verso uno stato avente una maggiore energia rispetto a quello precedente. L'idea di base della macchina di Boltzmann è quella di poter "sfuggire" all'attrazione dovuta ai minimi locali, introducendo la possibilità probabilistica di poter "tornare indietro" e tentare una strada evolutiva differente. La probabilità con cui un determinato stato possa sfuggire dal minimo locale in cui risiede dipende dalla temperatura T — più essa è alta, maggiore sarà la probabilità per lo stato di aumentare la propria energia e percorrere un'evoluzione differente.

Supponendo di disporre di un valore di attivazione $A = \sum_{j \neq i} T_{ij} V_j - U_i$ pari a $A = +0.5$, e una temperatura di $T = 0,25$, avremo che la probabilità di eccitazione del neurone sarà pari a $0,78$ invece che pari ad 1, come espresso in Figura 3.5. Dunque, si dà la possibilità al neurone di *risalire la china dell'energia*, violando il comportamento originale che invece aveva nella rete di Hopfield — con questo stratagemma, la rete può probabilisticamente "scappare" dagli stati di minimo locali non desiderati. Si aggiungono dunque nuovi percorsi che con una certa probabilità dipendente dalla temperatura T *aumentano l'energia della rete*: valori più alti di temperatura garantiscono una maggiore probabilità di aumento dell'energia.

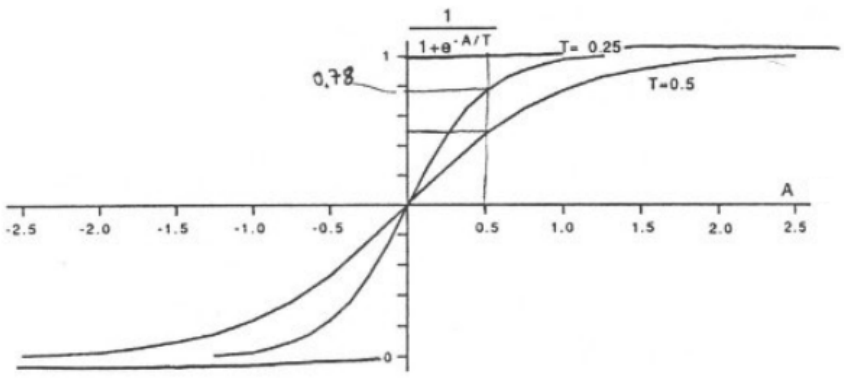


Figura 3.5: La funzione di attivazione probabilistica, con la sua tipica forma "ad esse" e comportamento dipendente dalla temperatura T. La sua forma sarà tanto più simile al gradino ideale quanto la sua temperatura sarà tendente allo 0.

Questo procedimento è chiamato **simulated annealing**, termine tratto dalla metallurgia (ricottura), dove variazioni di temperatura di un materiale sono adoperate per ottenere determinate caratteristiche dal materiale in questione. In particolare, la rete di Hopfield sarà inizialmente molto calda, per poi raffreddarsi progressivamente. La Figura 3.6 illustra un esempio di rete dove esiste una possibilità di "risalire la china", o più precisamente c'è una probabilità non nulla di ritornare ad un livello di energia superiore.

E dunque grazie allo stratagemma della macchina di Boltzmann è possibile progettare una rete di Hopfield che sia quantomeno robusta al problema dei minimi locali, introducendo di fatto una sorta di *perturbazione* alla funzione di attivazione, che rende concretamente e probabilisticamente possibile percorrere strade nuove e che consentono alla rete di "fuoriuscire" dalla situazione di stallo causata dallo stato di minimo locale in cui si troverebbe imprigionata.

3.3 La rete continua di Hopfield

La **rete continua di Hopfield** rappresenta un cambio di paradigma: mentre prima la verifica sullo stato veniva fatto in tempi discreti, assumendo una certa probabilità di interrogazione di un neurone, nella rete continua ogni neurone è, con continuità, connesso ad ogni altro neurone e la verifica della loro coerenza avviene istante per istante,

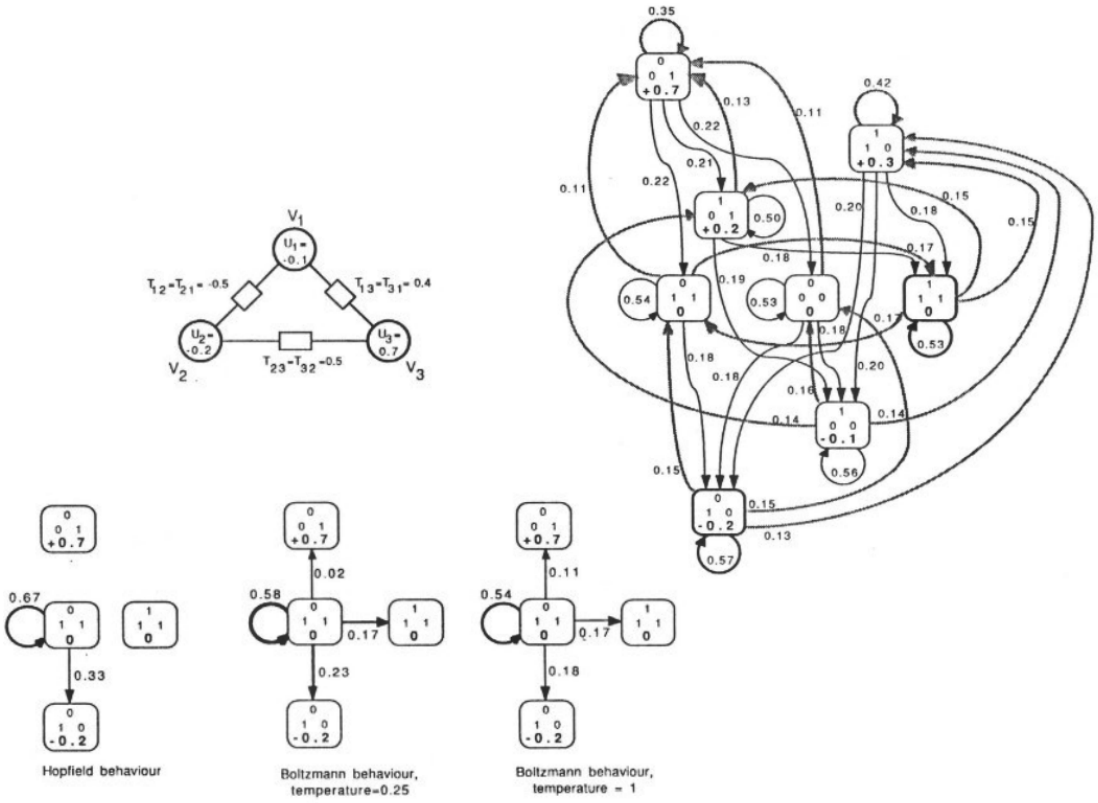


Figura 3.6: Probabilità di transizione a differenti temperature T (in basso); rete di Hopfield con estensione a macchina di Bolztmann (sopra).

mediante un *circuito analogico*, con un amplificatore operazionale posto a produrre la funzione di attivazione.

L'idea quindi è quella di lasciar evolvere il sistema composto dalla rete neurale, facendo sì che le grandezze elettriche in questione sviluppino il loro transitorio completamente: il risultato finale sarà la rete a minor energia.

