Capitolo 3

Elementi di Complessità

ATTENZIONE: Questa parte, benché sia stata riletta, contiene imperfezioni e inesattezze che potrebbero fuorviare *totalmente* il lettore — inutile ripetere che queste note sono state scritte per uso personale da parte dell'insegnante. Ogni commento, correzione o segnalazione di errori è benvenuto.

3.1 Una teoria quantitativa degli algoritmi

Nella prima parte abbiamo studiato alcuni aspetti estensionali della teoria della calcolabilità. Infatti, abbiamo caratterizzato i problemi risolubili, rappresentati mediante funzioni calcolabili totali o insiemi ricorsivi, quelli insolubili o semi-decidibili, rappresentati da funzioni calcolabili o insiemi ricorsivamente enumerabili, e infine abbiamo discusso problemi nemmeno semi-decidibili, rappresentati da funzioni non calcolabili o insiemi non ricorsivamente enumerabili. Quindi l'enfasi è stata posta su *cosa* si calcola.

Passiamo ora a studiare *come* si calcola e *quali risorse* siano necessarie per calcolare, ovvero spostiamo l'accento sugli aspetti *intensionali* dei problemi, cioè introduciamo una *teoria quantitativa* dei problemi analizzando il comportamento degli *algoritmi* durante la loro esecuzione.

Considereremo nel seguito solo problemi *decidibili*, limitandoci a quelli di *decisione*, cioè guarderemo solo quei problemi, o insiemi, per cui esiste un algoritmo che ne calcola la funzione caratteristica. Per fare ciò, useremo alcune opportune varianti delle macchine di Turing, considerandole come automi accettori. In altre parole, le macchine usate finiranno in uno stato di acettazione se il dato in ingresso appartiene all'insieme, e quindi quel caso particolare del problema è risolto positivamente, altrimenti raggiungeranno uno stato di rifiuto. Un esempio di macchina di questo tipo è (un'immediata generalizzazione di) quella presentata nell'esempio 1.2.6 che risolve il problema delle stringhe palindrome.

Le risorse di cui si tiene principalmente conto quando si scrivono o si utilizzano programmi sono relative al *tempo* e allo *spazio*. Tuttavia queste non sono le uniche risorse critiche usate durante il calcolo; per esempio può essere importante valutare il dispendio energetico, che nel caso di macchine mobili può non coincidere esattamente con il tempo di calcolo, oppure il numero delle comunicazioni che avvengono tra i processori, in macchine parallele o distribuite. Nel seguito ci limiteremo a studiare, da un punto di vista astratto, solo quanto tempo e/o spazio sono necessari alla soluzione di un problema, ovvero alla soluzione di tutti i suoi casi; esamineremo anche in gran fretta pregi e manchevolezze di un tentativo di prescindere da tempo e spazio, in un quadro ancor piú astratto. In ogni caso, è necessario calcolare le risorse necessarie alla soluzione di un caso del problema in funzione dei suoi dati di ingresso, o meglio della loro *taglia*. La taglia del dato di ingresso x, che esprimeremo come $|x|$, verrà opportunamente misurata in base ad alcuni criteri, quali una stima dell'occupazione di memoria, il numero dei componenti del dato medesimo o altri ancora.

Ricapitolando: dato un problema I (sotto forma di un insieme) cercheremo una funzione $f(|x|)$ che esprima la quantità di risorse in spazio o tempo

necessaria al calcolo della soluzione di $x \in I$, chiamato il *caso* x di I. Quella che andremo a delineare risulta quindi una teoria della *complessità asintotica*, perché si studia come cresce l'uso delle risorse al crescere dei dati di ingresso.

La giustificazione concreta per lo sviluppo di una teoria della complessità astratta è, come già detto, quello di misurare l'efficienza degli algoritmi e, in ultima analisi, rispondere quando possibile alla domanda seguente: *qual è il modo piú efficiente di risolvere un problema?* Noi non risponderemo a questa domanda dando le tecniche e i metodi per progettare e realizzare algoritmi — argomento di altri corsi — anche se considereremo alcuni problemi paradigmatici e alcuni algoritmi per deciderli. Ci limiteremo piuttosto a capire, seppur superficialmente, le caratteristiche che tali problemi e quegli algoritmi hanno. Detto in maniera molto informale e intuitiva, cercheremo di individuare una parte della struttura *profonda* che accomuna molti problemi, o meglio quella del modo di risolverli, in un senso molto simile a quanto fatto nella prima parte del corso, quando abbiamo raggruppato i problemi decidibili tra loro e li abbiamo separati da quelli indecidibili. Ancora: cercheremo di evidenziare per quanto possibile la struttura matematica comune a gruppi di problemi, manifestata dal fatto che i loro algoritmi risolutori si "trasformano" gli uni negli altri, mediante opportune funzioni di riduzione.

Per far ciò è necessario prima studiare come si possono esprimere limiti alle risorse necessarie al calcolo: come si definisce e quali proprietà ha una funzione f (dai naturali ai naturali) che sia in grado di stimare la quantità minima di risorse necessarie per risolvere un certo problema I. Ovviamente, vogliamo far sí che un algoritmo M che risolve il caso $x \in I$ richieda una quantità di risorse spazio/temporali inferiore, o meglio ancora uguale a $f(|x|)$: f è quindi la minima delle funzioni che maggiorano la quantità di risorse necessarie al calcolo di $x \in I$. Si noti che se un algoritmo ha bisogno di una certa quantità di risorse, funzionerà benissimo con risorse maggiori e che invece se fornissimo a tale algoritmo una quantità di risorse inferiore a $f(|x|)$, allora potremmo non essere in grado di decidere con esso il caso $x \in I$. Trovata una tale funzione che stima la quantità di risorse necessarie, diremo piuttosto imprecisamente che il problema I ha *complessità* in tempo o spazio f. Si noti che f deve stimare le risorse necessarie alla soluzione di *tutti* i casi del problema, incluso il caso piú difficile che ne richiede in misura maggiore; stiamo dunque parlando di complessità nel caso *pessimo* — ciò implica anche che il dato di ingresso va *letto per intero*.

È importante osservare che determinare questa funzione f, se e quando ciò è possibile, richiede di considerare *tutti* gli algoritmi che risolvono il problema I, i quali sono tanti quanti i numeri naturali (v. il teorema 1.9.1). Infatti, per quanto appena detto, per stabilire che I ha complessità $f(n)$ *bisogna costruire un* programma che risolve il caso $x \in I$ in tempo o spazio inferiore o uguale a

$f(|x|)$. Ecco allora che, data una certa funzione f, si può definire una *classe di complessità*, dipendente da f, cui appartengono tutti quei problemi per cui esiste almeno un algoritmo che li risolve e che richiede risorse in misura inferiore o uguale a $f(n)$.

Allora viene naturale chiedersi: se il limite $f(n)$ alle risorse disponibili viene aumentato fino a un certo $g(n)$, cioè $g(n) > f(n)$, la classe di problemi che si possono risolvere diviene piú ampia? Ad esempio, aumentando le risorse di memoria di una certa macchina da $f(n) = 2n$ a $g(n) = n^2$, riusciamo a risolvere problemi che prima non potevamo risolvere? In altre parole, esistono delle *classi* i cui problemi sono *risolubili* con dei limiti alle risorse *prefissati*? ovvero esistono davvero *classi di complessità* diverse tra loro? e se sí, quali sono le relazioni che sussistono tra esse? Vedremo che in alcuni casi interessanti si possono stabilire delle gerarchie precise tra classi di problemi, e quindi classificare i problemi in base alla quantità di risorse necessarie per risolverli. Tuttavia, vedremo anche che vi sono dei casi in cui non è noto se vi sia una gerarchia in senso stretto, o addirittura che non è possibile stabilirne alcuna, a causa di funzioni di stima bizzarre.

Nel caso in cui si possano definire precisamente delle classi di complessità, queste hanno dei *problemi completi?* (Cioè problemi che appartengono a una classe e che sono i piú *ardui* da risolvere con quelle limitazioni spazio/temporali? cf. la definizione 1.10.12.)

E inoltre: *esistono classi particolarmente interessanti* dal punto di vista computazionale? e siccome una classe non è interessante senza almeno un problema completo che sia noto prima dell'introduzione della classe medesima, la stessa domanda implica anche: *vi sono problemi completi particolarmente interessanti?* Avete certamente già incontrato nei corsi precedenti le seguenti due classi di complessità:

- **Ptime**, o semplicemente $\mathcal{P}$, la classe dei problemi risolubili in tempo polinomiale *deterministico*;
- **NPtime**, o semplicemente $\mathcal{NP}$, la classe dei problemi risolubili in tempo polinomiale *non-deterministico*.

Queste due classi sono particolarmente rilevanti e spesso vengono associate rispettivamente alla classe dei problemi "trattabili" e a quella dei problemi "intrattabili". Questa associazione è talmente forte, sebbene non scevra da critiche che riporteremo nel seguito, da essersi meritata il nome di TESI DI COOK E KARP (E LEVIN), in analogia con la tesi di Church e Turing.[1]

[1]Non va dimenticato il contributo della scuola sovietica, in paricolare di Levin, Kolmogoroff, Markov, Trakhtenbrot, che per molti aspetti anticiparono molti dei risultati raggiunti indipendentemente in Nord-America, originati dal suggerimento di Shannon a considerare aspetti quantitativi degli algoritmi.

Prima di descrivere la teoria che abbiamo intuitivamente delineato, ci sono tre condizioni preliminari sulle quali ci soffermiamo.

Innanzitutto, bisogna mettersi d'accordo su *come si misura* lo spazio o il tempo necessario per risolvere un problema. Di solito, il costo di una computazione è la somma dei costi dei suoi passi elementari. Ad esempio, se il modello di calcolo sono le macchine di Turing, il tempo è misurato di solito dal numero dei passi necessari per arrivare alla configurazione terminale; se il modello è quello di un linguaggio funzionale, può essere il numero delle chiamate di funzione o delle riduzioni; se è un linguaggio di tipo WHILE può essere il numero dei confronti, o delle operazioni di somma o di moltiplicazione o il numero degli accessi in memoria, opportunamente pesati. Nel seguito non studieremo in dettaglio i vari modi per definire le funzioni di costo, ma ci limiteremo a osservare che bisogna aver molta cura nella loro scelta. Inoltre, le funzioni di costo cosí scelte devono consentire una buona formalizzazione di ciò che sembra accadere in realtà, quale ad esempio l'intuizione forte che al crescere delle risorse cresca anche la classe dei problemi risolvibili con esse. Infine, riporteremo anche un paio di fatti, intuitivamente paradossali, che si verificano quando si scelgano funzioni di costo in una certa misura arbitrarie, seppur apparentemente soddisfacenti.

Altra condizione preliminare è quella di garantire che la complessità di un problema non cambi al variare del modello di calcolo in cui la funzione che decide il problema è espressa. In altre parole, vogliamo essere sicuri che i risultati della teoria della complessità che andremo a enunciare sono *invarianti rispetto al modello di calcolo* scelto. Ci poniamo cioè l'obbiettivo di avere una teoria *robusta*. Fortunatamente, sotto ragionevolissime ipotesi, la scelta del modello di calcolo *non influenza* le classi di complessità che esso induce al variare delle risorse spazio/temporali disponibili.[2] Quindi potremo parlare di classi di complessità in generale, senza specificare da quale modello siano state indotte. Non studieremo approfonditamente il problema di garantire che tali classi siano effettivamente chiuse rispetto a "ragionevoli" trasformazioni di modelli, né caratterizzeremo in dettaglio quali trasformazioni siano ragionevoli. Mostreremo con un esempio che modifiche apparentemente innocue apportate a un modello possono mutare drasticamente le classi di complessità, se il modo in cui si definiscono le funzioni di costo non viene anch'esso opportunamente modificato. Infine, ci limiteremo a enunciare, senza dimostrarli, alcuni teoremi che assicurano che le classi di complessità, in particolare $\mathcal{P}$ e $\mathcal{NP}$, sono invarianti rispetto al modello di calcolo.

[2]Per far ciò si ricorre a riduzioni da un modello di calcolo all'altro, che sono "semplici" in un senso diverso da quello visto nella prima parte, ma altrettanto preciso; ne faremo solo un breve cenno.

La terza condizione per costruire una teoria quantitativa degli algoritmi accettabile, è che la complessità di un problema sia *invariante rispetto alla rappresentazione* del problema stesso. Ad esempio, la complessità di un problema sui grafi *non deve* dipendere dal fatto che questi siano rappresentati come matrici o come liste di adiacenza o ancora come coppie di insiemi (N, A), dove N sono i nodi e A sono gli archi. In altre parole, la richiesta di invarianza rispetto alla rappresentazione significa che le classi di complessità sono chiuse rispetto al cambio di rappresentazione dei dati. Abbiamo già incontrato la stessa richiesta quando abbiamo affermato che i risultati di calcolabilità *non* dipendono dal formato dei dati e allora abbiamo usato funzioni di codifica (calcolabili e) totali per passare da una rappresentazione a un'altra; qui chiederemo inoltre che le funzioni che trasformano i dati da un formato a un altro siano *facili*, in un senso che sarà piú chiaro in seguito. Naturalmente ci possono essere casi in cui la complessità cambia al cambiare della rappresentazione, ma sono casi degeneri, seppure molto interessanti da indagare perché ammettono algoritmi che li risolvono in modo particolarmente efficiente, sfruttando la loro struttura matematica. Ad esempio, si considerino i casi in cui si debbano moltiplicare matrici sparse o di altra forma speciale. Non si dimentichi che una notevole parte dell'ingegno dell'infernatico si esercita proprio nel trovare le giuste strutture dei dati!

Concludiamo questa breve introduzione ripetendo una domanda che ci siamo già posti: *le classi di complessità formano una gerarchia*? La letteratura riporta, sotto ragionevolissime ipotesi, un gran numero di risultati che stabiliscono le relazioni tra varie classi di complessità. Noi vedremo solamente un piccolissimo frammento di questa gerarchia, riassumibile nella seguente catena di inclusioni tra classi, la cui definizione posponiamo

$$\text{LOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subset R \subset RE$$

Oltre alle inclusioni proprie esplicitamente rappresentate nella parte destra della formula, è noto anche che LOGSPACE $\subset$ PSPACE. Non è ancora noto invece quali altre inclusioni siano proprie. In particolare, non è noto se $\mathcal{P} \subset \mathcal{NP}$ o se piuttosto $\mathcal{P} = \mathcal{NP}$ e nemmeno se ciò sia dimostrabile o meno: non si è tuttora riusciti a ottenere quel risultato di separazione tra le due classi che ci è riuscito nella prima parte considerando R ed RE – là il gioco di separarle riesce perché decidere la non appartenenza di un elemento a un insieme ricorsivo richiede un tempo *finito*, mentre questa limitazione di finitezza non può essere soddisfatta quando l'insieme è ricorsivamente enumerabile e non ricorsivo; qui il gioco non riesce affatto, non tanto perché sia i problemi in $\mathcal{P}$ che quelli in $\mathcal{NP}$ sono decidibili e quindi lo sono *per definizione* in tempo finito, ma soprattutto perché, e lo vedremo piú in dettaglio, tali insiemi sono per l'appunto decidibili in tempo *limitato*.

3.2 Misure di complessità deterministiche

In questo capitolo studieremo brevemente come associare a una macchina di Turing una funzione che stimi il tempo che le è necessario per risolvere un caso $x \in I$ del problema I che decide. Poi vedremo un paio di teoremi che mostrano come e di quanto questo tempo possa essere ridotto, al prezzo di usare macchine con un hardware piú "efficiente." Preliminarmente introdurremo una variante "parallela" delle macchine di Turing, permettendo alla macchina di operare simultaneamente su molti nastri. Questa estensione naturalmente non modifica la classe dei problemi decidibili, tuttavia ci consente in alcuni casi un trattamento piú agevole. Inoltre è un buon modello delle attuali macchine parallele sincrone (ma non di quelle concorrenti e distribuite e men che meno di quelle impiegate nel paradigma chiamato "mobile computing!"), nella stessa misura in cui le macchine di Turing a un nastro lo sono delle macchine sequenziali mono-processore. Vedremo infine quale sia il prezzo in termini di tempo che si deve pagare per simulare una macchina a molti nastri su una a un nastro solo, dando cosí un preciso limite teorico ai vantaggi ottenibili dall'introduzione di calcolatori paralleli.

Lo stesso schema verrà seguito per misure che riguardano lo spazio necessario al calcolo. Introdurremo un'ulteriore variante delle macchine di Turing in cui si identificano i nastri di lavoro (ignorando quelli destinati ai dati in ingresso e in uscita), in modo da definire lo spazio necessario per risolvere un problema. Anche in questo caso, la potenza espressiva non cambia. Infine, mostreremo che lo spazio può essere compresso in modo analogo a quanto fatto per ridurre il tempo.

L'aggettivo *deterministico* che compare nel titolo è legato al fatto che le macchine di Turing che impieghiamo usano una *funzione* di transizione, come quelle usate nella prima parte del corso. Ciò garantisce che ogni passo di computazione è univocamente determinato dallo stato e dal simbolo correnti, in altri termini, la macchina è *deterministica*. In seguito vedremo cosa succede impiegando *relazioni* di transizione piuttosto che funzioni, il che rende le macchine di Turing *non deterministiche*, in un senso che sarà precisato.