Il valore d'uscita dipende da una funzione sigmoidale, ad esempio

$$V_i = g_i(u_i(t)) = \tanh(\frac{u_i(t)}{u_0}).$$

Alle conduttanze d'ingresso, viene aggiunto un gruppo circuitale resistenza-condensatore-generatore di corrente che produce correnti parallelamente a quelle di ingresso, cioè che si somma a quelle di ingresso. La totale conduttanza di neurone sarà pari a

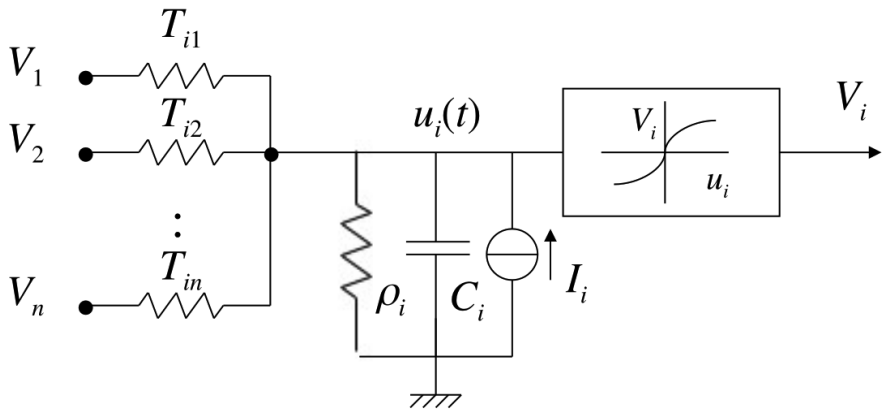


Figura 3.7: La *rete continua di Hopfield*.

$$\frac{1}{R_i} = \frac{1}{\rho_i} + \sum_j \frac{1}{R_j},$$

e tramite le leggi di Norton, si ottiene l'equazione al nodo d'ingresso del neurone,

$$\sum_j (V_j - u_j(t))T_{ij} + I_i = C_i \frac{du_i(t)}{dt} + \frac{u_i(t)}{\rho_i}$$

mentre l'energia della rete sarà

$$E = -\frac{1}{2} \sum_{ij, j \neq i} T_{ij} V_i V_j + \sum_i \frac{1}{R_i} \int_0^{V_i} g_i^{-1}(V) dV - \sum_i I_i V_i. \tag{3.2}$$

Calcolando la variazione dell'energia,

$$\frac{dE}{dt} = \sum_i \frac{dE}{dV_i} \frac{dV_i}{dt},$$

pertanto si avrà che

$$\begin{aligned}
\frac{dE}{dV_i} &= -\sum_j T_{ij}V_j + \frac{u_i}{R_i} - I_i = -C_i\frac{du_i}{dt} \\
\frac{dE}{dt} &= -\sum_i \left(\sum_j T_{ij}V_j + \frac{u_i}{R_i} - I_i\right)\frac{dV_i}{dt} = -\sum_i C_i\frac{du_i}{dt}\frac{dV_i}{dt} \\
&= -\sum_i C_i(\frac{du_i}{dV_i})\left(\frac{dV_i}{dt}\right)^2 \\
&= -\sum_i g_i^{-1}(V_i)C_i\left(\frac{dV_i}{dt}\right)^2 \leq 0,
\end{aligned}$$

dove si è osservato che $g_i^{-1}(V_i) > 0, \forall V_i$ poiché monotona crescente. In definitiva la variazione netta di energia della rete sarà

$$\frac{dE}{dt} \leq 0, \tag{3.3}$$

con l'uguaglianza a zero rispettata solo ed esclusivamente nel caso in cui $\frac{dV_i}{dt} = 0, \forall i$. Dunque, anche nel caso della rete continua di Hopfield la variazione di energia sarà sempre negativa.

In ultima analisi, *il tempo di risposta* di una rete di Hopfield dipende esclusivamente dalla costante di tempo $\tau = C_iR_i$ del neurone, poiché per ottenere una soluzione è necessario che il transitorio dell'evoluzione della rete abbia termine. Di fatto, il tempo di risposta non dipende dalla complessità della rete, bensì unicamente dalla costante di tempo — diversamente per quanto accadeva con le reti discrete di Hopfield.

Degno di nota è il fatto che il rapporto ingresso-uscita è fissato da una sigmoide, non è garantito che il valore d'uscita sia ad un valore nettamente basso o nettamente alto, cioè non è garantito un perfetto comportamento a soglia. Esistono infatti valori in concomitanza con lo 0 d'ingresso che possono suscitare dubbi interpretativi, cioè il neurone potrebbe collocarsi ad un valore intermedio fra valore alto e basso.

3.4 Soluzione dei problemi con la rete di Hopfield

Con una rete di Hopfield si può risolvere un problema solo se esso può essere codificato nei termini della *minimazione di una forma quadratica*. La soluzione sarà *ottima* se l'energia ottenuta è quella corrispondente al minimo assoluto — viceversa, sarà *sub-ottimale* se l'energia ottenuta corrisponderà invece a quella di un minimo locale, o comunque di uno stato stabile vero o falso che sia ad energia maggiore di quella fornita dal minimo globale.

La strategia è la seguente: individuata la forma quadratica associata al problema, per confronto si ricavano i valori T_{ij} ed I_i che consentono la costruzione della rete. Successivamente, la rete è lasciata evolvere; la soluzione corrispondente viene prelevata e il problema viene risolto con la soluzione ricavata mediante la codifica.

3.4.1 Il problema della memoria indirizzabile

Si desidera che $V^s (s = 1, 2, \ldots, n)$ siano stati stabili in una rete dotata di n neuroni.

La quantità da minimizzare sarà $\hat{E} = -\frac{1}{2} \sum_s (V^s \cdot V)^2$, con $V = V^s$ valore negativo di energia minimo: ciò accade poiché siamo di fronte ad un prodotto scalare che massimizza la quantità all'interno della sommatoria, con il risultato di minimizzare il valore dell'energia.

Se V è casuale e $(-1 \leq V_i \leq +1)$, la $\hat{E}$ risulterà essere circa nulla – eseguendo un prodotto scalare fra due vettori sostanzialmente molto differenti; il valore minimo si raggiunge con $V = V^s$, ed è tale che $\hat{E} = -\frac{1}{2} n^2$.

Bisogna in pratica fissare anticipatamente i V^s di stato stabile, e con essi si imposta l'equazione

$$E = -\frac{1}{2} \sum_s (V^s V)^2 = \cdots = -\frac{1}{2} \sum_i \sum_j (\sum_s V_i^s V_j^s) V_i V_j,$$

ma sappiamo anche che

$$E = -\frac{1}{2}\sum_{ij} T_{ij} V_i V_j - \sum_i I_i V_i.$$

Dal confronto fra le due equazioni è presto ricavato che

$$\begin{cases} T_{ij} &= \sum_s V_i^s V_j^s \\ I_i &= 0 \end{cases}$$

cioè abbiamo trovato la conduttanze e il valore del generatore ideale di corrente per poter costruire la rete continua di Hopfield associata al problema impostato come minimazione di un funzionale quadratico. Innescata la rete, essa andrà presumibilmente (a meno di falsi stati stabili) a stabilirsi in uno degli stati V^s determinati dal progettista.

3.4.2 Il problema delle somme parziali

Il problema delle somme parziali è sostanzialmente intrattabile. Vi sono alcuni interi $a = a_1, a_2, \ldots, a_n$ e alcuni numeri binari $V = V_1, V_2, \ldots, V_s, V_i \in \{0, 1\}$, con S risultato del prodotto scalare fra a e V e si chiede di trovare il vettore V sapendo che

$$a \cdot V = S$$

e conoscendo a ed S. In altre parole, si chiede di invertire il prodotto scalare trovando il vettore di binari V tale da fornire il risultato S a fronte del prodotto con il vettore a di numeri interi, anch'esso noto a priori.

Il funzionale quadratico da minimizzare sarà il seguente,

$$E = \frac{1}{2}(S - \sum_i a_i V_i)^2 - \frac{1}{2}\sum_i a_i^2 V_i (V_i - 1), \tag{3.4}$$

dove il secondo termine rappresenta le condizioni al contorno tali per cui esso è nullo esclusivamente nel caso in cui $V_i \in \{0, 1\}$, ovvero esso funge da *penalità* per tutti i termini tali che $V_i \neq 0$ oppure $V_i \neq 1$, in quanto termine positivo da sommarsi.

Dunque,

$$\begin{aligned} E &= \frac{1}{2}(S - \sum_i a_i V_i)^2 - \frac{1}{2}\sum_i a_i^2 V_i(V_i - 1) \\ &= -\frac{1}{2}\sum_i \sum_{j \neq i} (-a_i a_j) V_i V_j - \sum_i (-\frac{a_i^2}{2} + S a_i) V_i + \frac{S^2}{2}, \end{aligned}$$

e dunque si avrà

$$T_{ij} = \sum_{j \neq i} (-a_i a_j),$$

$$I_i = \sum_i (-\frac{a_i^2}{2} + S a_i).$$

Supponendo di avere $a_i = 2^i$ la soluzione si risolve in tempo polinomiale, ed ha un *unico minimo assoluto*. Vi è una forte dipendenza fra la natura del problema e la quantità di minimi locali — se la complessità del problema è elevata, la risoluzione con la rete di Hopfield diventa maggiormente difficile a fronte di un più grande numero di minimi locali. La difficoltà del problema viene pertanto in qualche modo incastonata nella *complessità strutturale* della rete e dunque in ultima analisi la **dimensione della rete dipende dalla complessità del problema**.