3.2.1 Macchine di Turing con k-nastri

Ricordiamo che per le macchine di Turing introdotte nella definizione 1.2.1 si postula l'esistenza di un nastro semi-infinito e di una funzione di transizione δ che opera su di esso. Adesso arricchiamo l'hardware delle macchine di Turing fornendole di k nastri; del resto questo era già stato fatto nella presentazione intuitiva della macchina di Turing universale. Poiché trattiamo solo problemi di decisione, ci prendiamo la libertà di spezzare lo stato di

arresto h in due nuovi stati di arresto SI, NO, a rappresentare che la macchina si ferma con successo nel primo caso e con insuccesso nel secondo (si noti che nell'esempio 1.2.6 avevamo usato SI come un *simbolo* e non come uno stato, mentre lo stato NO era chiamato q_N). Formalmente:

Definizione 3.2.1 (MdT a k nastri)**.** Dato un numero naturale k, *una macchina di Turing con k nastri* è una quadrupla $M = (Q, \Sigma, \delta, q_0)$, con

- $\#, \triangleright \in \Sigma$ e $L, R, - \notin \Sigma$
- $SI, NO \notin Q$
- $\delta : Q \times \Sigma^k \to Q \cup \{SI, NO\} \times (\Sigma \times \{L, R, -\})^k$ è la funzione di transizione, soggetta alle stesse condizioni della definizione 1.2.1 sull'unicità della stringa in $(\Sigma \times \{L, R, -\})^k$, in modo che δ sia una funzione, e sull'uso del carattere di inizio stringa $\triangleright$.

La funzione di transizione δ per uno stato q e k simboli $\sigma_1, \ldots, \sigma_k$ ha allora la forma seguente

$$\delta(q, \sigma_1, \ldots, \sigma_k) = (q', (\sigma'_1, D_1), (\sigma'_2, D_2), \ldots, (\sigma'_k, D_k)).$$

Una configurazione di una macchina a k nastri ha la forma

$$(q, u_1\sigma_1 v_1, u_2\sigma_2 v_2, \ldots, u_k\sigma_k v_k),$$

dove il carattere corrente sull'i-esimo nastro è σ_i, che abbiamo evitato di sottolineare per non appesantire la notazione; le sue computazioni lunghe n saranno rappresentate da

$$(q, w_1, w_2, \ldots, w_k) \to^n (q', w'_1, w'_2, \ldots, w'_k)$$

le cui mosse, derivabili in accordo con la funzione di transizione δ, hanno la forma

$$(q, u_1\sigma_1 v_1, u_2\sigma_2 v_2, \ldots, u_k\sigma_k v_k) \ \to \ (q', u'_1\sigma'_1 v'_1, u'_2\sigma'_2 v'_2, \ldots, u'_k\sigma'_k v'_k).$$

Si noti che se si volesse rappresentare una situazione in cui ci sono davvero k processori che evolvono in sincronia, ciascuno con il suo carattere corrente e con il suo stato preso da un insieme Q_i, basterebbe definire l'insieme della macchina a k nastri come $Q = Q_1 \times Q_2 \times ... \times Q_k$ e interpretare nel modo ovvio la funzione di transizione in modo da tenerne conto.

Vediamo adesso un esempio di macchina di Turing con due nastri.

Esempio 3.2.2. La seguente MdT con 2 nastri riconosce le stringhe palindrome costruite sull'alfabeto $\{a, b\}$. La funzione di transizione può essere suddivisa in tre "blocchi omogenei." Nel primo blocco ci sono le istruzioni

che ricopiano la stringa in ingresso sul secondo nastro; nel secondo la testina del primo nastro vien portata sul simbolo di inizio nastro, mentre quella del secondo nastro è lasciata sul primo carattere # dopo la stringa. Il terzo blocco di istruzioni effettua il controllo vero e proprio, cancellando dal secondo nastro a partire da *destra* i caratteri della stringa di ingresso se e solamente se corrispondono a quelli della stringa originale, incontrati muovendo il cursore del primo nastro da *sinistra*. Ovviamente, la stringa è palindroma se le due testine si trovano su caratteri sempre uguali e il secondo nastro viene svuotato.

q	σ_1	σ_2	$\delta(q,\sigma_1,\sigma_2)$		
q_0	$\rhd$	$\rhd$	q_0	$(\rhd, R)$	$(\rhd, R)$
q_0	a	$\#$	q_0	(a, R)	(a, R)
q_0	b	$\#$	q_0	(b, R)	(b, R)
q_0	$\#$	$\#$	q_1	$(\#, L)$	$(\#, -)$
q_1	a	$\#$	q_1	(a, L)	$(\#, -)$
q_1	b	$\#$	q_1	(b, L)	$(\#, -)$
q_1	$\rhd$	$\#$	q_2	$(\rhd, R)$	$(\#, L)$
q_2	a	a	q_2	(a, R)	$(\#, L)$
q_2	b	b	q_2	(b, R)	$(\#, L)$
q_2	a	b	NO	(a, R)	$(a, -)$
q_2	b	a	NO	(b, R)	$(b, -)$
q_2	$\#$	$\rhd$	SI	$(\#, -)$	$(\rhd, R)$

Come esempio di calcolo applichiamo la macchina alla stringa *abba*, raggruppando per quanto possibile i passi della computazione in blocchi omogenei.

$$(q_0, \trianglerighteq abba, \trianglerighteq \#) \to (q_0, \rhd \underline{a} bba, \rhd \underline{\#}) \to$$
$$(q_0, \rhd a \underline{b} ba, \rhd a \underline{\#}) \to^3 (q_0, \rhd abba \underline{\#}, \rhd abba \underline{\#}) \to$$

$$(q_1, \rhd ab \underline{b} a \#, \rhd abba \underline{\#}) \to (q_1, \rhd abb \underline{a} \#, \rhd abba \underline{\#}) \to^4 (q_1, \trianglerighteq abba, \rhd abba \underline{\#}) \to$$

$$(q_2, \rhd \underline{a} bba, \rhd abb \underline{a} \#) \to (q_2, \rhd a \underline{b} ba, \rhd ab \underline{b} \#) \to (q_2, \rhd ab \underline{b} a, \rhd a \underline{b} \#) \to$$
$$(q_2, \rhd abb \underline{a}, \rhd \underline{a} \#) \to (q_2, \rhd abba \underline{\#}, \trianglerighteq) \to (SI, \rhd abba \underline{\#}, \rhd \underline{\#}).$$

3.2.2 Complessità in tempo deterministico

Introduciamo adesso il modo che useremo per determinare il tempo necessario alla soluzione di un problema, ricordando che per problema qui intendiamo l'appartenenza o meno a un insieme, ovvero a un linguaggio di cui le macchine di Turing sono gli automi accettori; poiché le macchine usate sono *deterministiche*, anche le misure che introdurremo sono tali e spesso ometteremo tale aggettivo, dandolo per inteso.

Definizione 3.2.3. Diciamo che t è *il tempo deterministico richiesto* da una MdT M a k nastri [3] per *decidere* il caso $x \in I$ se e solo se

$$(q_0, \trianglerighteq x, \trianglerighteq, \ldots, \trianglerighteq) \rightarrow^t (H, w_1, w_2, \ldots, w_k), \text{ con } H \in \{SI, NO\}$$

In realtà vorremmo ottenere una misura del tempo necessario a risolvere un problema mediante la macchina M come una funzione della taglia dei suoi possibili dati di ingresso x; cioè, indicando la taglia di x con $|x|$, vorremmo una funzione $f(|x|)$. La definizione della taglia dei dati è arbitraria, ma spesso è molto naturale. Per esempio, spesso la taglia di un grafo è il numero dei suoi nodi (e/o dei suoi archi), quella di una stringa o di un vettore la sua lunghezza, indipendentemente dagli elementi costitutivi. La funzione *taglia* deve essere ovviamente calcolabile totale e *facile*; in ogni caso deve restituire un numero naturale. In queste note useremo funzioni di taglia spesso senza definirle e nel modo che ci sarà piú conveniente. Nella maggior parte dei casi e senza avvertenza contraria, misureremo i dati di ingresso x in relazione alle caselle della MdT necessarie a contenerli.

Nel seguito, non pretenderemo che la funzione $f(|x|)$ dia il numero *esatto* di passi necessari al calcolo di $M(x)$, perché ciò potrebbe rivelarsi troppo complicato; ci contenteremo allora di approssimare tale numero per eccesso: la macchina non richiederà un tempo maggiore di quello stimato. In conclusione, la funzione che determina la complessità di M è una funzione calcolabile totale $f : \mathbb{N} \rightarrow \mathbb{N}$, la quale limita superiormente il numero dei passi che M compie per risolvere il problema in questione — torneremo sulle caratteristiche delle funzioni di misura nella definizione 3.4.1. Si noti che questo non contrasta con la richiesta di avere classi di complessità, indotte da funzioni di misura, che esprimono la quantità *minima* di risorse necessarie alla decisione dei problemi in esse contenuti: basta trovare la minima funzione che limita superiormente i passi di M. L'aggettivo *deterministico* che compare nella definizione dipende dal fatto che le macchine di Turing usate sono deterministiche, nel senso che sarà piú chiaro dopo la definizione 3.3.1 (la componente δ è una funzione e non una relazione); quando sarà chiaro dal contesto che parliamo di questo tipo di macchine, ometteremo tale aggettivo.

Definizione 3.2.4. *M decide I in tempo deterministico f* se e solo se per ogni dato di ingresso x il tempo t richiesto da M per decidere x è minore di o è uguale a $f(|x|)$.

Adesso possiamo introdurre il concetto di *classe di complessità* in tempo *deterministico*.

[3] Se volessimo considerare il tipo di macchine viste in precedenza, cioè se $H = \{h\}$, allora si potrebbe definire che il *tempo richiesto* da M *su* x è t. Inoltre, questa definizione sarebbe accettabile anche per la complessità di problemi *tout court* e non solo di problemi decidibili come facciamo, ponendo ∞ il tempo richiesto, se $M(x) \uparrow$.

Definizione 3.2.5 (Classe di complessità in tempo deterministico)**.**

$$\mathrm{TIME}(f) = \{I \mid \exists M \text{ che decide } I \text{ in tempo deterministico } f\}$$

La classe di complessità appena introdotta contiene tutti e soli i problemi risolvibili in tempo deterministico f, ovvero affinché un problema vi appartenga occorre e basta che vi sia una macchina M che lo decide in tempo deterministico f.

Prendiamo ad esempio la macchina M dell'esempio 3.2.2 e calcoliamo la sua complessità in tempo, supponendo che la stringa di ingresso sia lunga n. Come abbiamo visto dianzi, il funzionamento della macchina può essere suddiviso in tre blocchi di operazioni "omogenee":

1.	copia il dato di ingresso sul secondo nastro in	$n+1$	passi
2.	rimette la prima testina sul $\triangleright$ in	$n+1$	passi
3.	sposta le due testine se i simboli sono uguali in	$n+1$	passi
		$3n+3$	passi

cui va aggiunto il passo per l'accettazione. Quindi il problema di verificare se una stringa è palindroma appartiene a $\mathrm{TIME}(3n+4)$ o, scordandosi le costanti, è dell'ordine di n.

Notazione. Nella discussione precedente abbiamo menzionato l'ordine di una funzione. Poiché questo concetto viene ripetutamente usato in seguito, in quanto in questa porzione di teoria della complessità si preferisce ignorare le costanti (ne discuteremo piú avanti le ragioni), vale la pena di introdurre esplicitamente la seguente abbreviazione, dove "quasi ovunque" significa per ogni argomento, eccetto che per un insieme finito di essi:

$$\mathcal{O}(f) = \{g \mid \exists r \in \mathbb{R}^+.\ g(n) < r \times f(n) \text{ quasi ovunque}\}$$

a indicare che la funzione f cresce allo stesso modo o piú velocemente delle funzioni g appartenenti alla classe $\mathcal{O}(f)$, la quale viene quindi chiamata *ordine* di f.[4] (Inutile notare che "piú velocemente" dipende solo dal fattore moltiplicativo r.)

In effetti, il calcolo di complessità fatto sopra non tiene conto del fatto che, se la stringa non è palindroma, il numero di passi necessari è minore. Abbiamo infatti definito una misura della complessità *nel caso pessimo*. Ci sono altri modi per misurare la complessità che tengono conto della distribuzione dei dati di ingresso. Un esempio particolarmente rilevante è quello della complessità *nel caso medio*, che tuttavia non tratteremo in queste note.

Può essere interessante confrontare adesso, dal punto di vista della complessità, le macchine che decidono se una stringa è palindroma nelle loro

[4] Di solito si introducono anche le classi
$\Omega(f) = \{g \mid f \in \mathcal{O}(g)\}$ — f cresce piú lentamente di g e
$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$ — f cresce come g.

versioni "parallela", appena vista, e "sequenziale", cioè quella con un nastro introdotta nell'esempio 1.2.6. Il tempo delle computazioni di quest'ultima è in $\mathcal{O}(n^2)$, perché servono n passi per controllare se il primo simbolo è uguale all'ultimo e questo controllo va ripetuto per $n/2$ volte. Una prima osservazione che possiamo fare è che il problema di decidere se una stringa è palindroma sta *anche* in $\mathcal{O}(n^2)$, il che non sorprende perché abbiamo appena visto che sta in $\mathcal{O}(n)$ e se una funzione è superiormente dominata da $g \in \mathcal{O}(n)$ lo è a maggior ragione da $h \in \mathcal{O}(n^2)$ — se avendo poco tempo a disposizione risolvo un problema, lo risolverò a maggior ragione se ne avessi di piú.

Un'altra osservazione, forse piú interessante, è che usando una macchina parallela abbiamo "guadagnato" tempo in modo quadratico (meglio: perduto da parallelo a sequenziale). Questo è un fatto vero in generale.

Teorema 3.2.6 (Riduzione del numero dei nastri)**.** *Data una macchina di Turing M con k nastri che decide I in tempo deterministico f, allora $\exists M'$ con 1 nastro che decide I in tempo deterministico $\mathcal{O}(f^2)$.*

Dimostrazione. Riportiamo solo una traccia della dimostrazione, confidando nella diligenza del lettori che vorranno certamente precisare i passi appena accennati. Costruiamo M' in modo che simuli la data M, in modo analogo a quanto fatto nella costruzione della macchina di Turing universale. Ogni configurazione di M della forma

$$(q, \triangleright w_1\sigma_1 u_1, \triangleright w_2\sigma_2 u_2, \ldots, \triangleright w_k\sigma_k u_k)$$

viene simulata da:

$$(q', \triangleright \triangleright' w_1\overline{\sigma}_1 u_1 \triangleleft' \triangleright' w_2\overline{\sigma}_2 u_2 \triangleleft' \ldots w_k\overline{\sigma}_k u_k \triangleleft'), \text{ per qualche } q'$$

cioè racchiudiamo ciascun nastro $w_i\sigma_i u_i$ tra due nuove parentesi $\triangleright'$ e $\triangleleft'$ e usiamo $\#\Sigma$ nuovi simboli $\overline{\sigma}_i$ per ricordarci di qual era la posizione della testina sull'i-esimo nastro.

Per cominciare, la macchina M' applicata a x dovrà generare la configurazione che simula la configurazione iniziale di M, cioè dobbiamo passare dalla configurazione iniziale di M $(q_0, \underline{\triangleright} x, \triangleright, ..., \triangleright)$ a

$$(q, \underline{\triangleright} \triangleright' x \triangleleft' (\triangleright' \triangleleft')^{\overline{k-1}}), \text{ per qualche } q.$$

Per far ciò, bastano $2k + \#\Sigma$ nuovi stati [5] e un certo numero di passi che, essendo dell'ordine di $|x|$, non influenza la complessità asintotica (consideriamo infatti il caso pessimo, quindi tutto il dato iniziale va letto).

[5] Supponendo che il carattere $\#$ non appaia in x, un modo per farlo è il seguente: arrivare al primo carattere $\#$ (il che richiede $|x| + 1$ passi e un nuovo stato); cambiare stato; tornare indietro di una casella e ricordarsi, codificandolo in un nuovo stato il simbolo ($\neq \#$) corrente, sia a, scriverci $\#$, spostarsi a destra e scrivere a; poi bisogna ripetere le ultime due mosse per $|x| - 1$ volte. In questo modo abbiamo "spostato" x di una casella a destra, impiegando $2 \times |x|$ passi e $2 + \#\Sigma$ nuovi stati. A questo punto si scrive sulla casella corrente, che è vuota la parentesi $\triangleright'$; si torna sulla prima casella vuota muovendosi a destra e si scrivono $k - 1$ coppie $\triangleright'\triangleleft'$, usando altri $2 \times (k - 1)$ nuovi stati.

Per simulare una mossa di M, la macchina M' scorre il dato di ingresso da sinistra a destra e *viceversa* due volte:

- la prima volta M' determina quali sono i simboli correnti di M, $\overline{\sigma}_i$ (si noti che, per ricordare quale sia la stringa $\overline{\sigma}_1 \dots \overline{\sigma}_k$ sono sufficienti $(\#\Sigma)^k$ nuovi stati).
- la seconda volta M' scrive i nuovi simboli nel posto giusto — attenzione! se un $\lhd'$ deve essere spostato a destra per far posto a un nuovo simbolo da scrivere, si verifica una cascata di spostamenti a destra!