Se si intende risolvere il problema delle somme parziali con $n = 1000$, la rete prodotta sarà di enormi dimensioni e complessità, con ciascun coefficiente T_{ij} e generatore ideale di corrente I_i da determinare di conseguenza. Diversamente dal caso del paradigma a computazione procedurale, dove la complessità è di tipo *temporale*, nel paradigma a reti neurali l'aspetto temporale è invece di secondaria rilevanza, poiché è sempre possibile scegliere un valore di capacità sufficientemente piccolo (ad esempio, dell'ordine dei pico-Farad[4]) ed ottenere una risposta della rete pressoché istantanea. La complessità delle reti neurali, però, si sviluppa lungo la dimensione della loro *struttura*.

[4]Si ricorda a tal proposito che $\tau = C_i R_i$ con R_i totale resistenza di ingresso del neurone.

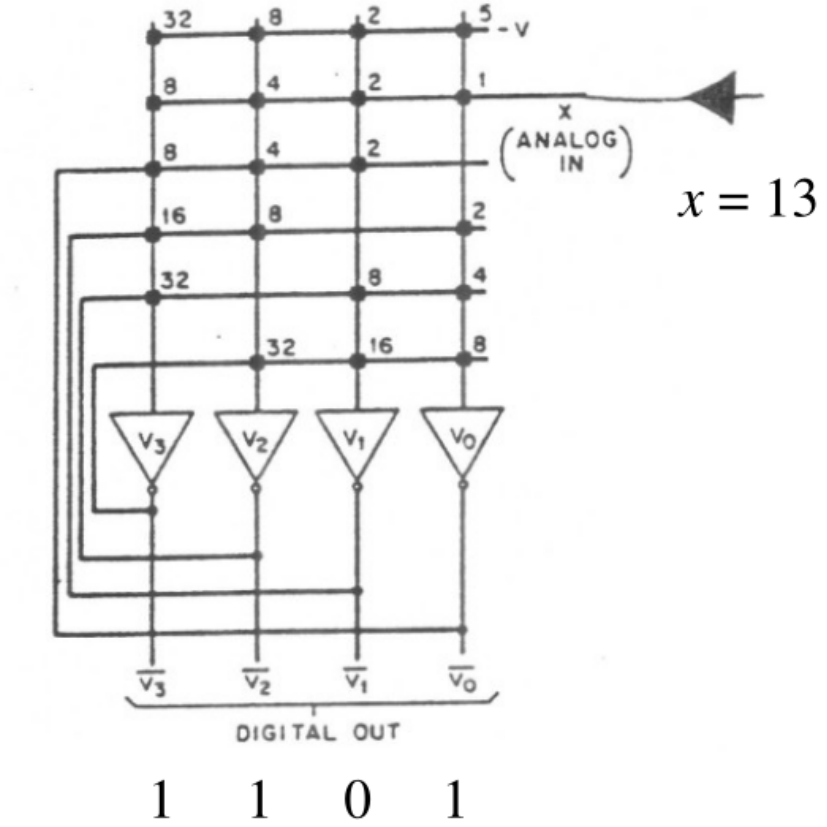


Figura 3.8: Un dispositivo convertitore Analogico-Digitale. Esso può essere modellato mediante una rete continua di Hopfield.

3.4.3 Il traveling salesman problem

Il traveling salesman problem è un problema presumibilmente intrattabile – benché esistano ottimi algoritmi euristici di tipo approssimato, la migliore soluzione esatta viene ricoperta in tempo almeno esponenziale.

Supponendo di trovarci di fronte n città A, B, C, …, e di possedere le distanze reciproche $d_{AB}, d_{AC}, d_{BC}, \ldots$, e così via. La tecnica adoperata per la codifica della soluzione è quella di una *matrice città—-percorso ottimale*, ad esempio

è la matrice che corrisponde alla codifica del percorso C-A-E-B-D.

	1	2	3	4	5
A	0	1	0	0	0
B	0	0	0	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

Dunque, esprimendo il funzionale dell'energia da minimizzare, si ottiene

$$\begin{aligned} E &= \frac{\alpha}{2}\sum_x\sum_i\sum_{j\neq i} V_{X_i}V_{X_j} + \frac{\beta}{2}\sum_i\sum_X\sum_{X\neq Y} V_{X_i}V_{Y_i} \\ &+ \frac{\delta}{2}(\sum_x\sum_i V_X - n)^2 + \frac{\gamma}{2}\sum_X\sum_{X\neq Y}\sum_i d_{XY}V_{X_i}(V_{Y_{i+1}} + V_{Y_{i-1}}). \end{aligned}$$

Si tratta di un funzionale particolarmente complesso, composto da 4 distinti termini:

1. il primo termine è la condizione al contorno per le righe, pari a zero se ogni riga contiene esclusivamente un singolo 1;
2. il secondo termine è la condizione al contorno per le colonne, pari a zero se ogni colonna contiene esclusivamente un singolo 1;
3. il terzo termine conta il numero di 1: esso è pari a zero solo se vi sono *esattamente* n termini 1 nella matrice;
4. il quarto ed ultimo termine, infine, è quello necessario per minimizzare il funzionale relativo al percorso.

3.5 Sommario delle caratteristiche delle reti neurali

A.K. Dewdney 1997,

"Although neural nets do solve a few toy problems, their powers of computation are so limited that I am surprised anyone takes them seriously as a general problem-solving tool."

I pregi delle reti neurali di Hopfield sono di seguito elencati,

- esse rappresentano un "nuovo" ed interessante *paradigma computazionale*;
- sono *robuste rispetto ai guasti* dei singoli neuroni;
- le risposte sono *in tempo reale*, nel caso di realizzazione "fisica" con hardware;
- forniscono un'interessante *metafora biologica*.

I difetti invece rendono molto difficile il loro utilizzo. In particolare,

- manca una adeguata *formalizzazione matematica*;
- non vi sono *garanzie sulla qualità* della soluzione trovata, non si ottengono in genere soluzioni ottime;
- soffrono del *problema dei falsi minimi*;
- sono inferiori ad *algoritmi dedicati*;
- necessitano di un'*altissima complessità strutturale* che dipende dall'istanza del problema;
- *dipendono fortemente dai pesi del problema*: cambiando **istanza** del problema, cambiano **tutti i coefficienti**, rendendo di fatto **praticamente irrealizzabili** queste reti se dotate di una complessità apprezzabile.

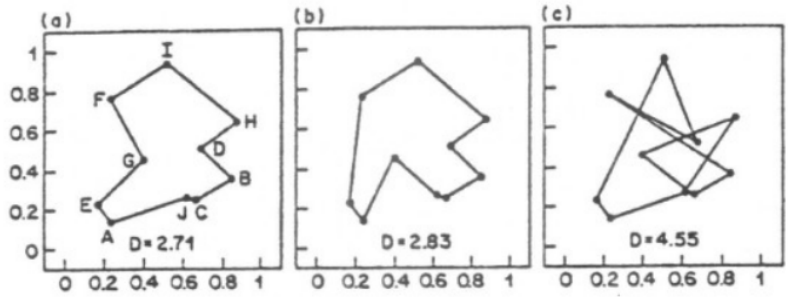


Fig. 3a–c. **a, b** Paths found by the analog convergence on 10 random cities. The example in **a** is also the shortest path. The city names *A* ... *J* used in Fig. 2 are indicated. **c** A typical path found using a two-state network instead of a continuous one

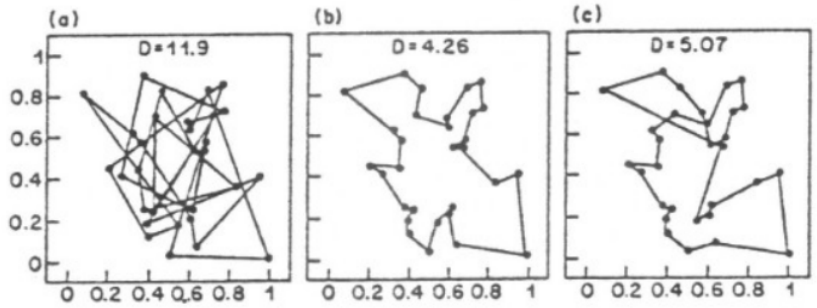


Fig. 4a–c. **a** A random tour for 30 random cities. **b** The Lin-Kernighan tour. **c** A typical tour obtained from the analog network by slowing increasing the gain

Figura 3.9: Esempi della risoluzione del travelling salesman problem con reti neurali di Hopfield.