Infine, quando M si ferma, anche M' si ferma, eventualmente rimuovendo tutte le parentesi $\rhd'$ e $\lhd'$ e sostituendo i caratteri $\overline{\sigma}_i$ con σ_i.

Adesso ricordiamo un fatto generale e ovvio: una macchina non può toccare un numero di caselle maggiore del numero dei passi che compie. Di conseguenza, la lunghezza totale del nastro scritto è al piú $P = k \times (f(|x|)+2)+1$ (l'addendo 2 è dovuto alle parentesi $\rhd'$ e $\lhd'$, l'addendo 1 al simbolo $\rhd$). Allora, andare due volte avanti e indietro costa, in termini di tempo, per ogni stringa simulata $4P$ (più al massimo $3P$ per gli spostamenti a destra, nel caso in cui la casella corrente sia all'estrema sinistra (è il caso pessimo nel quale ci poniamo sempre): P per arrivare alla fine del nastro scritto e $2P$ per spostare a destra i P caratteri, come descritto nella nota precedente). Poiché né k né le altre costanti sono rilevanti, possiamo concludere che per simulare un *singolo* passo di M la macchina M' richiede $\mathcal{O}(f(|x|)$ passi sul dato x. La stima del numero dei passi di M' sull'intera computazione è quindi in $\mathcal{O}(f(|x|)^2)$, perchè M richiede tempo $f(|x|)$ e perché M' impiega $\mathcal{O}(f(|x|)$ per simulare ogni passo di M. Infine, per costruzione M' è equivalente a M (ipse dixit!), e quindi le due macchine decidono lo stesso problema; allora M', che ha un nastro solo, decide tale problema in tempo deterministico $\mathcal{O}(f(|x|)^2)$. □

Il teorema appena dimostrato mostra che le MdT sono molto stabili! Infatti, miglioramenti che siano accettabili "algoritmicamente", come aggiungere nastri e processori che operano in parallelo, non solo non cambiano le funzioni calcolate, come ci aspettavamo, ma non modificano il tempo deterministico richiesto *se non polinomialmente*, quindi le MdT appaiono stabili anche rispetto la tesi di Cook-Karp (ancora da vedere con precisione!).

Ribadiamo adesso l'osservazione fatta nella dimostrazione di sopra, che mette in relazione il tempo e lo spazio necessari alla soluzione di un problema. Se una macchina di Turing M richiede tempo $f(|x|)$ per decidere $x \in I$, significa che si arresta in un numero di passi inferiore a $f(|x|)$, e quindi non può aver visitato, in alcuno dei suoi nastri, un numero di caselle maggiore di $f(|x|)$. Abbiamo quindi il seguente fatto basilare:

Osservazione 1: non si può usare piú spazio che tempo!

Usiamo ancora la dimostrazione appena riportata per dedurre un ulteriore osservazione. Infatti, il ragionamento fatto ci suggerisce come misurare, beninteso solo da un punto di vista teorico, i vantaggi che derivano dall'introduzione di macchine parallele. Negli ultimi passi della dimostrazione abbiamo potuto ottenere l'ordine $\mathcal{O}(f(|x|)^2)$ eliminando il fattore k^2, perché k è indipendente da x. Però l'elevamento al quadrato del fattore $f(|x|)$ non potrà mai essere eliminato! Quindi possiamo dedurre una stima del vantaggio che deriva dall'uso del parallelismo.

Corollario 3.2.7. *Le macchine parallele sono (al massimo) polinomialmente piú veloci di quelle sequenziali.*

Finora, nei nostri conti abbiamo impiegato solo gli ordini di crescita *trascurando* le costanti. Ovviamente, quando si cerchino stime piú precise, le costanti contano terribilmente, tanto che l'astuzia dei progettisti di algoritmi si dispiega spesso proprio nello scoprire come ridurle. Ciò non ostante, continueremo nel seguito a trascurare le costanti, a meno di casi particolari in cui esse saranno menzionate espressamente. Due sono le ragioni:

i) la teoria che si sviluppa è molto piú semplice; inoltre per valori grandi di n, cioè per dati di grandi dimensioni, le costanti tendono a valere "poco";

ii) macchine sempre piú potenti tendono a far rimpicciolire le costanti.

L'ultima osservazione è sostenuta da un teorema che riportiamo qui sotto, detto di accelerazione lineare. L'idea è che se $I \in \mathrm{TIME}(f)$, (ovvero se esiste una MdT M che lo risolve in tempo deterministico $f(n)$) allora I appartiene anche a $\mathrm{TIME}(\epsilon \times f)$, qualunque sia la scelta per $\epsilon > 0$ (attenzione: poiché la nostra complessità è nel caso pessimo, quanto detto è impreciso e ci sarà bisogno di una correzione per mantenere lineare la misura del tempo). In altre parole, dato un algoritmo che decide un problema, se ne può sempre trovare uno equivalente che è piú veloce per una costante moltiplicativa ϵ (supponendo ovviamente che questa sia minore di 1). Attenzione però: se p.e. $I \in \mathrm{TIME}(2^n)$, ovvero se il problema I è deciso da un algoritmo in tempo esponenziale, *non* è possibile trovare un algoritmo che lo risolva in tempo deterministico *polinomiale*, per mezzo del *solo* teorema di accelerazione. Un'analoga osservazione vale ovviamente per lo spazio, come vedremo. Quindi il teorema non inficia una eventuale gerarchia, ancora da stabilire.

Il trucco fondamentale è quello di codificare l'alfabeto Σ in un alfabeto "piú ricco" Σ^m, con m arbitrario. In pratica, questo significa avere macchine con parole di dimensioni via via crescenti $(32, 64, 128 \ldots 2^m$ bit). Si vede quindi che l'accelerazione è legata al *cambio di hardware*.

Questo non è chiaramente del tutto fattibile in pratica, e mostra che le costanti *sono* importanti quando la macchina sia fissata; inoltre, non si può aumentare a piacere l'efficienza di un tuo programma cambiando semplicemente l'hardware delle macchine!

Prima di enunciare il teorema di accelerazione lineare e di darne una traccia di dimostrazione, vediamo brevemente un esempio di come opera.

Esempio 3.2.8. Sia data una macchina che opera sui numeri naturali, codificati in notazione unaria, e che si arresti con successo solo quando il dato in ingresso sia un numero pari, ad esempio $|||| = 4$. Codifichiamo ogni $||$ con un nuovo simbolo $\diamond$ e $|\#$ con $\odot$. Se la macchina originale deve muoversi alternativamente tra due stati (pari/dispari corrispondenti a SI/NO), quella che la simula conta le barrette $|$ due alla volta, e quindi decide piú in fretta.

Nell'enunciato del teorema seguente compare un addendo $n + 2$ il quale dipende unicamente dal tipo di MdT usata (a 1 o a k nastri, con nastro semi-infinito o infinito, ecc.). La presenza dell'addendo n garantisce che, anche se la $f(n)$ fosse lineare, la complessità risultante rimarrebbe tale (nell'esempio di sopra abbiamo barato, supponendo che l'ingresso sia *già* nella forma che ci interessa, quindi non lo leggiamo tutto). In enunciati diversi del teorema si possono trovare diversi addendi, che variano in funzione dei vari tipi delle MdT; in tutti i casi essi garantiscono che il risultato sia una funzione almeno lineare. Si noti anche che in questa dimostrazione e in altre che seguiranno, si misura la taglia del dato con il numero di caselle del nastro di ingresso che servono a memorizzarlo.

Teorema 3.2.9 (Accelerazione lineare MdT)**.**
Se $I \in \mathrm{TIME}(f)$*, allora* $\forall \epsilon \in \mathbb{R}^+$ *si ha che* $I \in \mathrm{TIME}(\epsilon \times f(n) + n + 2)$.

Dimostrazione. Omessa, perché lunga e piena di dettagli insidiosi: si tratta di simulare una data macchina M con una macchina M', sulla falsariga di quanto fatto nella costruzione della macchina universale o nella dimostrazione 3.2.6. Può essere tuttavia interessante notare un fatto che potrebbe guidare il lettore a una maggiore comprensione del teorema stesso e dell'uso che si può fare degli stati per "ricordare" porzioni di nastro. Quello che faremo è vedere che si può determinare il numero m di simboli di M da compattare in un unico simbolo di M' in funzione del *solo* ϵ.
Il primo passo "condensa il dato di ingresso" (in $n + 2$ passi, con $n = |x| \leq m \times \lceil \frac{|x|}{m} \rceil + 2$): ogni sequenza di m simboli di M origina un singolo simbolo di M', cioè $\sigma_{i_1} \dots \sigma_{i_m}$ viene codificata come il *singolo* simbolo $[\sigma_{i_1} \cdots \sigma_{i_m}]$ (si noti che non c'è alcun problema se esiste $m' > 1$ tale che $\sigma_{k_h} = \#$ per $h \geq m' > 1$). In maniera analoga, gli stati di M' saranno formati da triple

$[q, \sigma_{i_1} \ldots \sigma_{i_m}, k]$, con $1 \leq k \leq m$, in modo da "rappresentare" il fatto che M si trova nello stato q e ha il cursore sul k-esimo simbolo della stringa $\sigma_{i_1} \ldots \sigma_{i_m}$. Data una configurazione, alla macchina M$'$ bastano 6 passi per simularne m della macchina M. Nei primi 4 passi M$'$ va a sinistra, poi a destra, poi ancora a destra e infine ritorna sul carattere corrente $s = \sigma_{i_1} \ldots \sigma_{i_m}$, in modo da raccogliere i simboli che M potrebbe vistare e codificarli nel suo stato. Infatti, M con m mosse può spostare il suo cursore all'interno della stringa di m caratteri che si trova a sinistra del carattere corrente, o di quella a destra o lasciarlo all'interno della stringa s considerata. Quando M compie m mosse, M' le simula "a blocchi" muovendosi a sinistra, oppure a destra del simbolo corrente s, ma in ogni caso ne modifica solo due, incluso s — le altre 2 mosse. Basta quindi "prevedere" il risultato di ciascun blocco di 6 transizioni di M', che dipende *solo* dalla funzione di transizione di M e non dal tipo di mosse fatte e men che meno dalla taglia del dato di ingresso. Allora M' farà $|x| + 2 + 6 \times \lceil \frac{f(|x|)}{m} \rceil$ passi e la traccia della dimostrazione si conclude scegliendo m in modo tale che $m = \lceil \frac{6}{\epsilon} \rceil$. □

Prima di introdurre una delle classi di complessità piú importanti, quella dei problemi decidibili in tempo polinomiale deterministico, usiamo i teoremi precedenti, per fare alcune osservazioni che giustificano ulteriormente la scelta fatta di usare solo ordini di grandezza trascurando le costanti.

Preliminarmente, notiamo che vi sono misure di complessità in tempo che sono sub-lineari, cioè vi sono macchine che richiedono un tempo $f(n) < n$ per risolvere un problema di taglia n. Per esempio, la ricerca di una parola in un dizionario effettuata con un metodo dicotomico porta a leggere $log\ n$ parole, certo non tutte quelle contenute nel dizionario stesso (ma tale misura è invariante rispetto al cambiamento di rappresentazione dei dati o si basa proprio su una caratteristica specifica della rappresentazione?). Non considereremo nel seguito misure in tempo sub-lineari, perché per ipotesi vogliamo ottenere la complessità nel caso pessimo, e quindi le macchine leggono sempre l'intero dato di ingresso x, il che richiede appunto $n = |x|$ passi e quindi ogni funzione di complessità in tempo f è tale che $f(n) \geqslant n$.

Adesso supponiamo di avere una funzione f. Se $f(n) = c \times n$ (cioè f è lineare), allora il teorema di accelerazione ci consente di "rimpicciolire" la costante fino a renderla uguale 1, ponendo $\epsilon = \frac{1}{c}$. Se invece $f(n) = c_1 n^k + c_2 n^{k-1} + \ldots + c_k$ (cioè è un polinomio), ancora una volta il teorema ci dice che possiamo rendere c_1 uguale 1 e inoltre gli addendi con esponente minori di k si possano trascurare perchè quello di grado massimo li domina per n sufficientemente grande: ecco allora giustificato l'uso di $\mathcal{O}(n^k)$. Infine, quanto detto sopra ci porta a concludere che, se I è decidibile polinomialmente, allora esiste un k tale che $I \in \mathcal{O}(n^k)$. Analoghe considerazioni si possono applicare

quando la funzione f considerata maggiori ogni polinomio, per esempio sia una funzione esponenziale.

Quanto osservato sopra basta per introdurre la classe dei problemi decidibili in tempo polinomiale deterministico, ovvero di quei problemi per cui esiste una macchina deterministica che li decide in tempo deterministico limitato da un polinomio.

Definizione 3.2.10. La classe dei problemi decidibili (da MdT) in tempo polinomiale deterministico è

$$\mathcal{P} = \bigcup_{k \geq 1} \mathrm{TIME}(n^k).$$

Il prossimo passo dovrebbe essere quello di dimostrare che la classe $\mathcal{P}$ appena introdotta è invariante rispetto al cambio di modelli. Dopo aver fatto questo, potremmo anche eliminare nella definizione precedente il riferimento alle macchine di Turing, ciò che implicitamente abbiamo già fatto. Tuttavia la mancanza di tempo ci impedisce di affrontare il problema della robustezza delle classi di complessità con la dovuta attenzione. Ci limiteremo allora a ritornare di sfuggita su questo punto piú avanti, affermando senza dimostrarlo che si può passare *in tempo polinomiale* da un algoritmo rappresentato in modello a uno equivalente rappresentato in un altro modello. In altre parole, la classe $\mathcal{P}$ è chiusa rispetto a trasformazioni di modelli, il che ne garantisce la robustezza.

Tuttavia, come accennato in precedenza, bisogna far molta attenzione al modo in cui si misura il tempo necessario a risolvere un problema. A titolo di esempio, riportiamo qui sotto il modello delle Random Access Machine, o RAM, che appare un po' piú vicino ai comuni elaboratori di quanto non lo siano le MdT, e vedremo a quali guai si va incontro scegliendo con leggerezza le funzioni di costo.

Esempio 3.2.11 (Random Access Machine)**.** Andiamo a definire il seguente modello, chiamato *Random Access Machine* (RAM) che è un'astrazione abbastanza fedele di un calcolatore alla von Neumann-Turing. Una RAM è formata da un certo numero di registri in cui si memorizzano valori, nel nostro caso numeri naturali, e da un insieme di semplici operazioni su di essi. Anche per questo modello vi sono moltissime varianti, che si distinguono l'una dall'altra sostanzialmente per le operazioni permesse. Qui consideriamo dapprima un insieme molto piccolo, che consiste, oltre alle operazioni di lettura e scrittura, del *successore*, del *predecessore* (limitato) e di un predicato di confronto con 0. Poi lo estenderemo con l'operazione di *prodotto* e vedremo che ciò rende necessario una ri-definizione delle funzioni di costo per

avere classi di complessità che riflettono l'intuizione e che corrispondono a quelle presentate di solito nella letteratura, ad esempio la classe $\mathcal{P}$ appena introdotta.

Sintassi
Supponiamo che vi siano un insieme di registri $Reg = \{x_i \mid 1 \leq i \leq n\}$ e un insieme di etichette $\ell, \ell', \ell'', \ldots \in \mathcal{L} \subset \mathbb{N}$, finiti, ma sufficientemente grandi. Allora, un *programma* è una successione finita di istruzioni

$$P \to I_1 I_2 \ldots I_n \quad n \geq 1$$

dove le istruzioni sono definite dalla seguente grammatica

$$\begin{array}{rcl} I & \to & x_i := x_j \mid x_i := x_i + 1 \mid x_i := x_i - 1 \mid x_i := 0 \mid \\ & & \texttt{if } x_i = 0 \texttt{ then } \ell' \texttt{ else } \ell'' \mid \textit{STOP} \end{array}$$

Postuliamo nel seguito che se l'istruzione $\texttt{if } x_i = 0 \texttt{ then } \ell' \texttt{ else } \ell''$ compare nel programma P con n istruzioni, allora $1 \leq \ell', \ell'' \leq n$.[6]

Semantica
Definiamo la semantica di un programma P specificando quello delle istruzioni che lo compongono, seguendo lo stile *small-step* (vedi nota su questo stile nel capitolo 1.3). Naturalmente, le operazioni + e − corrispondono alle operazioni semantiche di successore *succ* e predecessore limitato *pred*, mentre il significato di $@x_i$ è quella di indirizzamento indiretto attraverso il contenuto di x_i.

Come al solito supponiamo di modellare la memoria con una funzione $\sigma : \text{Reg} \to \mathbb{N}$ che associa ad ogni registro x_i il valore in esso contenuto (per brevità scriveremo $\sigma(i)$ al posto di $\sigma(x_i)$), dotata della solita operazione di aggiornamento $\sigma[m/i]$.

Una configurazione è una coppia (ℓ, σ), dove ℓ individua all'interno del programma P l'istruzione I che appare nell'ℓ-esima posizione.

La semantica di un'istruzione I rappresenta le modifiche a σ apportate da I, ha la forma $(\ell, \sigma) \to (\ell', \sigma')$ ed è definita per casi piú sotto.[7] La semantica di un programma [8] è quindi la chiusura transitiva della relazione di transizione $\to$; in simboli, dati un programma P e una memoria σ la

[6]Questo controllo può venir fatto facilmente a tempo statico, da un'opportuna routine di semantica statica.

[7]Si noti che la definizione della semantica $(\ell, \sigma) \to (\ell', \sigma')$ è essa stessa parziale: basta che nelle clausole che gestiscono i riferimenti indiretti i valori $x_{\sigma(j)}$ e $x_{\sigma(i)}$ non appartengano all'insieme dei registri *Reg*. Ovviamente la relazione di transizione è totale se ipotizziamo che $Reg = \{x_i \mid i \in \mathbb{N}\}$, come avviene solitamente al fine di dimostrare la Turing-equivalenza delle RAM.

[8]Anche qui come nel capitolo 1.3 non abbiamo istruzioni di ingresso/uscita

semantica di P è la funzione parziale definita come

$$[\![P]\!]_\sigma = \sigma' \quad \text{se } (1,\sigma) \to^* \sigma'.$$

Vediamo adesso la definizione di $(\ell,\sigma) \to (\ell',\sigma')$.

$$\begin{array}{ll}
(\ell,\sigma) \to (\ell+1, \sigma[\sigma(j)/i]) & \text{se } I_\ell = x_i := x_j \\
(\ell,\sigma) \to (\ell+1, \sigma[succ(\sigma(i))/i]) & \text{se } I_\ell = x_i := x_i + 1 \\
(\ell,\sigma) \to (\ell+1, \sigma[pred(\sigma(i))/i]) & \text{se } I_\ell = x_i := x_i - 1 \\
(\ell,\sigma) \to (\ell+1, \sigma[0/i]) & \text{se } I_\ell = x_i := 0 \\
(\ell,\sigma) \to (\ell',\sigma) & \text{se } I_\ell = \texttt{if } x_i = 0 \texttt{ then } \ell' \texttt{ else } \ell'' \\
 & \quad \text{e} \quad \sigma(i) = 0 \\
(\ell,\sigma) \to (\ell'',\sigma) & \text{se } I_\ell = \texttt{if } x_i = 0 \texttt{ then } \ell' \texttt{ else } \ell'' \\
 & \quad \text{e} \quad \sigma(i) = succ(n) \\
(\ell,\sigma) \to \sigma & \text{se } I_\ell = STOP
\end{array}$$

Possiamo assegnare un costo unitario a ciascuna delle istruzioni e quindi assegnare come costo dell'esecuzione di un programma il numero di passi che portano dalla configurazione iniziale $(1,\sigma)$ a quella finale σ'. Questo modello, con queste funzioni di costo, definisce le stesse classi di complessità che ci si aspettano e che sono definite da altri modelli di calcolo, in particolare la già citata $\mathcal{P}$.

Invece, se arricchiamo l'insieme delle istruzioni con la moltiplicazione, cioè con

$$x_i := x_j \times x_k$$

e gli associamo l'ovvia semantica

$$(\ell,\sigma) \to (\ell+1, \sigma[\sigma(j)\ per\ \sigma(k)/i]) \text{ se } I_\ell = x_j \times x_k$$

le cose cambiano drasticamente perchè i valori contenuti nei registri crescono rapidissimamente in breve tempo e quindi si calcolano funzioni che crescono altrettanto in fretta con "pochi" passi. Basta notare che con x_i moltiplicazioni si può calcolare $x_j^{x_i}$, ovvero il calcolo di un esponente viene fatto con un numero polinomiale di passi, il che contrasta con l'intuizione. Per evitare situazioni sgradite di questo genere, bisogna allora impiegare misure per il tempo piú fini, per esempio quelle "logaritmiche" che tengono conto della dimensione dei dati contenuti nei registri: quanto piú ingombranti saranno tali dati, tanto piú tempo ci vorrà per usarli. Si osservi infine che la presenza di speciali unità per la moltiplicazione nelle moderne architetture degli elaboratori non inficia il discorso appena fatto, anche se la moltiplicazione richiede tempo costante: bisogna infatti ricordare che tali unità operano su dati di dimensione prefissata, e non su numeri qualsiasi (cf. la condizione (v) dell'idea intuitiva di algoritmo).

3.2.3 Macchine di Turing I/O

Per studiare la complessità in spazio è conveniente usare un'ulteriore ragionevole e ben motivata variante di macchine di Turing, quelle che hanno un nastro dedicato a contenere il dato di ingresso, che sarà di sola lettura, uno destinato a memorizzare il risultato, che sarà di sola scrittura, e $k-2$ nastri di lavoro, gli unici rilevanti ai fini della complessità.

Definizione 3.2.12. Una MdT con k nastri $M = (Q, \Sigma, \delta, q_0)$ è di tipo I/O se e solamente se la funzione di transizione δ è tale che, tutte le volte che $\delta(q, \sigma_1, \ldots, \sigma_k) = (q', (\sigma'_1, D_1), \ldots, (\sigma'_k, D_k))$

- $\sigma'_1 = \sigma_1$ — quindi il primo nastro è a sola lettura;
- $D_k = R \vee (D_k = - \implies \sigma'_k = \sigma_k)$ — quindi il k-esimo nastro è a sola scrittura;
- $\sigma_1 = \# \implies D_1 \in \{L, -\}$ — la macchina visita al massimo una cella bianca a destra del dato di ingresso (che ipotizziamo non contenere # al suo interno, o che abbia una marca di fine stringa, v. dopo).

Si noti che nulla cambia nella definizione di funzione calcolata, né rispetto alle macchine di Turing usate nella prima parte, né rispetto quelle con k nastri. Inoltre, le relazioni delle macchine con k nastri di tipo I/O con quelle non di tipo I/O sono facili da stabilire.

Proprietà 3.2.13. *Per ogni MdT con k nastri M che decide I in tempo deterministico f esiste una MdT a $k+2$ nastri M' di tipo I/O che decide I in tempo deterministico $c \times f$, per qualche costante c.*

Dimostrazione. La macchina M' copia il primo nastro di M sul proprio secondo nastro, impiegando $|x| + 1$ passi; opera come M senza piú toccare il proprio primo nastro; e quando M si arresta, anche M' si arresta dopo aver copiato il risultato sul proprio $k+2$-esimo nastro, in al piú $f(|x|)$ passi. In totale, la macchina M' ha richiesto su x un numero di passi inferiore o uguale a $2 \times f(|x|) + |x| + 1$. Determinare la costante c è ora immediato. □

Infine, per stabilire l'equivalenza tra le MdT a k nastri di tipo I/O o le MdT usate nella prima parte, basta ricorrere alla simulazione vista nel teorema 3.2.6. Quindi, ancora una volta le macchine di Turing si dimostrano estremamente robuste: modifiche algoritmicamente "ragionevoli" non ne alterano il potere espressivo.

3.2.4 Complessità in spazio deterministico

Al fine di avere una nozione di misura di spazio sensata, modifichiamo la definizione di configurazione in modo da ricordare *tutte* le celle visitate, incluse quelle che erano o sono diventate bianche. A esser pignoli, questa modifica richiederebbe l'introduzione di un nuovo simbolo ausiliario, per esempio $\triangleleft \notin \Sigma$, da usarsi su ciascun nastro come delimitatore destro della parte scritta, e una semplice modifica alla funzione di transizione perché ne tenga conto e lo sposti, *ma solo a destra*, quando necessario. Per esempio, prendiamo la macchina "parallela" per decidere se una stringa è palindroma ed estendiamola con un nastro di ingresso e uno di uscita. Alcuni passi della computazione su *aba* verranno allora rappresentati cosí, dove q_i, q_j, q_k e q_h sono stati opportuni:

$(q_0, \trianglerighteq aba\triangleleft, \triangleright\triangleleft) \to^* (q_i, \triangleright aba\underline{\triangleleft}, \triangleright aba\underline{\triangleleft}) \to^*$
$(q_j, \trianglerighteq aba\triangleleft, \triangleright aba\underline{\triangleleft}) \to (q_k, \triangleright\underline{a}ba\triangleleft, \triangleright ab\underline{a}\triangleleft) \to (q_h, \triangleright a\underline{b}a\triangleleft, \triangleright a\underline{b}\#\triangleleft)$

Ci sentiamo liberi di non apportare queste modifiche e di immaginare che, una volta toccata, una casella del nastro apparirà sempre nella rappresentazione del nastro. Quindi in una configurazione $(q, u_1\sigma_1 v_1, \ldots, u_k\sigma_k v_k)$, u_i comincia per $\triangleright$ e v_i può finire con (molti) $\#$, tutti quelli su cui l'i-mo cursore è venuto a posizionarsi durante il calcolo. In questo modo il numero delle caselle in uso nei nastri non diminuisce mai, né

- nel nastro di ingresso, il primo, perché è di sola lettura;
- nel nastro di uscita, il k-esimo, perché è di sola scrittura;
- nei nastri di lavoro $1 < i < k$, perchè i caratteri bianchi a destra non scompaiono mai.

Adesso siamo pronti per definire lo spazio necessario a una computazione come il numero totale delle caselle toccate *solamente* sui nastri di lavoro. Come per il tempo, anche qui si parla di spazio deterministico perché usiamo macchine di Turing deterministiche.

Definizione 3.2.14. Sia M una MdT a k nastri di tipo I/O tale che $\forall x$

$$(q_0, \trianglerighteq x, \trianglerighteq, \ldots, \trianglerighteq) \to^* (H, w_1, w_2, \ldots, w_k) \text{ con } H \in \{\textsc{si}, \textsc{no}\}.$$

Lo *spazio richiesto* da M per decidere x è

$$\sum_{i=2}^{k-1} |w_i|.$$

Inoltre, *M decide I in spazio deterministico* $f(n)$ sse per ogni dato di ingresso x lo spazio richiesto da M per decidere x è minore o uguale a $f(|x|)$.
Infine, se M decide I in spazio deterministico $f(n)$, allora $I \in \text{SPACE}(f(n))$.

Ovviamente, la definizione appena vista può essere immediatamente adattata nel caso generale, in cui si consideri anche lo stato di arresto h come elemento dell'insieme H.

Alcuni autori preferiscono definire lo spazio richiesto come

$$\max_{2 \leq i \leq k-1} |w_i|$$

ma la sola differenza con la nostra definizione è un fattore k, il quale viene trascurato quando si considerano solamente ordini di grandezza.

La ragione principale per cui si trascura lo spazio necessario a contenere i dati di ingresso e quelli di uscita è che si vogliono misure abbastanza fini per la complessità in spazio. Infatti, se uno considerasse sempre anche la dimensione dei dati di ingresso, cioè la sommatoria di sopra partisse da 1 anziché da 2, si avrebbero sempre complessità almeno lineari. Ciò perché la misura del primo nastro è proprio $|x| + 1$, in quanto tale nastro è di sola lettura e contiene la stringa $w_1 = \triangleright x$. La dimensione del nastro d'uscita non è rilevante nel caso di problemi di decisione I considerati: il risultato è semplicemente un segnale che il caso $x \in I$ è risolto positivamente o meno. Inoltre, a differenza di quanto accade per le classi di complessità in tempo, ci sono delle classi interessanti e importanti che sono sub-lineari e che rivestono un ruolo rilevante nella trattazione successiva, per esempio, LOGSPACE(n), la classe dei problemi decisi in spazio deterministico logaritmico che definiamo piú precisamente qui sotto (la quale potrebbe coincidere con $\mathcal{P}$: si veda il frammento di gerarchia introdotto a pagina 132). In questo caso, sommare anche lo spazio per il dato di ingresso porterebbe a trascurare l'addendo $\log n$ che cresce meno di n e a schiacciare cosí LOGSPACE(n) su PSPACE(n).

Passiamo ora a vedere che anche lo spazio è suscettibile di essere ridotto linearmente, come già visto per il tempo. Si noti che un teorema analogo a quello di riduzione dei nastri sarebbe banale: avremmo ancora la stessa misura, dopo aver ricopiato fianco a fianco i k nastri di lavoro su un solo nastro.

Teorema 3.2.15 (Compressione lineare dello spazio)**.**
Se $I \in \text{SPACE}(f(n))$*, allora* $\forall \epsilon \in \mathbb{R}^+.\ I \in \text{SPACE}(2 + \epsilon \times f(n))$.

Come fatto precedentemente per $\mathcal{P}$, i problemi decidibili in tempo polinomiale deterministico, introduciamo ora la classe dei problemi decidibili in spazio deterministico polinomiale; poi, come promesso, definiremo quella dei

problemi decidibili in spazio deterministico logaritmico, che giocheranno un ruolo importante nel capitolo 3.5.

Definizione 3.2.16. La classe dei problemi decidibili (da MdT) in spazio polinomiale deterministico è

$$\text{PSPACE} = \bigcup_{k \geq 1} \text{SPACE}(n^k)$$

La classe dei problemi decidibili (da MdT) in spazio logaritmico deterministico è

$$\text{LOGSPACE} = \bigcup_{k \geq 1} \text{SPACE}(k \times \log n).$$

Per fortuna anche queste classi sono invarianti rispetto al cambio di modelli, e quindi come già fatto per $\mathcal{P}$ possiamo eliminare nella definizione precedente il riferimento alle macchine di Turing. In altre parole, le classi PSPACE e LOGSPACE sono chiuse rispetto a trasformazioni di modelli, il che garantisce la loro robustezza, oltre a quella della teoria che stiamo passando in rassegna.

Concludiamo questo capitolo stabilendo alcune relazioni tra le classi di complessità appena introdotte e facendo un'ulteriore osservazione su come tempo e spazio siano correlati. Iniziamo enunciando senza dimostrazione il teorema che stabilisce la relazione di stretta contenenza tra le due classi in spazio appena viste:

Teorema 3.2.17. LOGSPACE $\subsetneq$ PSPACE.

Inoltre, confrontiamo LOGSPACE con $\mathcal{P}$, stabilendo un altro piccolissimo frammento della gerarchia che intercorre tra classi di complessità in spazio e in tempo; ulteriori risultati si trovano nel capitolo 3.4.

Teorema 3.2.18. LOGSPACE $\subseteq \mathcal{P}$

Dimostrazione. Poiché il problema I appartiene a LOGSPACE, c'è una macchina di Turing M che decide ogni sua istanza $x \in I$ in $\mathcal{O}(\log |x|)$ spazio deterministico, basta notare che M può attraversare al massimo $\mathcal{O}(|x| \times \log |x| \times \#Q \times \#\Sigma^{\log |x|})$ configurazioni non terminali diverse. Una computazione non può ripassare su una stessa configurazione, altrimenti va in ciclo, quindi una computazione ha al massimo $\mathcal{O}(|x|^k)$ passi per qualche k. □

Infine, dalla dimostrazione di sopra, si può vedere che, seppure in modo meno preciso, vale anche il "duale" di quanto affermato a pagina 139, e cioè che, nel caso di algoritmi per problemi decidibili

Osservazione 2: lo spazio limita il tempo di calcolo!

3.3 Misure di complessità non deterministiche

C'è un modo per risolvere i problemi che è un po' ottuso, ma funziona sempre benissimo, complessità a parte, soprattutto quando uno non conosca o non sappia definire l'algoritmo giusto. Prendiamo come semplicissimo esempio il problema di calcolare il massimo numero naturale che sia minore della radice quadrata di 29 senza sapere né come fare né avere a disposizione la tabellina pitagorica: basterà allora cominciare a moltiplicare tra loro tutti i numeri tra 1 e 6 per scoprire che il risultato è 5 (se lo volete vedere come un problema di decisione, chiedetevi se $n \in \{\lfloor\sqrt{29}\rfloor\}$). In alternativa, si può tirare un dado, calcolare il quadrato del numero uscito e controllare se è una soluzione, magari ripetendo il lancio (avendo per magia rimosso il numero appena uscito dalle facce del dado) e confrontando l'ultimo numero uscito con i precedenti e con 29 — basta quindi che esista un lancio "fortunato" e la soluzione è trovata. Quest'ultima modalità (e anche l'altra!) origina un albero di scelte, del tipo di quello rappresentato nella figura 3.1, i cui livelli rappresentano i lanci e nei cui nodi immaginiamo di aver scritto il numero uscito — *basta che esista* un cammino nell'albero che porta a una soluzione.