Chapter 4
La computazione DNA

La **computazione DNA** è un metodo di computazione che corrisponde ad una procedura di *ricerca esauriente*: tutte le possibili soluzioni vengono costruite in modo sistematico, e vengono poi analizzate "in vitro", cioè una per volta, per poi conservare esclusivamente le soluzioni desiderate. Non si tratta, perciò, di una nuova tecnica con differenze dal punto di vista strutturale, bensì di un metodo molto efficace per realizzare una ricerca esauriente della soluzione.

Il modello a computazione DNA adotta il DNA biologico come contenitore delle informazioni: le sue dimensioni sono talmente ridotte da permettere una ricerca esauriente in spazi ridottissimi. La ricerca esauriente non viene adottata mediante un classico algoritmo, bensì in modo "nativo" tramite l'uso dei DNA.

Storicamente, le scienze matematiche e quelle naturali si sono evolute seguendo due rami ben distinti. Attorno agli anni 50, la biologia molecolare scoprì il DNA — immediatamente furono aperte nuove strade, compresa quella dell'*informatica nella biologia*. In particolare, furono usati risultati dell'*algoritmica* per trattare le lunghissime sequenze del DNA e il ripiegamento delle proteine e t-RNA, ambedue problemi più facilmente trattabili con le nozioni avanzate dell'informatica teorica e più precisamente dell'algoritmica. Non molto tardi, le *misure dell'informazione* figlie della teoria dell'informazione furono adottate anche per la costruzione degli *alberi filogenetici*, diagrammi che mostrano le relazioni fondamentali di discendenza comune di gruppi tassonomici di organismi.

Più tardivamente, ovvero attorno alla metà degli anni 90, furono aperte strade nella direzione opposta: il sequenziamento del DNA fu di ispirazione per il nuovo modello di computazione appunto "a DNA" (Adleman, 1994). Allora fu sottolineata la capacità di manipolare le stringhe astratte dell'informatica tramite una *codifica nel DNA* — senza occuparsi del significato della struttura chimica, fisica, biofisica e genetica propriamente attribuiti alle molecole del DNA. Implicitamente si intende adoperare il DNA per contenere informazioni tramite i simboli (codificati) nel DNA, piuttosto che per scopi "biologici" e legati a quanto si faceva in passato.

L'idea è quella di risolvere con una computazione DNA certi problemi che sono *strutturalmente difficili*, per i quali algoritmi efficienti non sono noti. Esempi di tali problemi sono il *problema delle somme parziali*, il *problema della fattorizzazione di un intero*, e la determinazione del *cammino Hamiltoniano su un grafo*.

I DNA sono basati su lunghissime sequenze di lettere di un *alfabeto finito*, avente 4 simboli (A, C, G, T), detti *nucleotidi*. Anche le proteine sono basate su catene molecolari, in particolare su un alfabeto anch'esso finito di 20 simboli, detti *amminoacidi*. La codifica di queste catene di simboli può essere effettuata, come è ragionevole aspettarsi, mediante tecniche informatiche.

Ad esempio, di seguito sono riportate i 20 amminoacidi

Alanina (Ala, A)

Acido glutammico (Glu, E)

Arginina (Arg, R)

Glutammina (Gln, Q)

Asparagina (Asn, N)

Istidina (His, H)

Acido aspartico (Asp, D)

Isoleucina (Ile, I)

Cisteina (Cys, C)

Leucina (Leu, L)

Metionina (Met, M)

Fenilalanina (Phe, F)

Prolina (Pro, P)

Serina (Ser, S)

Treonina (Thr, T)

Tirosina (Tyr, Y)

Valina (Val, V)

Triptofano (Trp, W)

Lisina (Lys, K)

Glicina (Gly, G)

il cui alfabeto può essere opportunamente codificato. Con l'adeguata codifica, esse sono perciò in grado di *immagazzinare informazioni*.

La computazione DNA è tipicamente adoperata per la soluzione di problemi *presumibilmente* o *sostanzialmente intrattabili* — problemi di difficile soluzione, di cui non sono noti ad oggi algoritmi efficienti.

Dato il **problema delle somme parziali** descritto in Sezione 3.4.2, è noto che non è facile trovare gli elementi costituenti la somma parziale.

Un altro problema di difficile soluzione è quello della **fattorizzazione di un intero**: dati p e q interi primi, molto grandi, è facile calcolare

$$n = p \cdot q,$$

mentre è estremamente difficile calcolare p e q a partire dal solo n.

Un ultimo problema, che verrà analizzato di seguito, è il **problema del cammino Hamiltoniano su un grafo**, risolto con la computazione DNA proprio da Adleman nel 1994.

4.1 Uso dell'informatica per la risoluzione di problemi di Biologia molecolare

Leggere il DNA tutto di seguito è tecnologicamente impossibile — ciò che si può fare è leggere sequenze di qualche centinaio di basi. Diventa dunque di estrema rilevanza il problema dell'**il problema dell'assemblaggio dei frammenti di DNA**.

Supponendo di voler sequenziare il frammento di DNA

`AGTATTGGCAATCGATGCAAACCTTTTGGCAATCACT`, si possono adoperare i cosiddetti *enzimi di restrizione* per estrarre alcune sequenze. Tipicamente, si estraggono varie sequenze che vanno riordinate, e ritagliate nel caso vi siano delle sovrapposizio-

ni. Ricostruire la sequenza esatta del DNA a partire dai lembi di poche centinaia di basi è molto difficile: in un vero esperimento biologico si hanno lunghezze di DNA da 30000 a 100000 basi, con lunghezze di frammenti che crescono fino a 700 basi; il numero di frammento totale sarà all'incirca dell'ordine di qualche migliaio.

La strategia che si adopera è quella dell'*allineamento risolutivo*, cioè quella di andare ad allineare i frammenti a seconda delle loro somiglianze (string matching).

```
AGTATTGGCAATC--------CCTTTTGG--------
---------AATCGATG--------TTGGCAATCACT
--------------ATGCAAACCT-------------
AGTATTGGCAATCGATGCAAACCTTTTGGCAATCACT
```

Il secondo problema è quello della **ricerca di moduli funzionali del DNA**, dove è richiesto di trovare i moduli funzionali del gene avendo sequenze di DNA relative allo stesso gene di organismi differenti. Un modulo funzionale è una porzione del DNA che svolge un qualche ruolo nella dinamica biochimica della cellula. Non tutto il DNA viene effettivamente adoperato per i principi metabolici — tipicamente, molte parti del DNA sono *ridondanti*. Attraverso i moduli funzionanti si svolgono le dinamiche interessanti del funzionamento della cellula. Un esempio di ciò sono **i geni** — di lunghezza molto più corta del DNA, grazie ad essi sono realizzate le proteine per cui si ha il funzionamento dell'organismo biologico.

I moduli funzionali nel corso delle generazioni possono subire piccole variazioni nella loro struttura, sia per via dell'evoluzione che per raggi cosmici o altre radiazioni ionizzanti, presentando lacune o lettere differenti.

```
----------------------AG--TTGTCAATCTTGGCAATCACT
ATCGATTCCGGCCCTTTTGG--AGTATTGGCAATCATGCAAACCT--
---GTCGATCGAATGCAAACCTAGTAATGGCAATCCTATGCTCGATC
-----------------GTACCAGTATTGGCA-TCTGC---------
----------ATCTTCAATTATAGTATTGACAATCTCGCTAGTCG--
---CCGGCTCTAAAATTTGGGGAGTGTTGGCAATCATTCCGGCTCTA
------------ATTGCATGCAAG-ATTGGCAATCATCGA-------
----CGTCGATCCAATTTCAGCAGTAT--GCAATCATTCGTCGATCG
```

Pur essendo i moduli di cui sopra formalmente diversi, essi manifestano un patrimonio comune, presentando una grossa quota di nucleotidi simili dal punto di vista della *Levenshtein distance* (distanza fra stringhe simile alla distanza di Hamming fra numeri binari). Dunque, dall'esempio sopra si individua il modulo funzionale `AGTATTGGCAATC`.

Per i problemi di cui sopra, la struttura dati importante è quella dello **suffix tree** (albero a suffisso). Viene assegnata una stringa, e viene chiesto di trovare la più lunga sottostringa ripetuta. Un suffix tree può essere costruito in tempo lineare, rendendo possibile l'individuazione delle sottostringhe da ricercare appunto in tempo lineare.

Si parte da sinistra, ricopiando l'intera stringa. Ogni stringa dell'albero viene etichettata con la posizione sulla stringa dalla quale la lettera è stata determinata (con il *suffisso*), e ad ogni nuova lettera si apre un nuovo ramo. Ogni qualvolta si ripresenta una medesima lettera, il percorso viene riutilizzato: si aprirà un ramo solo nel punto dove i due cammini in comune presentano la prima differenza, ed il nuovo ramo è etichettato a sua volta con il suffisso indicante la posizione del nodo sulla stringa.

L'interrogazione dell'albero avviene individuando il *nodo a profondità massima* dell'albero, poiché fino a quel nodo la stringa si è conservata la stringa la stringa a lunghezza massima su almeno due percorsi.

TODO aggiungi alberi a suffisso 17 18

4.1.1 Il problema dei legami A-T e C-G e del ripiegamento delle proteine o del t-RNA

Il DNA ha una struttura "a scala" o "a doppia elica" — nella struttura vi è una *complementarietà* fra guanina e citosina, e adenina e tiamima. Le due catene saranno (e devono essere) complementari, perciò la doppia catena avrà un *verso* che va da $5'$ a $3'$.

Il terzo problema è dunque quello del **ripiegamento delle proteine o del t-RNA**. Data la sequenza primaria, bisogna determinare la struttura secondaria, cioè trovare mediante la complementarietà A-T, G-C la stringa secondaria ad essa associata. Bisognerà dunque determinare con tecniche informatiche la maniera in cui la sequenza si "ripiega" su se stessa.

TODO struttura t-RNA 20

Il medesimo problema si ha, ad esempio, per il ripiegamento della proteina (ancor più difficile, poiché presentano ripiegamenti senza complementarietà e tridimensionali).

4.1.2 La costruzione di alberi filogenetici

La costruzione degli *alberi filogenetici* è un problema di distanza fra stringhe proteiche. Si può adoperare la distanza di Levenshtein per determinare se due sequenze hanno un *avo comune*, cioè se ambedue le sequenze derivano da una sequenza ancestrale. In altre parole, si individua una misura di distanza fra stringhe che sia anche *biologicamente significativa*.

I trasposoni (le sequenze) vengono allineate, e si effettua una ricerca approssimata per rilevare le somiglianze fra le due stringhe. Le somiglianze possono essere sia *esatte*, cioè porzioni di stringhe sono allineate e identiche fra le due stringhe, oppure presentare *amminoacidi affini* (indicati con un +). Gli amminoacidi imparentati presentano un grado di parentela, espresso nella **Matrice BLOSUM**. La Matrice BLOSUM contiene un valore intero associato ad ogni coppia di amminoacidi — tanto più alto il valore associato alla coppia, tanto più la coppia sarà biologicamente affine. La somma completa di parentela fra tutti gli amminoacidi dei trasposoni conferisce un punteggio globale BLOSUM, da interpretare tramite conoscenze di Biologia.

TODO aggiungi figure 22+

4.2 La computazione mediante DNA

Le dimensioni di DNA e proteine sono estremamente ridotte: in poche frazioni di millimetro cubo è infatti possibile immagazzinare una quantità enorme di informazione sotto forma di sequenze di simboli. Tanto minori saranno le dimensioni richieste dal DNA, tanto maggiori saranno le capacità computazionali di tale sistema. Il trucco sarà prima di sfruttare la cellula — molto efficiente nella gestione di A, C, G, T — per immagazzinare l'informazione, e successivamente di adoperare il calcolatore e tecniche informatiche per manipolare successivamente le sequenze di simboli.

Adleman propose nel 1994 un metodo per la costruzione di un **calcolatore bio-molecolare**, basato su un modello di computazione *non* algoritmico procedurale, benché il paradigma sia lo stesso (ricerca esauriente).

4.2.1 Il problema del cammino Hamiltoniano su un grafo

Il problema del cammino Hamiltoniano su un grafo è esprimibile nella seguente maniera: dato un grafo con n vertici, $v_1, v_2, \ldots, v_n$ ed m archi, fissati due vertici di inizio e fine v_i e v_f, verificare se esiste un cammino che va da v_i a v_f con $n-1$ archi che tocchi ciascun vertice una ed una sola volta.

Per il problema del cammino Hamiltoniano **non sono noti** algoritmi efficienti: resta soltanto la soluzione bruta, cioè generare tutte le $(n-2!)$ permutazioni dei vertici, escludendo v_i e v_f, e verificare l'esistenza della soluzione. Inoltre,

$$n! \sim \sqrt{2}\pi\sqrt{n}e^{-n}n^n,$$

il che evidenzia che è un problema praticamente intrattabile con n grande, poiché anche i migliori algoritmi hanno complessità minima dell'ordine di 2^n. Già infatti con $n = 85$, si ottiene una complessità di $2^{85} = 3,86 \cdot 10^{25} op$, le quali richiedono oltre un miliardo di anni operando 10^9 operazioni al secondo!

La soluzione adottata per il problema del cammino Hamiltoniano è la seguente

1. si codificano nodi ed archi in sequenze di DNA;
2. si generano catene di DNA casuali, ciascuna di esse sarà associata ad un percorso — ciò si fa lasciando le sequenze in un contenitore archi-vertici, con l'aggiunga dell'*enzima Ligasi*;
3. si prendono soltanto quelli con il giusto nodo d'inizio v_i e finale v_f, si scartano gli altri poiché fondamentalmente errati – ciò è realizzato mediante *polymerase chain reaction*;
4. si trattengono tutti quelli che visitano n vertici — mediante *elettroforesi con gel di agarosio*[1]; item si trattengono solo quelli che visitano ciascun vertice almeno una volta — si formano singoli strand di DNA accoppiati con tutti i vertici;
5. i cammini rimanenti sono *la soluzione* — estratti in laboratorio.

[1]La gelatina di agarosio (uno zucchero), se sottoposta a differenza di potenziale, fa sì che le molecole si polarizzino e si muovano all'interno del gel. Per via della viscosità del gel, le molecole più corte viaggiano più velocemente, mentre le molecole maggiormente lunghe saranno complessivamente più lente. Dopo un lasso di tempo si può controllare il posizionamento delle molecole, che formeranno agglomerati e saranno disposte in cluster a seconda della lunghezza. Ecco che così facendo si potrà discernere fra i percorsi "troppo lunghi" e "troppo corti" e quelli della lunghezza corretta, che saranno collocati nello stesso cluster.

TODO figure 30

Per costruire i *vertici* si determinano stringhe casuali assieme alle loro inverse complementari. Supponiamo di costruire n stringhe casuali di lunghezza pari a 20 nucleotidi, da $5'$ a $3'$. Il primo passo è quello di determinare le stringhe *inverse complementari*, cioè quelle da sovrapporre e tali che $3' \rightarrow 5'$: le due stringhe potranno incollarsi, poiché complementari.

Per costruire gli *archi* $u \rightarrow v$ S_{uv}, si procede costruendo una stringa di lunghezza 20 con la coda del nodo u (S_u) e con la testa del nodo v (S_v).

Si generano dunque moltissime (milioni di) copie delle $\overline{S}_j$ complementari e si mettono nel *contenitore dei vertici*, mentre si generano moltissime (anch'esse milioni di) copie delle S_{uv}, ponendole nel *contenitore degli archi*. Per la computazione, semplicemente, *si mescolano i contenitori*, e se per caso le sequenze si attaccano le une le altre, queste formano una sequenza di vertici e nodi: abbiamo dunque realizzato la seconda fase di generazione casuale dei percorsi.

TODO figure 33 34

Supponiamo di veder realizzato il cammino u-v-w. Gli archi sono costituiti con metà di entrambi i nodi, mentre i vertici sono casualmente generati. Nel contenitore dei DNA le sequenze possono muoversi liberamente, ed incrociarsi a seconda delle affinità: può crearsi, in definitiva, una catena come in Figura **??**.