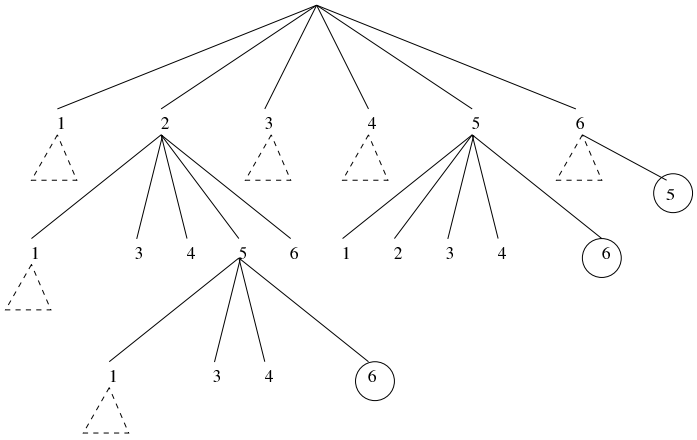

Figura 3.1: *Un frammento di albero di scelte non deterministiche; i nodi cerchiati indicano (alcune) situazioni di successo: 5 è minore di* $\sqrt{29}$*, 6 no.*

Questo metodo di soluzione, seppure ottuso, non è cosí assurdo come può apparire a prima vista e ritorneremo piú avanti sui problemi che sembra si possano risolvere solo ricorrendovi, perché non si conoscono o non si sanno sfruttare proprietà matematiche, strutturali del problema. In entrambi i casi delineati nell'esempio, vi è un procedimento di tipo *non deterministico* — l'unica proprietà matematica usata per giustificare l'impiego di un dado a sei facce è che $\sqrt{29} \leq 6$. Nel primo metodo, il non determinismo è meno evidente, essendo stato ridotto a una soluzione operazionalmente accettabile;

infatti, si generano tutte le scelte, ovvero l'intero spazio di ricerca, e ciascuna di queste viene esaminata, ovvero si fa una ricerca esaustiva della soluzione; questo metodo viene anche detto di *forza bruta* (in inglese *brute force* e in russo перебор = *perebor* ricerca disordinata). Con il secondo procedimento si prende invece una potenziale soluzione *a caso* (si ricordi che basta che ne esista una!) e si controlla se essa lo è davvero; in inglese questo metodo si chiama *guess-and-try* (botta di fortuna?!) — se c'è una soluzione, ci vien fornita per magia. Si noti che questo metodo *non* richiede né di generare né di visitare l'intero spazio di ricerca se non *implicitamente*, come sarà forse piú chiaro in seguito. Inoltre, l'albero delle possibili scelte è sempre finito, perché stiamo considerando solo problemi decidibili. Quindi, una soluzione appare *sempre* a profondità finita o, sempre a profondità finita, *tutti* i rami dell'albero di scelta finiscono su nodi che riportano fallimento (non è il caso nel nostro esempio, in cui tutti i rami terminano con successo al massimo dopo sei lanci del dado, cioè a profondità sei nell'albero; non è difficile immaginare casi in cui vi siano situazioni di successo e situazioni di fallimento e ne vedremo in seguito). Si noti infine che un albero delle possibili scelte, per brevità *albero non deterministico delle computazioni* o semplicemente *albero non deterministico* (v. la figura 2.6 che organizza in un albero le computazioni di un ASFD), può sempre essere visitato in modo deterministico, livello per livello.

Per formalizzare le intuizioni descritte sopra, è opportuno introdurre una variante delle macchine di Turing, dette non deterministiche. Essenzialmente, una macchina non deterministica differisce da una deterministica per il fatto che la relazione di transizione δ non è necessariamente una funzione, cioè una configurazione può evolvere in piú di una configurazione successiva, originando per cosí dire un albero (di computazioni) non deterministico (sia ben chiaro però che una computazione continua a essere una *singola successione* di configurazioni). L'osservazione fatta prima, che tale albero può esser visitato per livelli, ci assicura che introdurre le macchine non deterministiche non cambia affatto la classe dei problemi decidibili, né alcuno dei risultati di calcolabilità presentati nella prima parte del corso.

Con la stessa osservazione ci si può facilmente convincere che le macchine di Turing deterministiche simulano quelle non deterministiche con una perdita di efficienza *esponenziale* (si ricordi che i nodi di un albero sono in numero esponenziale rispetto la profondità dell'albero stesso); per una formulazione esatta, si veda il teorema 3.3.6. Quanto detto giustifica, almeno in parte, la tesi di Cook-Karp, cioè che la classe dei problemi decidibili in tempo polinomiale *deterministico*, $\mathcal{P}$, è formata dai problemi *facili*, mentre la classe dei problemi decidibili in tempo polinomiale *non deterministico*, $\mathcal{NP}$, che definiremo precisamente in seguito, è quella dei problemi *difficili*. Torneremo

piú avanti su questa distinzione, analizzando piú in dettaglio $\mathcal{P}$ e $\mathcal{NP}$.

Però a questo punto non possiamo non chiederci se l'introduzione del non-determinismo dia davvero un potere maggiore alle MdT dal punto di vista della complessità del calcolo, ovvero quanto sia fondata la tesi di Cook-Karp. Abbiamo già toccato questo argomento introducendo un frammento di una gerarchia di complessità, concludendo col dire che è ancora irrisolto il famoso problema $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, ovvero se non vi sia differenza tra il tempo polinomiale non deterministico e quello deterministico, nel qual caso aver introdotto il non determinismo non separerebbe i problemi "difficili" da quelli "facili." Per quanto riguarda lo spazio, preannunciamo che la classe dei problemi che richiedono spazio polinomiale *deterministico* PSPACE coincide con NPSPACE, quella dei problemi risolubili in spazio polinomiale *non deterministico*, ancora da definire. Se valesse anche l'eguaglianza $\mathcal{P} = \mathcal{NP}$, il meccanismo del non determinismo, che sembra avere scarso significato computazionale, almeno dal punto di vista della sua concreta realizzabilità, sarebbe completamente irrilevante anche dal punto di vista della complessità.

3.3.1 Macchine di Turing non deterministiche

Introduciamo di seguito l'estensione non deterministica alle macchine di Turing come definite in 1.2.1, dando esplicitamente conto solo delle cose che cambiano. Come detto sopra, una macchina non deterministica differisce da una deterministica per il fatto che la relazione di transizione δ non è necessariamente una funzione. A differenza di altre estensioni viste, questo nuovo modello non appare però sufficientemente "realistico." Infatti, le estensioni viste nel capitolo precedente si basano su meccanismi che hanno un'interpretazione computazionale immediata; ad esempio, le macchine con molti nastri sono una formalizzazione accettabile dei calcolatori paralleli. Invece, non si conoscono, almeno per ora, né ci riesce di immaginare, allo stato presente della tecnologia, macchine che siano davvero non deterministiche.[9]

[9]Ricevo da Clelia De Felice la seguente precisazione: "Tra le differenti varianti della macchina di Turing, introdotte allo scopo di confermare o confutare la tesi di Church-Turing, vanno citati alcuni modelli di computazione introdotti abbastanza recentemente e ispirati a meccanismi fisici o biologici.

Piú precisamente, in alcuni articoli di Deutsch, Feynman ed altri autori, apparsi verso la metà degli anni ottanta, viene esaminata criticamente l'adeguatezza del sistema fisico (computer o agente umano) alla base del modello di macchina di Turing. Senza entrare nei dettagli, appare chiaro che tale sistema fisico obbedisce alle leggi della meccanica classica laddove la comunità fisica è abbastanza concorde che il comportamento di un sistema fisico vada descritto sulla base delle leggi della fisica quantistica [magari facendo riferimento alla teoria delle stringhe]. Una macchina che modella un tale tipo di sistema (macchina di

L'importanza di queste macchine però, sia come le presentiamo qui, e a maggior ragione in versioni assai piú elaborate studiate in letteratura, è rilevantissima per organizzare in modo adeguato una teoria quantitativa degli algoritmi, per cui, pur consci della loro astrattezza e apparente (?) irrealizzabilità, non esitiamo a usarle.

Definizione 3.3.1. Una MdT *non deterministica* (a k nastri, di tipo I/O) è una quadrupla $N = (Q, \Sigma, \Delta, q_0)$, dove

- Q, Σ, q_0 sono come nella definizione 1.2.1 (nella 3.2.1 delle macchine con k nastri, nella 3.2.12 di quelle I/O);
- $\Delta \subseteq (Q \times \Sigma) \times ((Q \cup \{SI, NO\}) \times \Sigma \times \{L, R, -\})$ è la *relazione* di transizione (estesa nel modo ovvio nel caso delle macchine a k nastri e di tipo I/O).

Le configurazioni non vengono affatto modificate: esse hanno la stessa forma di quelle già viste: $(q, w\sigma u)$. Allo stesso modo non cambia il passo di computazione $(q, w\sigma u) \to_N (q', w'\sigma' u')$, e quindi le computazioni sono anche qui una *successione* (*non* un albero!) di configurazioni; infine continueremo a usare gli apici n e $*$ per indicare computazioni di lunghezza n o qualunque.

A rimarcare che nelle macchine di Turing non deterministiche si usa una relazione piuttosto che una funzione, abbiamo usato la lettera maiuscola Δ al posto di quella minuscola δ. Poiché la relazione di transizione Δ può contenere piú quintuple associate allo stesso stato e allo stesso simbolo, ci possono essere molte configurazioni $(q', w'\sigma' u')$ che sono raggiungibili da $(q, w\sigma u)$ in un *solo* passo (e ovviamente una sola lo sarà). Dovrebbe essere adesso piú chiaro che le computazioni possono venir organizzate in un albero non

Turing quantistica) è stato formalizzato in un articolo di Deutsch.

Meccanismi computazionali basati su fenomeni biologici hanno invece conosciuto un particolare sviluppo soprattutto dopo un esperimento effettuato da Adleman e che consiste essenzialmente nell'utilizzare reazioni biologiche su molecole di DNA per "codificare" un algoritmo per la soluzione del problema del cammino hamiltoniano in un grafo.

C'è da dire che né tali meccanismi, né la macchina di Turing quantistica ampliano le capacità espressive delle macchine di Turing: la classe delle funzioni calcolate è quella delle funzioni T- [o μ- o WHILE-] calcolabili. Dato però il parallelismo estremo di tali dispositivi, quello che esse alterano è piuttosto la classificazione di alcuni problemi nelle varie classi di complessità; ad esempio, l'algoritmo quantistico di Shor che fattorizza un intero in numeri primi con complessità di tempo polinomiale" — piú precisamente $\mathcal{O}(n^3)$ mentre non se ne conoscono di non-quantistici con complessità polinomiale.

Si tenga però presente che Lov Grover ha proposto un algoritmo quantistico ottimo per la ricerca in un insieme di n elementi che richiede $\mathcal{O}(\sqrt{n})$ passi, e siccome (quasi) sempre i problemi in $\mathcal{NP}$ (completi o meno) sono risolti con una ricerca esaustiva (v. oltre), usando tale algoritmo l'accelerazione massima ottenibile è quadratica (a meno che $\mathcal{P} = \mathcal{NP}$).

deterministico del genere visto sopra. Inoltre, la vera potenza del non determinismo appare nel modo in cui si accetta (o se preferite si calcola, il che richiede una semplice e ovvia estensione alla definizione seguente).

Definizione 3.3.2. La macchina non deterministica N (a k nastri) decide I tutte e sole le volte che

$x \in I$ se e solamente se

esiste una computazione tale che $(q_0, \trianglerighteq x, \triangleright, ..., \triangleright) \rightarrow_N^* (SI, w_1, w_2, ..., w_k)$

Per accettare una stringa di ingresso, *basta che esista* una computazione che porta a una configurazione il cui stato è SI – ecco il pizzico di magia: basta che ci sia una configurazione che accetta e non è affatto rilevante che ci siano altre computazioni che rifiutano, finendo in configurazioni con stato NO (o nel caso generale che non terminano).[10] Si noti come questa definizione di accettazione introduca un'asimmetria rispetto a quella di non accettazione. In quest'ultimo caso, infatti, per rifiutare una stringa di ingresso bisogna che *tutte* le computazioni della macchina portino a configurazioni con stato NO o siano non terminanti. La situazione di non terminazione verrà esclusa nel seguito, perchè parliamo solo di problemi decidibili; per avere un'idea di come ci possano essere computazioni non terminanti, si consideri di nuovo il banale esempio fatto all'inizio del capitolo, e si cerchi di determinare che 5 è una soluzione lanciando ripetutamente il dado da cui però nessun mago ha sottratto le facce che portano i numeri già usciti: vi saranno allora sequenze di lanci in cui esce sempre lo stesso numero. Si noti tuttavia che, anche senza magia, le soluzioni si trovano *sempre* a profondità finita.

Non tedieremo il lettore con un esempio di macchina non deterministica, sicuri che la sua capacità di astrazione è stata sufficientemente esercitata e sviluppata, e che l'albero non deterministico abbozzato in figura 3.2 sia sufficiente a illustrare come dalla configurazione iniziale si possa transire in una delle quattro disegnate, da quella piú a destra di esse in altre tre e cosí via. Ci limitiamo a notare che una computazione è *completamente determinata* da una sequenza di scelte tra le varie configurazioni raggiungibili passo dopo passo. Per esempio, immaginando che l'albero nella figura 3.1 sia quello delle computazioni di una macchina non deterministica, la soluzione rappresentata piú a destra è individuata dalla sequenza di scelte $(5, 4)$, quella rappresentata piú a sinistra dalla sequenza $(1, 3, 3)$, numerando gli archi uscenti da ciascun nodo a partire da sinistra, cominciando da 0.

[10] Per questa ragione il non determinismo che abbiamo introdotto si chiama anche *angelico*; la versione *demoniaca* prevede che tutte le computazioni raggiungano uno stato di successo.

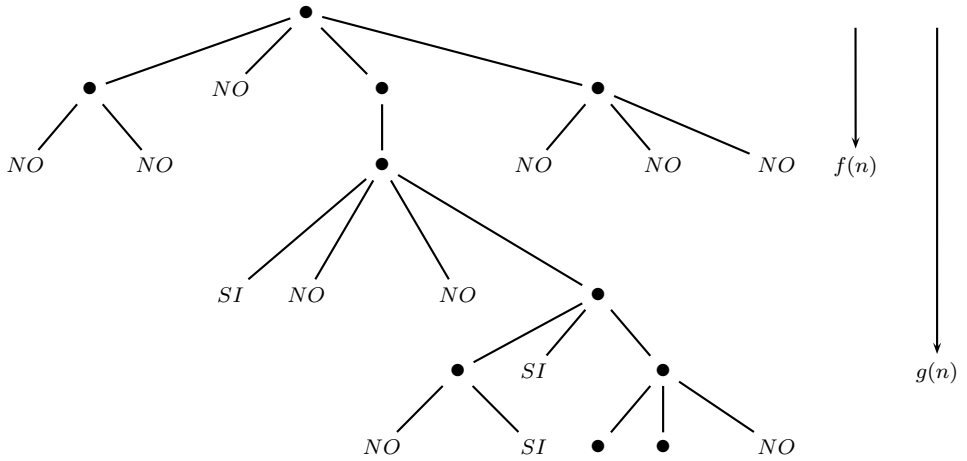

Figura 3.2: *Un albero delle computazioni di una MdT non deterministica; il simbolo • è da intendersi come un nodo intermedio di una computazione, anche non terminante; non vi sono computazioni che terminano con successo su un dato ingresso in due passi (illustrato da $f(n)$), mentre ve ne sono due in $g(n)$ passi (vedi Def. 3.3.3), determinate dalle sequenze di scelte* $(2,0,0)$ *e* $(2,0,3,1)$; altre ancora eccedono tale stima.

3.3.2 Complessità in tempo e in spazio non deterministici

Introduciamo ora i corrispettivi non deterministici delle classi di complessità definite nel precedente capitolo, i cui nomi inizieranno tutti con la lettera N, per non determinismo. Non si dimentichi che nel seguito considereremo *solo* problemi decidibili, quindi le macchine che useremo terminano per ogni ingresso; ciò implica che se una di queste macchine N decide I, non solo si ha che $\forall x \in I$ esistono $w_1, w_2, ..., w_k$ tali che $N(x) = (q_0, \trianglerighteq x, \triangleright, ..., \triangleright) \rightarrow_N^* (\textsc{si}, w_1, w_2, ..., w_k)$, ma anche che $\forall x \notin I$ per ogni $w_1, w_2, ..., w_k$ tali che $N(x) \rightarrow_N^* (q, w_1, w_2, ..., w_k) \not\rightarrow_N$ si ha che $q = \textsc{no}$, cioè si raggiunge lo stato di rifiuto.

Definizione 3.3.3. La macchina nondeterministica N *decide* I *in tempo non deterministico* $f(n)$ se e solamente se

- N decide I e
- $\forall x \in I \; \exists t$ tale che $(q_0, \trianglerighteq x, \triangleright, ..., \triangleright) \rightarrow_N^t (\textsc{si}, w_1, w_2, ..., w_k),\ t \leqslant f(|x|)$.

Intuitivamente, una MdT non deterministica N accetta x in tempo non deterministico $f(|x|)$ se e solo se c'è *almeno una* computazione che termina in uno stato $\textsc{si}$ in t passi, $t \leqslant f(|x|)$; di fatto, occorre e basta che la piú breve delle computazioni accettanti ci metta meno passi di $f(|x|)$, o altrettanti. Invece, N non accetta x se *tutte* le sue computazioni lunghe al massimo $f(|x|)$

conducono allo stato NO. Ecco di nuovo l'asimmetria: non basta che per un elemento $x \notin I$ ci sia una computazione che porta allo stato NO in meno di $f(|x|)+1$ passi, ma si richiede che lo debbano fare *tutte* in meno di $f(|x|)+1$ passi. Quindi l'asimmetria tra il decidere I e il decidere $\bar{I}$ che abbiamo visto nel caso generale viene rafforzata dal richiedere che ciò venga svolto in tempo $f(|x|)$. Ripetiamo: affinché N decida il problema I basta che per *ogni* suo elemento x ci sia *una* computazione che lo accetta. Come esempio si veda la figura 3.2 e la relativa didascalia. Invece, affinché N decida $\bar{I}$ bisogna che *tutte* le computazioni sul dato di ingresso $x \notin I$ conducano allo stato NO in meno di $f(|x|)+1$ passi.