4.2.2 I vantaggi e le limitazioni delle tecniche a computazione DNA

I vantaggi *teorici* principali sono legati alla *velocità di computazione*, milioni di volte più rapida dei computer tradizionali, all'*efficienza energetica* di miliardi di volte maggiore, e specialmente alla capacità di *memorizzazione ad altissima densità* (miliardi di volte maggiore). Infatti, $1cm$ cubo può contenere fino a 10000 mld di molecole di DNA; 1 bit è contenuto in $1nm$ cubo col DNA, mentre in $10^{12}nm$ cubi con CD. Un calcolatore DNA potrebbe gestire $10TB$ di memoria con una velocità di calcolo di 10^{13} operazioni al secondo.

Le limitazioni sono purtroppo pesantissime: il numero di molecole implicate è lo stesso delle soluzioni possibili nella ricerca esauriente, cioè all'incirca $\sim 2^n$. Purtroppo,

scegliendo $n = 100$ il peso del DNA sarebbe superiore a $10t$, mentre con $n = 200$ il totale peso delle stringhe richieste sarebbe superiore al peso della Terra!

L'altra limitazione sono le *imperfezioni nelle tecniche di sintesi del DNA*, comportando una presenza non trascurabile di errori. Inoltre, il bio-calcolatore deve essere *dedicato al problema*, ed il *tempo* di ogni operazione biologica richiede ore o giorni. In altre parole, la complessità temporale viene sostituita con la **complessità sul peso**.

Chapter 5
Teoria degli automi e dei linguaggi formali

Gli **automi a stati finiti** sono la classe delle macchine con minore potenza computazionale.

Un automa a stati finiti può essere rappresentato mediante una testina che si muove su un nastro contenente dei simboli e potenzialmente illimitato, sempre nella stessa direzione. La testina può trovarsi in un certo *stato* — a seconda dello stato q e del simbolo s_i letto sul nastro, la testina si porta in un altro stato o rimane nello stesso in cui si trovava, per poi spostarsi a destra per apprestarsi a leggere il simbolo successivo.

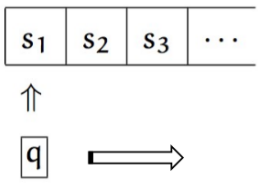


Figura 5.1: Rappresentazione schematica di un automa a stati finiti.

Quando la lettura dei simboli termina, a seconda dello stato raggiunto dalla testina, l'automa fornisce un risultato di *accettazione* o di *refutazione* della stringa.

Fra le macchine concepibili, le macchine a stati finiti sono le meno potenti. Volendole classificare in base alle caratteristiche e alle loro possibilità:

Macchina di Turing può leggere e scrivere a destra e sinistra della stringa;

Macchina di Post legge a sinistra, mentre scrive ed allunga la stringa a destra;

Macchine con memorie push-down leggono a sinistra di x, leggono e scrivono a sinistra di ciascuna memoria push-down. Le macchine con memorie push-down hanno differenti potenze computazionali a seconda del numero di tali memorie;

Automi, macchine a stati finiti leggono a sinistra, non possono scrivere. Eseguono essenzialmente delle *letture*, ma non delle scritture.

Gli automi non sono perciò in grado di scrivere qualcosa sul nastro. La memoria a disposizione sull'automa è una qualche sorta di *memoria di sola lettura* — il nastro supporta esclusivamente la stringa x, di lunghezza comunque finita. Un automa, dunque, può essere considerato come una macchina che svolge compiti in base alle istruzioni fornite sul nastro, oppure come una macchina che riconosce stringhe. Ci ritroviamo dunque in una situazione del tutto simile a quanto accadeva per la macchina di Turing, con la sola differenza che un automa è in grado di riconoscere un numero molto più limitato di stringhe.

5.1 I linguaggi regolari

La **chisura (o stellatura)** S^* dell'insieme S è l'insieme costituito dalla parola vuota e da tutte le parole formate concatenando un numero finito di parole in S, cioè quindi

$$S^* = S^0 \cup S^1 \cup S^2 \cup \dots$$

dove $S^0 = \{\Lambda\}$, e $S^i = S^{i-1}S, i > 0$.

Sia $\Sigma = \{a, b\}$ un alfabeto di simboli (contenente in questo caso due simboli), Λ sia la *parola vuota* e la **stellatura dell'insieme** $\Sigma^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$ definiamo il **prodotto (concatenazione)** UV di due sottoinsiemi U, V di Σ^* come

$$UV = \{x | x = uv, u \in U, v \in V\}.$$

In altre parole, ogni parola dell'insieme UV è formata concatenando una parola di U con una parola di V. Prendendo due sottoinsiemi $U = \{a, ab, aab\}$ e $V = \{b, bb\}$ allora l'insieme $UV = \{ab, abb, abbb, aabb, aabbb\}$. In generale, il prodotto non

è commutativo, nel senso che l'insieme ottenuto potrebbe non essere uguale. L'operazione prodotto è tuttavia associativa, cioè per qualunque $U, V, W \in \Sigma^*$ si ha che $(UV)W = U(VW)$.

Esaminiamo ora la classe degli **insiemi regolari**. La classe degli insiemi regolari su Σ è ricorsivamente definita come segue:

1. ogni insieme *finito* di parole su Σ, compreso l'insieme vuoto $\emptyset$, è un insieme regolare;
2. se U e V sono insiemi regolari su Σ, lo sono anche la loro unione $U \cup V$ ed il loro prodotto UV;
3. se S è un insieme regolare su Σ, lo è anche la sua chiusura S^*.

Dunque nessun insieme è regolare, a meno che non possa essere ottenuto con un numero finito di applicazione dei punti 1, 2 o 3. Perciò, la classe degli insiemi regolari su Σ è la più piccola classe che contiene tutti gli insiemi finiti di parole su Σ, e che è al contempo chiusa rispetto ad unione, prodotto e stellatura.

Gli insiemi regolari sono generalmente molto semplici: un sottoinsieme di Σ^* è regolare *se e solo se*

1. è accettato da qualche *automa finito*;
2. è accettato da qualche *grafo di transizione* — una definizione differente di un automa a stati finiti;
3. può essere espresso da qualche *espressione regolare*.

Esempi di espressioni regolari sono l'insieme di tutte le parole di $\Sigma = \{a, b\}$ che contengono o due a consecutive, o due b consecutive, e l'insieme delle parole di Σ che contengono un numero pari di a e pari di b. Al contrario, l'insieme $\{a^n b^n | n \geq 0\}$ *non* è un imsieme regolare su Σ.

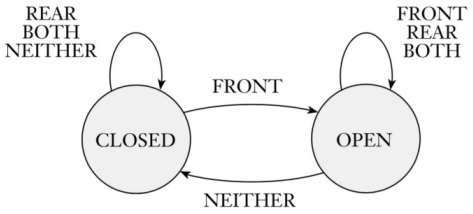
5.2 Gli automi

Un esempio reale di automa risiede nel controllore di un portone elettrico – la funzionalità dell'automa è insita nei suoi stati: il cancello può essere nello stato di APERTO oppure CHIUSO, rappresentando le corrispondenti condizioni del portone. Vi

sono quattro possibili condizioni di input, dipendenti dalla posizione delle persone relativamente al cancello: FRONTE, RETRO, ENTRAMBI, NESSUNO.

Due modalità di rappresentazione:

diagramma di stato



matrice di transizione

input signal

state	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

Figura 5.2: Doppia modalità di rappresentazione per un automa a stati finiti. Esso può essere rappresentato o come *diagramma di stato*, o come *matrice di transizione*.

Vi sono due possibili rappresentazioni per un automa: mediante il **diagramma di stato** o attraverso la sua **matrice di transizione**. La differenza fra quest'ultima e la matrice della macchina di Turing è che negli automi non è prevista un'operazione da svolgere — il sistema è capace solo di transitare verso uno stato successivo, non di effettuare operazioni particolari o di immagazzinare informazioni.

Siamo ora pronti a fornire la prima definizione di automa a stati finiti.

Un **automa finito** A su Σ, dove $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ è un grafo finito orientato in cui da ogni vertice escono n archi, con ogni arco etichettato con un diverso σ_i, con $1 \leq i \leq n$. Esiste un vertice, etichettato con il segno $-$, detto *vertice iniziale*, ed un insieme di vertici eventualmente vuoto etichettati con un segno $+$, detti *vertici finali*, con la possibilità che il vertice iniziale sia anche un vertice finale. Talvolta i vertici sono chiamati **stati**. Si guardino le slides per tutti gli esempi sugli automi.

La seconda definizione di automa finito è più formale, ed equivalente alla precedente.

Un **automa finito** è una 5-tupla $Q, \Sigma, \delta, q_0, \mathcal{F})$ dove

1. Q è un insieme finito di *stati*;

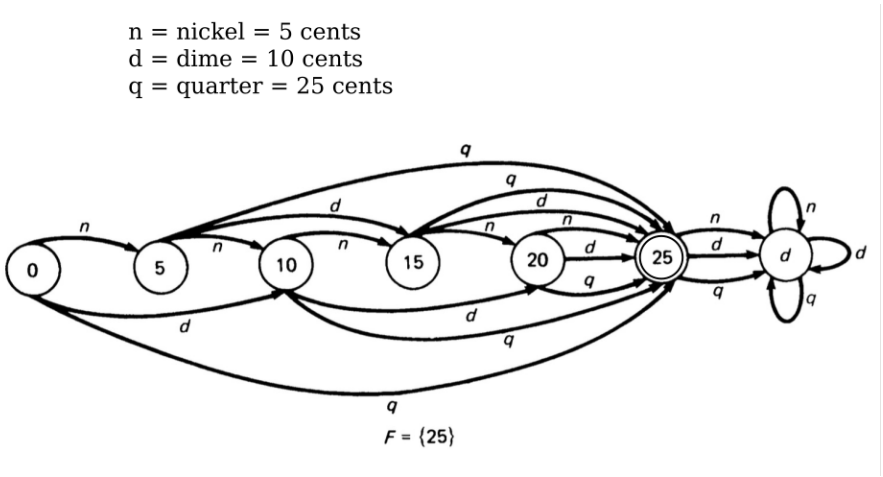


Figura 5.3: Macchina per la distribuzione di caramelle negli Stati Uniti, dal costo di 25 cents. Se vengono inseriti più di 25 cents, la macchina non dà resto.

2. Σ è un insieme finito, detto *alfabeto*;
3. $\delta : Q \times \Sigma \to Q$ è la *funzione di transizione*;
4. q_0 è lo *stato di partenza*;
5. $\mathcal{F} \subset Q$ è l'*insieme degli stati di accettazione*.

Se A è l'insieme di tutte le stringhe accettate dalla macchina M, si dice che A è il **linguaggio della macchina** M, e scriviamo $\mathcal{L}(M) = A$. Nell'esempio di cui sopra,

$\mathcal{L}(M) = A = \{w | w$ contiene almeno un 1 ed un numero pari di 0 seguono l'ultimo $1\}$.

5.3 La computazione degli automi deterministici (DFA) dal punto di vista formale

Sia $A_D = \{Q, \Sigma, \delta_D, q_0, \mathcal{F}\}$. Se $x \in \Sigma^*$ viene introdotto $\hat{\delta}_D(q, x)$ lo **stato raggiunto** da M partendo da q quando si è alla sinistra di x, ed esecuendo successivamente il cammino x.

Si può definire per ricorsione:

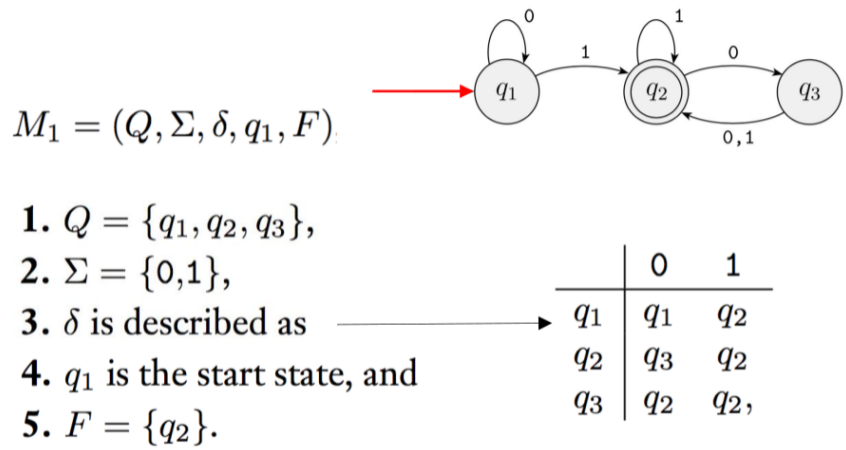


Figura 5.4: Automa a stati finiti che riconosce le stringhe A = {w | w contiene almeno un 1 ed un numero pari di 0 seguono l'ultimo 1}

$$\begin{cases} \hat{\delta}_D(q, \Lambda) = q \\ \hat{\delta}_D(q, xa) = \delta_D(\hat{\delta}_D(q, x), a) \end{cases}$$

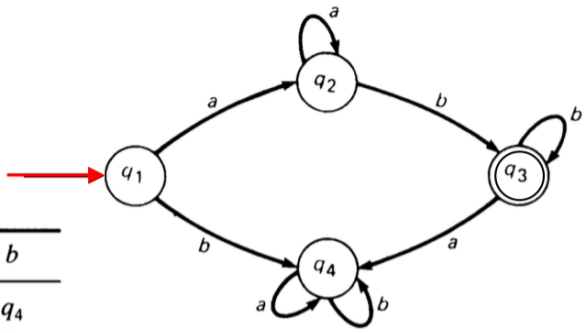
con $\delta_D(q, a) = \hat{\delta}_D(q, a)$. In altre parole, A_D accetta una parola x se $\hat{\delta}(q_0, x) \in \mathcal{F}$ insieme degli stati di accettazione, ovvero

$$\mathcal{L}(A_D) = \{x \in \Sigma^* | \hat{\delta}(q_0, x) \in \mathcal{F}\}.$$

Si dimostrerà che un linguaggio accettato da un automa a stati finiti è un linguaggio *regolare*: questo legame fra automi a stati finiti e linguaggi regolari consente di trattare gli automi mediante il formalismo dei linguaggio regolari.

5.3.1 Gli automi non deterministici

Un **automa non deterministico** (Non-deterministic Finite Automa) vede rimosso il vincolo che da ciascun nodo escano tutti e solid Σ caratteri diversi dell'alfabeto. Gli automi deterministici sono introdotti per semplificare enormemente il grafo cor-



δ	a	b
q_1	q_2	q_4
q_2	q_2	q_3
q_3	q_4	q_3
q_4	q_4	q_4

$$L(M_6) = \{a^n\, b^m \mid m,n > 0\}$$

Figura 5.5: Automa a stati finiti che riconosce qualsiasi stringa della forma $\mathcal{L}(M) = \{a^n b^m | m, n > 0\}$, ma **non** stringhe della forma $a^n b^n$ con $n > 0$.

rispondente — solitamente, automi deterministici producono grafi enormemente più complessi e meno sintetici rispetto agli automi non deterministici.

TODO esempio NFA 29 32

Il fatto di introdurre il non determinismo non introduce una maggiore potenza computazionale, bensì funge esclusivamente da semplificazione notevole per la rappresentazione a grafi.

5.3.2 Equivalenza fra automi NFA ed automi DFA

TODO 35

5.3.3 Automa non deterministico

Un **automa a stati finiti non deterministico** è una 5-tupla $Q, \Sigma, \delta, q_0, \mathcal{F})$ dove

1. Q è un insieme finito di *stati*;
2. Σ è un insieme finito, detto *alfabeto*;
3. $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ è la *funzione di transizione*;
4. q_0 è lo *stato di partenza*;
5. $\mathcal{F} \subset Q$ è l'*insieme degli stati di accettazione*.

La differenza principale rispetto agli automi deterministici è la definizione della funzione di transizione δ, dove il codominio è l'*insieme delle parti* di Q anziché l'insieme Q.

Similmente, la computazione per automi non deterministici dal punto di vista formale avviene con le seguenti differenze:

Sia $A_N = \{Q, \Sigma, \delta_D, q_0, \mathcal{F}\}$. Se $x \in \Sigma^*$ viene introdotto $\hat{\delta}_D(q, x)$ l'**insieme degli stati raggiunti** da A_N partendo da q quando si è alla sinistra di x, ed esecuendo successivamente *tutti i percorsi* x.