Come fatto per TIME($f(n)$) possiamo ora definire la classe dei problemi decidibili da MdT (e da altri modelli "ragionevolmente" equivalenti) in tempo *non deterministico* $f(n)$.

Definizione 3.3.4. Se una macchina di Turing non deterministica decide il problema I in tempo f, allora $I \in \text{NTIME}(f)$.

Adesso possiamo finalmente introdurre la classe dei problemi decidibili in tempo polinomiale non deterministico.

Definizione 3.3.5. La classe dei problemi decidibili in tempo non deterministico polinomiale è

$$\mathcal{NP} = \bigcup_{k \geq 1} \text{NTIME}(n^k)$$

Ovviamente $\mathcal{P} \subseteq \mathcal{NP}$, perchè una macchina deterministica M è anche non deterministica. Sarebbe bello se fosse vero anche il contrario, cioè $\mathcal{P} \supseteq \mathcal{NP}$, da cui si dedurrebbe che $\mathcal{P} = \mathcal{NP}$. Se cosí fosse, ci sarebbe un modo per trasformare un algoritmo non deterministico polinomiale in uno deterministico polinomiale, quindi "facile e fattibile."

Come abbiamo già piú volte detto, quello che si sa fare per ora è di simulare una macchina non deterministica N che decide un problema in tempo polinomiale con una macchina deterministica M con una perdita di efficenza in tempo *esponenziale*. L'idea è di generare prima le potenziali soluzioni e poi verificare se lo sono davvero; vedremo anche che il primo passo è difficile, nel senso che (solitamente) richiede un numero di passi esponenziale, mentre il secondo è facile e può venir fatto in tempo deterministico polinomiale.

Teorema 3.3.6. *Se I è deciso in tempo non deterministico $f(n)$ dalla macchina non deterministica N (a k nastri), allora, con una perdita esponenziale, è deciso in tempo $\mathcal{O}(c^{f(n)})$ da una macchina deterministica M (a $k+1$ nastri), con $c > 1$ dipendente solo da N. Piú precisamente:*

$$\text{NTIME}(f(n)) \subseteq \text{TIME}(c^{f(n)}).$$

Dimostrazione. Sia d il grado di non-determinismo di N, cioè poniamo

$$d = \max\{Grado(q, \sigma) \mid q \in Q, \sigma \in \Sigma\}$$

dove $Grado(q, \sigma) = \#\{(q', \sigma', D) \mid ((q, \sigma), (q', \sigma', D)) \in \Delta\}$.
(Per semplicità di trattazione, supponiamo nel seguito che la macchina abbia sempre d scelte; non è difficile modificare quanto segue al caso generale, si veda la dimostrazione del Teorema 3.5.26 di Cook.)
Per ogni stato $q \in Q$ e ogni simbolo $\sigma \in \Sigma$, ordiniamo totalmente l'insieme $\Delta(q, \sigma)$, per esempio lessicograficamente. Ogni computazione di N è una sequenza di scelte lunga $t \geq 1$, ovvero è una successione di numeri naturali minori di t nell'intervallo $[0..d-1]$, rappresentando con 0 la prima scelta, come fatto negli esempi. La macchina M considera queste successioni una alla volta, in ordine di lunghezza crescente (ovvero visita l'albero per livelli) e per ciascuna di esse simula N. (N.B. La costruzione di M deve essere indipendente da f e quindi deve esser fatta *a prescindere* dal valore corrente di $f(|x|)$ — se cosí non fosse, basterebbe generare tutte le computazioni lunghe al piú $f(|x|)$.)
Qualche dettaglio in piú: la macchina M tiene su un nastro di lavoro la lunghezza t e riproduce la successione di scelte $(c_1, \ldots, c_t)$, tenendo l'ultima successione, ovvero un numero in base d minore di t su un ulteriore nastro aggiuntivo. Se durante questa simulazione M arriva in uno stato s_I allora M termina accettando, altrimenti genera la prossima serie di successioni, ponendo $t+1$ sul primo nastro di lavoro, incrementando cioè la lunghezza delle prossime computazioni da simulare. Se tutte le successioni lunghe t corrispondono a computazioni che terminano portando allo stato $\textsc{no}$, allora M termina rifiutando.
È chiaro che M termina con successo se e solamente se N fa altrettanto — andrebbe però dimostrato in dettaglio!
Rimane da verificare che il tempo impiegato da M nella simulazione è quello dell'enunciato. Quante sono le successioni da visitare? al massimo $\mathcal{O}(d^{f(n)+1})$, ciascuna di costo massimo $f(n)$, quindi basta trovare un c opportuno. □

Terminiamo con la definizione di spazio non deterministico e della classe di problemi decidibili in spazio non deterministico polinomiale. Si noti il quantificatore esistenziale nel secondo punto.

Definizione 3.3.7. La macchina di Turing N, non deterministica a k-nastri di tipo I/O, *decide I in spazio non deterministico $f(n)$* se e solamente se

- N decide I e
- $\forall x \in I\ \exists w_1, \ldots, w_k$ tali che $(q_0, \trianglerighteq x, \ldots, \trianglerighteq) \to_N^* (s_I, w_1, w_2, \ldots, w_k)$ e $\sum_{2 \leq i \leq k-1} |w_i| \leq f(|x|)$.

Se N decide I in spazio non deterministico $f(n)$, allora $I \in \text{NSPACE}(f(n))$. Infine, la classe dei problemi decidibili (da MdT) in spazio non deterministico polinomiale è

$$\text{NPSPACE} = \bigcup_{k \geqslant 1} \text{NSPACE}(n^k).$$

Abbiamo già preannunciato che l'estensione con il non determinismo non allarga la classe dei problemi trattabili polinomialmente, quando la misura della complessità riguardi lo spazio. Enunciamo ora tale teorema, senza dimostrarlo.

Teorema 3.3.8 (Savitch)**.** NPSPACE = PSPACE.

Nonostante nella soluzione di un problema I ottenuta per forza bruta o fidandosi della buona sorte e della magia non si sfrutti per niente la struttura matematica di I, forse perché essa è troppo complessa, o perché essa sfugge al solutore del problema, vi sono numerosi esempi di problemi per cui quello è l'unico modo conosciuto di arrivare a una soluzione; ne menzioneremo piú avanti alcuni, sui quali ritorneremo ancora nel prossimo capitolo. Per ora limitiamoci a discutere brevemente un esempio paradigmatico per la soluzione del quale si conoscono solo algoritmi polinomiali non deterministici o esponenziali deterministici: il problema del commesso viaggiatore.[11]

Esempio 3.3.9 (Il problema del commesso viaggiatore)**.**
Vi sono n città, ciascuna individuata da un intero e collegata a tutte le altre da una strada percorribile nei due sensi; sia allora $d(i, j)$ la distanza tra le città i e j.[12] Il problema consiste nel trovare il cammino di costo minimo che attraversa tutte le città una e una volta sola — ovvero dobbiamo trovare una permutazione di indici (o città) $\Pi : [1..n] \to [1..n]$ che minimizza la seguente quantità, in cui intendiamo $i = \Pi(h), i + 1 = \Pi(k)$ per qualche h e k:

$$\sum_{1 \leqslant i \leqslant n-1} d(i, i + 1)).$$

Ovviamente, il problema si può rappresentare come un grafo (non diretto) i cui nodi sono le città e in cui c'è un arco tra ogni coppia di nodi i e j, pesato da $d(i, j)$.

[11] Stiamo parlando di soluzione *esatta*; nel caso in cui ci si accontenti di una soluzione approssimata vi sono algoritmi molto piú furbi, sulla cui natura ritorneremo brevemente piú avanti.

[12] Una distanza d è una funzione da coppie di punti, nel nostro caso città, nei reali positivi, tale che gode delle seguenti tre proprietà
- *riflessiva*: $d(i, j) = 0$ se e solamente se $i = j$;
- *simmetrica*: $d(i, j) = d(j, i)$;
- *triangolare*: $d(i, j) \leq d(i, k) + d(k, j)$.

Per vedere questo problema come un problema di decisione si abbia un valore limite B, da interpretarsi come il rimborso viaggi assegnato al commesso viaggiatore; allora il problema è risolto positivamente se c'è un cammino che tocca tutte le città una e una sola volta di costo minore o uguale a B.

Vediamo adesso di calcolare, in modo assai spiccio, la complessità prima di una (tipica) procedura che impiega il metodo di forza bruta per risolvere la variante di decisione del problema del commesso viaggiatore e poi di una (tipica) MdT non deterministica che lo risolve; entrambe procedono in due fasi separate. Il dato iniziale delle due macchine è ovviamente la rappresentazione della rete stradale e del costo B (per esempio come matrice di incidenza determinata dalle distanze).

Come tutte le procedure a forza bruta, il primo metodo ha la seguente struttura:

i) genera tutte le potenziali soluzioni, come permutazioni di tutti gli interi fino a n, il che costa $\frac{(n-1)!}{2}$

ii) scegli tra le permutazioni la prima che ha costo accettabile, ovvero minore di B, e questo controllo può essere fatto in tempo deterministico cubico (bisogna accedere $\mathcal{O}(n)$ volte alle $\mathcal{O}(n^2)$ coppie (i, j) per ottenere la distanza $d(i, j)$ memorizzata nel nastro di ingresso).

(Può essere interessante vedere che lo spazio necessario alla procedura sopra delineata è in $\mathcal{O}(n)$, perché è sufficiente generare una permutazione alla volta e memorizzare in uno spazio di lavoro la permutazione in esame.)

Costruiamo adesso una macchina di Turing non deterministica N che decide il problema in $\mathcal{O}(n^3)$. N procede con le due fasi seguenti:

i) N scrive su un nastro di lavoro una stringa di n numeri naturali compresi tra 1 e n; cioè vi sono n transizioni possibili a partire dalla configurazione iniziale, ciascuna che scrive un numero nell'intervallo $[1..n]$ sul nastro di lavoro e dalle configurazioni cosí raggiunte vi sono n transizioni, che scrivono un numero in $[1..n]$, e cosí via — ovviamente la scelta della transizione da effettuare viene fatta *a caso* (naturalmente la macchina potrebbe esser piú furba). Questa fase richiede $\mathcal{O}(n)$ passi;

ii) N verifica se la stringa è un cammino accettabile, cioè è una permutazione degli indici, usando un altro nastro di lavoro, in tempo $\mathcal{O}(n^2)$;
N verifica di seguito (o contemporaneamente) se il costo del cammino è minore o uguale a B, di nuovo in tempo $\mathcal{O}(n^3)$, nel qual caso risponde positivamente; altrimenti risponde negativamente.

Si noti come la prima fase che compiono entrambe le procedure consiste nel generare una delle soluzioni, le quali sono in numero esponenziale; la differenza è che nel primo algoritmo si procede costruttivamente, generandole tutte, mentre nel secondo si tira a indovinare, o meglio ci si fa dare la soluzione, se c'è, da un mago o una fata, sfruttando cosí il meccanismo non deterministico della macchina usata. In altre parole, in entrambi i casi si genera esaustivamente tutto lo spazio del problema su cui fare poi la ricerca della soluzione, ma nel primo modo ciò avviene *esplicitamente*, nel secondo *implicitamente*. È importante osservare che la seconda fase è una *certificazione in tempo polinomiale* fatta in modo *deterministico* che la stringa di naturali è davvero una soluzione. Uno potrebbe, e spesso viene fatto, definire allora la classe $\mathcal{NP}$ come l'insieme dei problemi che ammettono una certificazione in tempo polinomiale. Tuttavia, dovrebbe a questo punto essere chiaro che i due modi per definire tale classe, o via MdT non deterministiche o via certificazione polinomiale, sono del tutto equivalenti e differiscono solo dal punto di vista con cui si esaminano i problemi che vi appartengono. Infine, notiamo che, quando il cammino in esame non è una soluzione, il metodo esaustivo esplicito ci dice che dobbiamo visitare tutto lo spazio degli stati, che sono in numero esponenziale, quindi la certificazione del fallimento avviene in tempo esponenziale: *tutti* i tentativi falliscono.

Come già detto piú volte, questo modo di procedere trova applicazioni in molti campi. Ad esempio, in logica per dimostrare che una formula, decidibile e lunga n, è un teorema, uno può generare "tutte le dimostrazioni" le cui ultime formule sono lunghe n e se ci trova la formula di partenza, allora questa è un teorema. Oppure, per decidere se c'è un assegnamento di valori di verità alle variabili di una formula che la rende vera, si considerano tutti i possibili assegnamenti e poi si calcola il valore della formula in ciascun caso. Strettamente correlato a quest'ultimo problema, c'è, in teoria dei circuiti, il problema di decidere quando in un dato circuito passa un segnale. Nell'area dell'ottimizzazione combinatoria, oltre al problema del commesso viaggiatore e di quelli a esso correlati, si risolvono in modo esatto con questa tecnica i problemi di assegnazione di risorse, tra i quali l'allocazione dei task (non prerilasciabili) ai processori, compito fondamentale dei sistemi operativi. Infine, nel campo dell'intelligenza artificiale, ci sono tutti i problemi legati ai giochi o alla ricerca e all'estrazione di informazione da grosse collezioni di dati, anche non strutturate.

3.4 Funzioni di misura, un po' di gerarchia e due assiomi

Prima di studiare un po' piú in dettaglio le classi $\mathcal{P}$ e $\mathcal{NP}$ facciamo una rapida carrellata su alcuni risultati di complessità, che enunceremo senza dimostrare; discuteremo anche brevemente l'influsso che sulla loro validità hanno la forma e il tipo delle funzioni che limitano le risorse che sono necessarie alle macchine per risolvere un problema, per esempio delle funzioni che stimano il numero di passi e di celle necessari.

In principio, una funzione di misura $f : \mathbb{N} \to \mathbb{N}$ può essere una qualunque funzione totale, anche avere essa stessa complessità enorme. Vedremo subito che funzioni di misura arbitrarie portano a risultati che sono quantomeno controintuitivi, e pertanto considereremo di seguito una classe di funzioni di misura, dette *appropriate*, anche se, a prima vista, possono sembrare particolarmente limitate. Intuitivamente, le funzioni appropriate non richiedono piú risorse di quanto stimato dal loro stesso valore, cioè l'algoritmo che calcola $f(x)$ deve richiedere tempo $\mathcal{O}(f(|x|) + |x|)$ (l'addendo $|x|$ interviene quando f è sub-lineare, per rispettare l'ipotesi fatta che le funzioni di complessità in tempo siano almeno lineari) e richiedere spazio $\mathcal{O}(f(|x|))$.

Sotto queste ipotesi possiamo enunciare il teorema che risponde a una delle nostre prime domande: esiste davvero una gerarchia di problemi. In altre parole, aumentando le risorse a disposizione delle nostre macchine si risolvono piú problemi — ciò che del resto ci dicono intuizione ed esperienza. Successivamente, vedremo anche che ci sono problemi arbitrariamente complessi, e che quindi non vi è una classe che domini tutte le altre e che di conseguenza la gerarchia non è composta da un numero finito di classi.

In effetti, l'ultimo teorema vale anche nel caso in cui si considerino funzioni di misura qualsiasi, purché totali. Ciò suggerirebbe di allargare la classe delle funzioni appropriate a quella delle funzioni totali: mal ce ne incoglierà! Vi sono un paio di teoremi che distruggono la gerarchia appena stabilita e quindi l'estensione fatta non raggiunge lo scopo che ci eravamo prefissato. Il primo teorema dice che, per funzioni di misura arbitrarie, esistono problemi che non hanno un algoritmo ottimo; il secondo che vi sono delle lacune tra classi di complessità.

Infine, a puro titolo di curiosità, discuteremo di una classe di funzioni di misura della complessità, la cui definizione, sottesa da un'intuizione accettabilissima, è molto elegante, in quanto consta di due assiomi e non richiede alcun riferimento esplicito alle risorse di calcolo che esse richiedono per la soluzione del problema. Purtroppo, valgono anche in questo caso i risultati anti-intuitivi citati sopra, e quindi ci tocca contentarci delle funzioni

appropriate che allora useremo nel seguito, salvo che in questo breve capitolo.

Iniziamo con una buona definizione di funzione per misurare il consumo delle risorse: le funzioni appropriate di cui abbiamo parlato sopra. Anche per esse vi sono molte definizioni leggermente diverse e molti nomi diversi: funzioni *oneste*, *costruibili* (in tempo o in spazio) e altre ancora. Per i nostri scopi è sufficiente la seguente definizione.

Definizione 3.4.1. La funzione $f : \mathbb{N} \to \mathbb{N}$ calcolabile totale è *appropriata* se

i) è monotona crescente (cioè $n \geqslant m$ implica $f(n) \geqslant f(m)$);

ii) esiste una MdT M a k nastri, tale che $\forall x \in \Sigma^*$ si arresta (dando come risultato $\diamond^{f(|x|)}$, con $\diamond \notin \Sigma$ simbolo speciale) in tempo $\mathcal{O}(f(|x|) + |x|)$ e in spazio $\mathcal{O}(f|x|)$.