Di conseguenza, si può definire per ricorsione:

$$\begin{cases} \hat{\delta}_N(q, \Lambda) = \{q\} \\ \hat{\delta}_N(q, xa) = \bigcup_{p \in \hat{\delta}_N(q,x)} \hat{\delta}_N(p, a) \end{cases}$$

Si avrà infine che A_N accetta una parola x se $\hat{\delta}_N(q_0, x) \cap F \neq \emptyset$, cioè

$$L(A_N) = \{x \in \Sigma^* | \hat{\delta}_N(q_0, x) \cap F \neq \emptyset\}.$$

Poiché ogni NFA può essere descritto da un DFA in cui la $\delta_N(q, x)$ restituisce un solo stato, ogni linguaggio regolare è accettato da un NFA. Dunque, un linguaggio accettato da un automa non deterministico risulterà sempre essere *regolare*. La dimostrazione di ciò è nelle dispense.

5.4 Pumping lemma

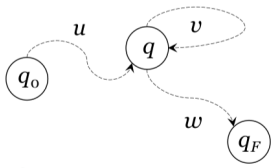
Un notevole risultato della teoria della computazione è noto col nome di **Pumping lemma**.

Teorema 4 *Pumping lemma Sia $\mathcal{L} = \mathcal{L}(\mathcal{A})$, dove $\mathcal{A}$ è un automa a m stati. Sia $x \in \mathcal{L}$, con $|x| \geq m$. Allora si può scrivere $x = uvw$, dove $v \neq \Lambda$ e $uv^{[i]}w \in \mathcal{L}$[1] per tutti i valori $i = 0, 1, 2, 3, \ldots$.*

DIMOSTRAZIONE— Poiché x consiste di almeno m simboli, $\mathcal{A}$ deve passare attraverso almeno m transizioni di stato mentre scansiona x; contando lo stato iniziale ciò richiede almeno $m + 1$ stati non necessariamente distinti. Ma poiché in tutto ci sono soltanto m stati, concludiamo (*principio della cassettiera* o principio della piccioniaia) che $\mathcal{A}$ deve essere in almeno uno stato più di una volta. Sia q uno stato in cui $\mathcal{A}$ si trova almeno due volte. Possiamo dunque scrivere $x = uvw$, dove

[1] La parte centrale della stringa è infinitamente replicabile, tuttavia il risultato continua ad appartenere al linguaggio $\mathcal{L}$.

$$\widehat{\delta}(q_0, u) = q$$
$$\widehat{\delta}(q, v) = q$$
$$\widehat{\delta}(q, w) \in F$$



$$\widehat{\delta}(q_0, uv^{[i]}w) = \widehat{\delta}(q_0, uvw) \in F$$

5.4.1 Utilizzo del Pumping lemma

Il pumping lemma è estremamente utile per dimostrare per assurdo la *non regolarità* di alcuni linguaggi. Ad esempio,

TODO 45

5.5 I grafi di transizione

Un **grafo di transizione** T **su** Σ è un *grafo orientato finito* in cui ogni arco è etichettato da qualche parola $w \in \Sigma^*$, eventualmente $w = \Lambda$ parola vuota. Esistono i vertici iniziali e vertici finali, i primi etichettati con il segno $-$, i secondi (se esistono) etichettati con il segno $+$. Un vertice può essere sia iniziale che finale.

In parole povere, gli archi non corrispondono ad un cambio di stato, bensì a delle stringhe, le quali comportano gli opportuni cambiamenti di stato all'interno del grafo. Le parole si possono concatenare, e la parola w viene accettata soltanto qualora la concatenazione delle etichette corrisponda proprio alla parola w.

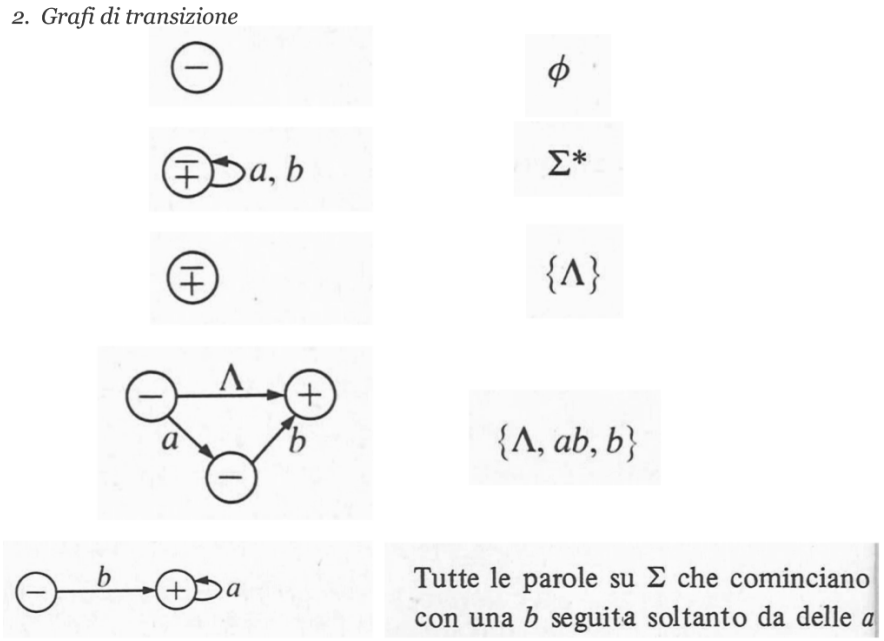


Figura 5.6: Alcuni esempi base di grafi di transizione. Altri esempi sono forniti sulle slide.

5.6 Le espressioni regolari

La **classe delle espressioni regolari**[2] su Σ è ricorsivamente definita come segue: sia $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, definiamo *alfabeto accessorio*

$$\Sigma_A = \{\sigma_1, \sigma_2, \ldots, \sigma_n, \Lambda, \emptyset, +, \cdot, *, (,)\}.$$

La classe delle **espressioni regolari** su Σ è definita come il sottoinsieme di Σ_A^* tale che

1. Λ e $\emptyset$ sono espressioni regolari su Σ;
2. ogni lettera $\sigma \in \Sigma$ è un'espressione regolare su Σ;
3. se R_1 e R_2 sono espressioni regolari su Σ, lo sono anche $R_1 + R_2$, $R_1 \cdot R_2$, ed R_1^*.

Dunque, ogni espressione regolare R su Σ descrive un insieme $\tilde{R}$ di parole su Σ, cioè si ha che $\tilde{R} \subseteq \Sigma^*$, definito ricorsivamente come segue:

[2] Tale definizione ricorda la definizione degli *insiemi regolari*. Infatti, l'operazione + è riconducibile all'unione fra insiemi, l'operazione $\cdot$ è riconducibile al prodotto fra insiemi, ed $*$ alla stellatura fra insiemi.

1. se $R = \Lambda$, allora $\tilde{R} = \{\Lambda\}$, cioè è l'insieme costituito dalla parola vuota Λ. Se $R = \emptyset$ allora $\tilde{R} = \emptyset$, cioè è uguale all'insieme vuoto;
2. se $R = \sigma$, allora $\tilde{R} = \{\sigma\}$, cioè è l'insieme costituito dalla lettera σ;
3. se R_1 ed R_2 sono espressioni regolari su Σ che descrivono gli insiemi di parole $\tilde{R}_1$ e $\tilde{R}_2$, rispettivamente,
   - se $R = (R_1 + R_2)$, allora

     $$\tilde{R} = \tilde{R}_1 \cup \tilde{R}_2 = \{x | x \in \tilde{R}_1 \text{ oppure } x \in \tilde{R}_2\}$$

     ovvero l'insieme unione;
   - se $R = (R_1 \cdot R_2)$, allora

     $$\tilde{R} = \tilde{R}_1\tilde{R}_2 = \{xy | x \in \tilde{R}_1 \text{ e } y \in \tilde{R}_2\}$$

     ovvero l'insieme prodotto;
   - se $R = (R_1)^*$, allora

     $$\tilde{R} = \tilde{R}_1^* = \{\Lambda\} \cup \{x | x \text{ ottenuti concatenando un numero finito di parole di } \tilde{R}_1\}$$

     ovvero l'insieme chiusura;

Esempi di espressioni regolari sono fornite nelle slide.

Un importante risultato è che dalla definizione delle espressioni regolari, consegue direttamente che *un insieme è regolare su Σ se e solo se può essere espresso da una espressione regolare su Σ*. Un insieme regolare può essere descritto da diverse espressioni regolari. Ad esempio, tutte le parole $\Sigma = \{a, b\}$ con a e b che si alternano e che cominciano e terminano con b.