Le condizioni poste a una funzione affinchè sia appropriata sono piuttosto onerose e sembrano mordersi la coda. Tuttavia le funzioni che si vedono di solito usare per dare la complessità degli algoritmi sono tutte appropriate. Ad esempio sono funzioni appropriate $k, n, n^k, k^n, n!, \lfloor \log n \rfloor, \lfloor \sqrt{n} \rfloor$ (si noti che le ultime due sono sub-lineari); inoltre se f, g sono appropriate, lo sono anche $f + g$ e $f \times g$, quindi tutti i polinomi sono funzioni appropriate; infine anche f^g e $g \circ f$ lo sono, assieme a molte altre funzioni d'uso comune.

Adesso andiamo a stabilire che le classi dei problemi decidibili con risorse fissate, misurate con funzioni appropriate, aumentano al crescere delle risorse stesse. Ciò significa che, al variare di queste, si può stabilire una gerarchia tra le classi di complessità.

Teorema 3.4.2 (Gerarchia di tempo e spazio)**.** *Se f è appropriata,*

- $\mathrm{TIME}(f(n)) \subsetneq \mathrm{TIME}((f(2n+1))^3)$;
- $\mathrm{SPACE}(f(n)) \subsetneq \mathrm{SPACE}(f(n) \times \log f(n))$.

Dimostrazione. Omessa. Notiamo soltanto che quella del primo punto si basa sulla dimostrazione che il problema (ancora la diagonalizzazione!):

$$\{x \mid \varphi_x(x) \textit{ converge entro } f(|x|) \textit{ passi}\}$$

appartiene a $\mathrm{TIME}(f^3)$, ma *non* a $\mathrm{TIME}(f)$ (si veda anche l'esempio 1.10.1 e la nota alla sua fine); la dimostrazione del secondo punto è analoga. □

Naturalmente, accanto a questo teorema ce n'è uno del tutto analogo che riguarda le misure non deterministiche. (Si noti di sfuggita che se $f(n)$ è un polinomio, lo è anche $f(2n+1)^3$.)

In realtà, la gerarchia è ben piú densa, perché l'esponente 3 che compare nella formulazione del teorema precedente è determinato dalla particolare dimostrazione scelta. In realtà, vi sono molte classi significative tra TIME(f^3) e TIME(f); analogamente per lo spazio. Questo frammento di gerarchia ci basta comunque per dimostrare il seguente corollario, in cui si afferma che la classe dei problemi decidibili in tempo polinomiale deterministico è strettamente inclusa in quella dei problemi decidibili in tempo deterministico esponenziale, che abbiamo già incontrato e che adesso introduciamo esplicitamente:

$$\mathrm{EXP} = \bigcup_{k \geqslant 1} \mathrm{TIME}(2^{n^k}).$$

Corollario 3.4.3.

$$\mathcal{P} \subsetneq \mathrm{EXP}$$

Dimostrazione. L'inclusione è ovvia perchè 2^n cresce piú velocemente di ogni polinomio; è propria perchè

$\mathcal{P} \subseteq \mathrm{TIME}(2^n) \subsetneq \mathrm{TIME}((2^{(2n+1)})^3) \subseteq \mathrm{TIME}(2^{n^3})$. □

Il fatto che $\mathrm{NTIME}(f(n)) \subseteq \mathrm{TIME}(c^{f(n)})$ (teorema 3.3.6), ci permette di concludere che $\mathcal{NP}$ è inclusa in EXP, ma non basta nemmeno con il corollario di sopra a gettar luce sul problema che l'inclusione $\mathcal{P} \subseteq \mathcal{NP}$ sia stretta o meno. Riceve invece giustificazione anche il modo di procedere per forza bruta: genera tutti i candidati a essere una soluzione in tempo deterministico esponenziale e poi certifica che un candidato è davvero una soluzione (in tempo deterministico polinomiale).

Per dare un quadro assai sommario, ma un pochino piú completo della gerarchia che si può stabilire tra le classi di complessità sia in tempo che in spazio e nelle loro versioni deterministiche e non deterministiche, riportiamo senza dimostrarli i seguenti risultati.

Teorema 3.4.4. *Siano f una funzione di misura appropriata e k una costante, allora*

- $\mathrm{SPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$
- $\mathrm{TIME}(f(n)) \subseteq \mathrm{NTIME}(f(n))$
- $\mathrm{NSPACE}(f(n)) \subseteq \mathrm{TIME}(k^{\log n + f(n)})$

Inoltre ricordiamo che avevamo già stabilito i seguenti fatti.

Teorema 3.4.5.

- LOGSPACE $\subseteq \mathcal{P}$
- LOGSPACE $\subsetneq$ PSPACE
- PSPACE = NPSPACE
- $\mathcal{NP} \subseteq$ EXP

Come anticipato, vediamo che la gerarchia non è superiormente limitata, ovvero che ci sono problemi arbitrariamente difficili, indipendentemente dagli algoritmi usati per risolverli. In questo teorema, la cui dimostrazione omettiamo, si può rilasciare senza timori la richiesta che le funzioni di misura siano appropriate.

Teorema 3.4.6. *Per ogni funzione calcolabile totale g esiste un problema $I \in \text{TIME}(f(n))$ e $I \notin \text{TIME}(g(n))$ con $f(n) > g(n)$ quasi ovunque.* [13]

Abbiamo già preannunciato che usare funzioni di misura arbitrarie, per esempio funzioni che crescono vertiginosamente, porta a conclusioni bizzarre. Le enunceremo, cercando di dar loro un minimo di intuizione, mediante i due teoremi che seguono e che non dimostreremo. [14]

Il primo risultato riguarda l'esistenza di algoritmi ottimi per risolvere un dato problema, ovvero algoritmi le cui prestazioni non possono essere migliorate che per una costante moltiplicativa. Una parte importantissima dell'informatica si occupa di trovare algoritmi ottimi per un problema, sia per ragioni di efficienza concreta, sia perché in questo modo si ottiene una misura precisa della difficoltà di un problema e dell'efficienza dei programmi che lo risolvono. Il risultato che riportiamo è negativo: non sempre esiste un algoritmo ottimo.[15]

Teorema 3.4.7 (di accelerazione, Blum)**.** *Per ogni funzione calcolabile totale h, esiste un problema I tale che, per ogni algoritmo M che decide I in tempo f esiste M' che decide I in tempo f' tale che*

$$f(n) > h(f'(n)) \text{ quasi ovunque}$$

[13] Cioè tranne che per dati di ingresso tratti da un insieme *finito*.

[14] Non sorprenderà che le loro dimostrazioni siano basate su un ragionamento di tipo diagonale.

[15] Una visione intuitiva di questo teorema, suggerita da Börger, è che ci sono dei programmi che sono piú veloci su una macchina (universale) vecchia e lenta che su una macchina (universale) nuova e veloce.

In altre parole, si può dimostrare l'esistenza di una successione di algoritmi via via piú efficienti per risolvere il problema costruito in funzione di h, che è una funzione calcolabile totale, ma del tutto arbitraria. Per esempio fissata la funzione h, il teorema dice che si può trovare un algoritmo, magari esponenzialmente piú efficiente di M, un terzo esponenzialmente piú efficiente del secondo, e cosí via; o peggio se la funzione h cresce ancor piú velocemente. Si noti che non poniamo alcun vincolo sulla funzione f.
ATTENZIONE: data la funzione h, il problema che si costruisce per dimostrare il teorema è "artificiale", nel senso che non corrisponde ad alcun problema incontrato prima; inoltre, si sa solo che questa successione di algoritmi via via piú efficienti *esiste*, ma *non* come si costruiscono l'uno dall'altro!

Se anche fosse sopportabile la situazione evidenziata dal teorema precedente, quando si usano funzioni di misura che non sono appropriate si scopre che all'aumentare delle risorse non si allarga la classe dei problemi decisi con esse. Di conseguenza si contraddice l'idea intuitiva che ci debba essere una gerarchia di classi di complessità, come espresso dal teorema 3.4.2 che la stabilisce nel caso in cui le funzioni di misura siano *appropriate*. L'enunciato che segue è una forma speciale del teorema generale: invece che prendere una generica funzione calcolabile h, con $h(n) > n$, al variare della quale si costruisce la f usata sotto, consideriamo per semplicità la funzione 2^n. Il risultato netto è che non si trova alcun problema decidibile in tempo deterministico (strettamente) superiore a $f(n)$ e (strettamente) inferiore a $2^{f(n)}$.[16]

Teorema 3.4.8 (della lacuna, Trakhtenbrot-Borodin)**.**
Esiste f calcolabile totale tale che $\mathrm{TIME}(f(n)) = \mathrm{TIME}(2^{f(n)})$.

Una formulazione piú fine di questo teorema richiede che la funzione f considerata sia monotona; se non lo fosse non sorprenderebbe poi molto che non vi siano problemi la cui soluzione richiede risorse comprese tra due funzioni che oscillano. Inoltre, si può vedere (dalla dimostrazione) che la funzione f che individua la lacuna cresce in maniera estremamente rapida, ciò che è proibito alle funzioni di misura appropriate.

Naturalmente, valgono anche i teoremi di accelerazione e della lacuna che si ottengono sostituendo misure di spazio invece che di tempo nell'enunciato dei due teoremi precedenti.

In un certo senso, l'ultima osservazione fatta ribadisce la sensazione che si potrebbe sviluppare una teoria della complessità che sia indipendente dallo spazio o dal tempo usati, ma dipenda solo da una nozione astratta di risorsa, istanziabile ad esse, ma anche ad altre risorse, quali l'energia consumata

[16]Di nuovo, una visione intuitiva, anche questa dovuta a Börger, suggerisce che c'è un insieme di programmi sui quali la macchina lenta e quella veloce si equivalgono.

o altro. Vediamo allora un'altra caratterizzazione delle funzioni di misura, dovuta a Blum, che si muove in quest'ottica e che sta alla base del cosidetta approccio assiomatico alla complessità. Questa caratterizzazione è particolarmente elegante in quanto è slegata non solo da ogni modello di calcolo ma anche dalle risorse il cui consumo si vuole misurare, e soprattutto perché la sua definizione consiste di soli due assiomi.

Definizione 3.4.9. Una funzione ϕ è una misura di complessità se restituisce un naturale a fronte di una funzione ψ e del suo dato di ingresso x e inoltre soddisfa entrambi i seguenti assiomi:

A1. $\phi(\psi, x)$ è definita se e solo se $\psi(x)$ lo è;

A2. per ogni ψ, x, k, è decidibile se $\phi(\psi, x) = k$.

Il primo assioma ci dice che ϕ misura la complessità del calcolo di $\psi(x)$; il secondo assicura che si può davvero ottenere la complessità del calcolo di $\psi(x)$. Quindi, se la funzione ϕ misurasse il numero dei passi, ri-otterremmo la vecchia definizione di complessità in tempo; se misurasse le celle toccate quella in spazio.[17]

Non è difficile verificare che funzioni di misura appropriate soddisfano gli assiomi A1 e A2. Tuttavia vi sono funzioni di misura che soddisfano tali assiomi e il cui impiego conduce a risultati ancora una volta sorprendenti. Intanto, molti dei teoremi sulla complessità possono venir dimostrati usando solamente questa formulazione, in particolare quelli dell'accelerazione e della lacuna. Inoltre, può capitare che la complessità della composizione sequenziale di due algoritmi risulti *minore* della complessità del primo dei due; in altre parole, fare ulteriori calcoli può ridurre la complessità del problema appena risolto! Non possiamo allora che rassegnarci a usare funzioni appropriate.

[17]Purché si accetti di non saper valutare lo spazio necessario a una macchina non terminante, anche nel caso in cui questa sia in ciclo.

3.5 $\mathcal{P}$ e $\mathcal{NP}$

In questo capitolo cercheremo di esaminare un po' piú in dettaglio le classi di complessità $\mathcal{P}$ e $\mathcal{NP}$, studiandone seppur superficialmente la struttura. Lo strumento principale che useremo, se non l'unico, è la ricerca di uno o piú problemi completi per ciascuna di tali classi, cioè quei problemi che vi appartengono e di questa sono i piú ardui, ovvero son tali per cui a loro si riducono tutti gli altri problemi della classe. Piú precisamente, considereremo la relazione di riduzioni $\leqslant_{\text{logspace}}$, cioè la relazione che impiega riduzioni, o algoritmi, di complessità LOGSPACE $= \bigcup_{k \geqslant 1} \text{SPACE}(k \times log\ n)$, la quale classifica $\mathcal{P}$ e $\mathcal{NP}$; poi andremo a cercare problemi H che siano $\mathcal{P}$-completi, cioè tali per cui ogni problema I in $\mathcal{P}$ si riduce a H (cioè H è arduo) e inoltre H appartiene a $\mathcal{P}$ (cioè H è completo) (si veda la definizione 1.10.12); analogamente per $\mathcal{NP}$.

I problemi completi caratterizzano davvero una classe di complessità, perchè ne esprimono la struttura profonda e l'essenza e la difficoltà dei suoi problemi. Di conseguenza attraverso un problema completo si esprime anche il potere espressivo di una classe. A questo proposito, si ricordi l'insieme K che è RE-completo per riduzioni calcolabili totali (cosí come lo sono K_0 e K_1) e il ruolo che esso gioca nel determinare i gradi di solubilità e insolubilità. Inoltre, sia detto per inciso, vale poco una classe che non ha problemi completi *pre-esistenti* alla sua definizione e che siano interessanti dal punto di vista computazionale. Vedremo in questo capitolo alcuni problemi completi per $\mathcal{P}$ e per $\mathcal{NP}$, i quali sono davvero interesssanti e che son stati posti ben prima che nascesse la teoria della complessità come la conosciamo ora. Per ritornare sull'analogia con le classi di problemi calcolabili e di problemi non calcolabili, notiamo che l'insieme K_0 è davvero interessante: vorremmo, anche se da questa ricerca ne usciamo frustrati, uno strumento che dica, per ogni programma, se esso terminerà sempre — in altre parole stiamo dichiarando che l'Entscheidungproblem proposto da Hilbert nel 1900 è interessante.

Prima di procedere, ricordiamo anche che si può stabilire il seguente frammento di gerarchia tra le classi di complessità che andremo a esaminare:

$$\text{LOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \text{PSPACE} = \text{NPSPACE}$$

e ricordiamo anche che LOGSPACE $\subsetneq$ PSPACE. Di conseguenza, almeno una delle inclusioni

$$\text{LOGSPACE} \subseteq \mathcal{P},\ \mathcal{P} \subseteq \mathcal{NP},\ \mathcal{NP} \subseteq \text{PSPACE}$$

deve essere stretta. A tutt'oggi però non sappiamo quale essa sia, e non ci aiuta sapere anche che

$$\mathcal{P} \subsetneq \text{EXP} \quad \text{e} \quad \mathcal{NP} \subseteq \text{EXP}.$$

Il problema piú interessante in questo momento è risolvere $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, che, oltre che estremamente affascinante si è rivelato resistentissimo, tanto che non si sa neppure se sia dimostrabile. Naturalmente, se facessimo vedere che un problema $\mathcal{NP}$-completo è risolto da un algoritmo che richiede tempo polinomiale deterministico, o che esso si riduce a un problema, necessariamente completo, che sta in $\mathcal{P}$, avremmo d'un botto dimostrato l'eguaglianza tra la classe dei problemi "trattabili" e dei problemi che divengono tali grazie al non determinismo (in accordo al paradigma: scommetti su una soluzione e dimostra che lo è in tempo polinomiale deterministico). Vi è tuttavia una sensazione diffusa che l'eguaglianza non valga, a ciò concorrendo molti indizi. Tra questi segnaliamo soltanto il fatto che se $\mathcal{P}$ coincidesse con $\mathcal{NP}$ ci sarebbe un modo computazionalmente accettabile di realizzare il meccanismo del non determinismo, cosa che appare per ora non poco sorprendente; a questo proposito si veda anche la nota a pagina 152 su due modelli di calcolo che paiono raggiungere tale obbiettivo.

Prima di iniziare discutiamo brevemente la robustezza delle classi $\mathcal{P}$ e $\mathcal{NP}$ e poi vediamo alcune critiche alla scelta di costruirre una teoria asintotica che considera sempre il caso pessimo, e come queste si riverberano sulla tesi di Cook-Karp.

3.5.1 Problemi trattabili e problemi intrattabili

Adesso che abbiamo le nozioni di $\mathcal{P}$ e $\mathcal{NP}$ e che abbiamo visto come operano le macchine di Turing non deterministiche, possiamo rivedere l'affermazione che i problemi in $\mathcal{P}$ sono *trattabili* e quelli in $\mathcal{NP}$ *intrattabili*, ovvero discutiamo della fondatezza della *tesi di Cook-Karp*.

A sostegno di questa tesi vi è la robustezza delle classi che stiamo considerando non solo rispetto alla rappresentazione dei dati e dei problemi, ma anche rispetto al cambio di modelli di calcolo e di trasformazioni "facili" dei problemi stessi; vediamo in un dettaglio appena maggiore di quanto fatto che cosa intendiamo con le ultime due cose.

i) Le classi $\mathcal{P}$ e $\mathcal{NP}$ sono robuste nel senso che non variano al variare dei modelli di calcolo, a patto che le funzioni per misurare il tempo siano "ragionevoli" (vedi l'esempio 3.2.11 sulla RAM). In altre parole $\mathcal{P}$ è robusta perché è chiusa rispetto la composizione polinomiale sinistra. [18] L'idea qui è che si può sempre trovare un algoritmo T di complessità p che, dato un problema I, *trasforma* una sua procedura di decisione M, espressa in un modello $\mathcal{M}$, in una sua procedura *equivalente* M',

[18] Una classe $\mathcal{C}$ di funzioni è chiusa rispetto la composizione polinomiale sinistra se per tutti i polinomi p il fatto che $f \in \mathcal{C}$ implica $p \circ f \in \mathcal{C}$.

espressa in un altro modello $\mathcal{M}'$. Se M ha complessità polinomiale f, allora M' avrà complessità $p \circ f$, che è ancora un polinomio. In termini piú generali, robustezza rispetto al cambio di modelli di calcolo significa che le trasformazioni tra essi sono polinomiali (ad esempio, si consideri come un programma *WHILE* può simulare in tempo polinomiale una macchina di Turing).

ii) La classe $\mathcal{P}$ è chiusa rispetto alla somma, al prodotto, alle riduzioni $\leqslant_F$ con F (sotto-)classe dei polinomi. Questo verrà formalizzato definendo appropriate riduzioni che classificano $\mathcal{P}$ e $\mathcal{NP}$ e dimostrando il teorema 3.5.2. Per amore di simmetria rispetto quanto fatto sopra, ciò si può rifrasare prendendo riduzioni polinomiali e dicendo che $\mathcal{P}$ è chiusa rispetto alla composizione polinomiale destra.[19] In altre parole, per risolvere un problema I, trasformalo "facilmente", ovvero *riducilo*, per mezzo di un algoritmo M, che ha come complessità un polinomio p, in $I' = M(I)$ e decidi I'; se $I' \in \mathcal{P}$, cioè I' è deciso in tempo deterministico polinomiale f, anche $I \in \mathcal{P}$ perché è deciso in tempo deterministico $f \circ p$, che è ancora un polinomio.

Una ulteriore difesa della tesi di Cook-Karp è che di solito gli algoritmi polinomiali hanno, o almeno se ne cercano in modo che le costanti moltiplicative e soprattutto gli esponenti siano piccoli, mentre gli algoritmi esponenziali diventano rapidamente inefficienti al crescere dei dati di ingresso. Tuttavia ci sono alcune critiche all'identificazione di $\mathcal{P}$ con i problemi trattabili:

i) Un algoritmo in $\mathcal{O}(n^{100})$ non è certo efficiente. Lo è molto di piú, almeno per n non enormi, uno in $\mathcal{O}(2^{\frac{n}{100}})$ o in $\mathcal{O}(n^{\log n})$.

ii) Ci sono algoritmi che richiedono tempo esponenziale nel caso pessimo, ma sono efficienti nei casi interessanti o almeno in quelli piú comuni. Si possono trovare degli esempi di questo comportamento bizzarro in:

- *programmazione lineare:* l'algoritmo del simplesso (Kantorovich, durante l'assedio di Leningrado 1941-43, perfezionato da Leontief 1949) è semplice e intuitivo e, benché sia esponenziale nel caso pessimo, di solito è efficiente (v. oltre), mentre il metodo elissoide è polinomiale, ma incredibilmente lento a causa di costanti moltiplicative enormi; val la pena di notare che recentemente è stato proposto un metodo, assai sofisticato dal punto di vista matematico, detto dei "punti interni" che è polinomiale, su cui sono basate alcune realizzazioni che sono in svariati casi pratici piú efficienti

[19] Una classe $\mathcal{C}$ di funzioni è chiusa rispetto composizione polinomiale destra se per tutti i polinomi P, $f \in \mathcal{C}$ implica che $f \circ p \in \mathcal{C}$.

del simplesso. Ovvero vale la pena di insistere nella ricerca di algoritmi piú efficienti!

- *algoritmi di paginazione:* si supponga di avere un programma che richiede che le pagine vengano continuamente caricate e scaricate; in questo caso tutti gli algoritmi di paginazione hanno la stessa complessità (la peggiore!), ma sappiamo bene che nella pratica ci sono algoritmi migliori di altri.
- *inferenza di tipi di Standard ML: ogni* algoritmo è efficiente in pratica, ma esponenziale nel caso pessimo, il quale è costruito *ad hoc* e risulta alquanto artificiale.

Il primo punto suggerisce anche una critica all'aver definito una teoria asintotica della complessità, mentre gli altri due sollevano una critica al calcolo della complessità nel caso pessimo. Queste scelta sono molto forti, anche se permettono una trattazione matematica semplice e consentono di dimostrare buone proprietà di chiusura della teoria.

Ci sono diversi modi alternativi per analizzare la complessità di un algoritmo, e qui ne citiamo un paio, rimandando il lettore alla letteratura per una trattazione piú accurata e completa.

Una prima alternativa alla complessità nel caso pessimo è considerare invece il caso *medio*: si valuta la complessità del problema quando i suoi dati siano *medi*. Ma come si determinano i dati medi? Farlo è in generale molto difficile e richiede di conoscere la distribuzione dei dati o quanto meno di poterla approssimare. A volte queste stime sono grossolane e arbitrarie o addirittura non si riescono a definire. A volte invece le approssimazioni proposte funzionano benissimo, soprattutto quando il problema abbia una "buona" struttura. Ad esempio, ci sono metodi probabilistici che definiscono il caso medio per il problema della programmazione lineare citato sopra e sotto tali ipotesi l'algoritmo del simplesso diventa efficiente, avendo complessità polinomiale con costanti piccole: sotto ipotesi accettabili, per un sistema con n disequazioni servono n^2 iterazioni — in barba al metodo dei punti interni.

L'efficienza del simplesso è inoltre sostenuta da una recente proposta, chiamata *smooth complexity* [Spielman e Teng], che tende a combinare i vantaggi dell'approccio alla complessità dal punto di vista del caso pessimo e di quello medio. Secondo questo tipo di analisi, che misura la complessità di un algoritmo perturbando leggermente i dati nel caso pessimo, il simplesso risulta di nuovo essere polinomiale, cosí come accade quasi sempre.

Un ulteriore approccio alla valutazione della complessità è quello che va sotto il nome di *analisi ammortizzata* [Tarjan]. Molto rozzamente, si considerano tipiche sequenze di k operazioni e il tempo per l'esecuzione di una di esse viene calcolato prendendo la media dei costi di tali operazioni — non

tutte si applicheranno al caso piú sfavorevole, quindi ci si differenzia dalla complessità nel caso pessimo. Inoltre non è una variante dell'analisi del costo medio, in quanto non c'è alcun uso di nozioni probabilistiche o statistiche.

Per la definizione di problemi completi, e quindi nello studio di $\mathcal{P}$ e di $\mathcal{NP}$, abbiamo bisogno di definire quelle che consideriamo riduzioni "facili" e di dimostrare che esse classificano $\mathcal{P}$ e $\mathcal{NP}$, nel senso definito nel capitolo 1.10. Ancora una volta, l'idea è che per risolvere il problema I lo si riduce efficientemente per mezzo di una funzione f a $f(I)$ appartenente a una data classe $\mathcal{D}$ e si risolve il problema ridotto; se la classe è chiusa rispetto a quelle riduzioni, che pertanto possiamo considerare efficienti rispetto $\mathcal{D}$, [20] allora anche il problema I sta in $\mathcal{D}$. Quando si tratta di $\mathcal{P}$ e di $\mathcal{NP}$, nella letteratura si considerano di solito efficienti le riduzioni che sono *polinomiali in tempo deterministico*; noi useremo invece riduzioni *logaritmiche in spazio deterministico*, stante l'uso fatto di macchine di Turing con k nastri che modellano i calcolatori paralleli e la diffusione di questi ultimi. Si noti tuttavia che ogni riduzione logaritmica in spazio è anche polinomiale in tempo, grazie al teorema 3.2.18. Infatti, se $f \in \text{LOGSPACE}$ allora $f \in \mathcal{P}$ e quindi le funzioni di riduzione che useremo in seguito sono piú "facili" di quelle classiche polinomiali in tempo, o almeno altrettanto difficili.

Definizione 3.5.1. (cf. definizione 1.10.10)
Un problema I si *riduce efficientemente* a I' ($I \leqslant_{\text{logspace}} I'$) se esiste un algoritmo $f \in \text{LOGSPACE}$ tale che

$$x \in I \text{ se e solamente se } f(x) \in I'.$$

Adesso vediamo che la classe delle funzioni in LOGSPACE induce una relazione di riduzione che classifica LOGSPACE e $\mathcal{D}$, dove $\mathcal{D}$ è una qualunque delle classi che abbiamo introdotto (si veda la definizione 1.10.10 e si osservi che banalmente $\text{LOGSPACE} \subseteq \mathcal{D}$). Nel teorema seguente enunciamo anche l'analogo per le riduzioni in $\mathcal{P}$, che spesso vengono usate al posto di quelle che impieghiamo noi, senza peraltro modificare i risultati principali. Il lettore è anche invitato a riguardare la breve discussione sulla tesi di Cook e Karp nel capitolo 3.5.1 alla luce del seguente teorema.

Teorema 3.5.2.
Siano $\mathcal{D}, \mathcal{E} \in \{\mathcal{P}, \mathcal{NP}, \text{EXP}, \text{PSPACE}, \text{NPSPACE}\}$ *e* $\mathcal{D} \subseteq \mathcal{E}$

- $\leqslant_{\text{logspace}}$ *classifica* $\{\text{LOGSPACE}\}$ *ed* $\mathcal{E}$
- $\leqslant_{\text{logspace}}$ *e a maggior ragione* $\leqslant_{\mathcal{P}}$ *classificano* $\mathcal{D}$ *ed* $\mathcal{E}$

[20] Ecco perché le riduzioni devono essere "facili" nel senso dato dalla tesi di Cook-Karp: comporre un polinomio con una funzione che domina ogni polinomio ci farebbe uscire dalla classe $\mathcal{P}$.

Dimostrazione. Immediata per entrambi gli enunciati, in quanto tutti i punti richiesti dal lemma 1.10.11 sono banalmente soddisfatti. Si noti in particolare che la composizione di due macchine che operano in spazio logaritmico è ancora una macchina in LOGSPACE. Infatti, basta lanciare la seconda macchina e, non appena questa necessita di un carattere di ingresso, farlo calcolare dalla prima macchina, lanciandola sull'ingresso dato e scrivendo i caratteri via via calcolati sulla medesima casella (si noti che l'output può essere di lunghezza polinomiale; naturalmente serve anche una casella che contenga la posizione del carattere di output appena generato); questo modo di procedere spreca un sacco di tempo e una casella per ricordarci quale carattere è necessario alla seconda macchina, ma stiamo misurando lo spazio! □

3.5.2 Alcuni problemi interessanti e riduzioni efficienti tra essi

Prima di cominciare la nostra indagine su problemi che sono $\mathcal{P}$-completi e su quelli $\mathcal{NP}$-completi, vediamone qualche esempio. Abbiamo già incontrato il problema del commesso viaggiatore che è tale; di seguito ne introdurremo altri presi dalla teoria dei grafi; essendo tutti $\mathcal{NP}$-completi, hanno tutti una struttura profonda molto simile, anche se ciò non è evidente a prima vista. Poi passeremo a considerare un problema preso dalla logica: il problema della soddisfacibilità di una proposizione.[21] Ancor maggiore può sembrare la differenza tra questo problema e quello del commesso viaggiatore, anche per l'apparente distanza tra il calcolo proposizionale e la teoria dei grafi. Faremo vedere che tutti i problemi menzionati in precedenza si riducono al problema della soddisfacibilità di una proposizione; in realtà dimostreremo che *tutti* i problemi in $\mathcal{NP}$ si riducono in spazio logaritmico a tale problema: cioè il problema della soddisfacibilità di una proposizione è esso stesso $\mathcal{NP}$-completo. Sempre nel campo della logica, o meglio in quello affine dei circuiti booleani, prenderemo anche il nostro principale problema $\mathcal{P}$-completo.

Val la pena di osservare che tutti i problemi che introduremo sono interessanti nel campo in cui sono stati studiati, indipendentemente dalla indagine della loro complessità che facciamo qui; ciò risponde positivamente alla domanda se $\mathcal{P}$ e $\mathcal{NP}$ siano classi di complessità interessanti *in se*.

Richiamiamo brevemente alcune nozioni di logica.

[21] Sebbene alcuni autori prevalentemente pisani propendano per un forse piú elegante *soddisfattibilità*, noi continueremo a usare *soddisfacibilità* in quanto tale termine è in uso nella comunità dei logici fin dagli anni trenta e, parafrasando Giacomo Leopardi, nella lingua l'uso è ragione.

Definizione 3.5.3 (Espressioni booleane)**.** Dato un insieme X di variabili $x, x_1, x_2, ...$, le *espressioni booleane* hanno la seguente sintassi

$$B \to tt \mid ff \mid x \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2.$$

Le espressioni $tt, ff, x, \neg x$ sono chiamate *letterali*.

Definizione 3.5.4 (Assegnamento booleano; soddisfacibilità)**.**
Dato un insieme finito di variabili X', un *assegnamento booleano* è una funzione

$$\mathcal{V} : X' \to \{tt, ff\}$$

Un'espressione B è *chiusa* se non vi appaiono variabili $x \in X$; un assegnamento $\mathcal{V}$ è *buono* se assegna un valore di verità a tutte le variabili di B; d'ora in avanti useremo solo assegnamenti buoni e ometteremo pertanto tale aggettivo.
Un assegnamento $\mathcal{V}$ *soddisfa* B se vale il predicato $\mathcal{V} \models B$ cosí definito:

$$\begin{array}{lll}
\mathcal{V} \models tt & & \\
\mathcal{V} \models x & \text{se } \mathcal{V}(x) = tt & \\
\mathcal{V} \models \neg B & \text{se } \mathcal{V} \not\models B & \textit{(tertium non datur)} \\
\mathcal{V} \models B_1 \vee B_2 & \text{se } \mathcal{V} \models B_1 \text{ o } \mathcal{V} \models B_2 & \\
\mathcal{V} \models B_1 \wedge B_2 & \text{se } \mathcal{V} \models B_1 \text{ e } \mathcal{V} \models B_2 &
\end{array}$$

Infine, una espressione B è *soddisfacibile* se e solamente se $\exists \mathcal{V}.\ \mathcal{V} \models B$ ed è *valida* se e solamente se $\forall \mathcal{V}.\ \mathcal{V} \models B$.

Esempio 3.5.5. $\mathcal{V} = (x \to ff, y \to ff, z \to tt) \models (x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee \neg z)$.
Infatti, tutti tre i congiunti risultano veri. Il primo $\mathcal{V} \models (x \vee \neg y)$ vale se
1. $\mathcal{V} \models x$, che non vale in quanto $\mathcal{V}(x) = ff$, oppure
2. $\mathcal{V} \models \neg y$ se $\mathcal{V} \not\models y$ che vale in quanto $\mathcal{V}(y) = ff$;
similmente valgono sia $\mathcal{V} \models y \vee z$, perché $\mathcal{V} \models z$, che $\mathcal{V} \models \neg x \vee \neg z$, perché $\mathcal{V} \models \neg x$.

Il seguente teorema è ben noto e utilizza le altrettanto ben note forme normali per le espressioni booleane che qui ricordiamo: B è in *forma normale congiuntiva* (rispettivamente *disgiuntiva*) se è nella forma $\bigwedge_{1 \leqslant i \leqslant n} C_i$, dove i congiunti hanno la forma $C_i = \bigvee_{1 \leqslant j \leqslant k_i} L_{i,j}$, con $L_{i,j}$ letterali (rispettivamente $\bigvee_{1 \leqslant i \leqslant n} C_i$, con disgiunti $C_i = \bigwedge_{1 \leqslant j \leqslant k_i} L_{i,j}$, e $L_{i,j}$ letterali).

Teorema 3.5.6. *Data un'espressione booleana B, se ne può costruire una equivalente B' in forma normale congiuntiva (rispettivamente disgiuntiva); cioè $\forall \mathcal{V}.\ \mathcal{V} \models B$ se e solamente se $\mathcal{V} \models B'$.*

Dimostrazione. Per costruire la dimostrazione, basta ricordare che

$$\begin{array}{rcll} B_1 \vee (B_2 \wedge B_3) & \text{sse} & (B_1 \vee B_2) \wedge (B_1 \vee B_3) & (\vee \text{ distribuisce su } \wedge) \\ B_1 \wedge (B_2 \vee B_3) & \text{sse} & (B_1 \wedge B_2) \vee (B_1 \wedge B_3) & (\wedge \text{ distribuisce su } \vee) \\ \neg(B_1 \vee B_2) & \text{sse} & \neg B_1 \wedge \neg B_2 & (\text{De Morgan } \vee) \\ \neg(B_1 \wedge B_2) & \text{sse} & \neg B_1 \vee \neg B_2 & (\text{De Morgan } \wedge) \end{array}$$

e applicare opportunamente tali regole. □

Osservazione 3.5.7. Data una espressione booleana B con n letterali, la sua forma normale può avere un numero di letterali *esponenziale* in n. Ad esempio

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) = (x_1 \vee (x_2 \wedge y_2)) \wedge (y_1 \vee (x_2 \wedge y_2)) = \\ (x_1 \vee x_2) \wedge (x_1 \vee y_2) \wedge (y_1 \vee x_2) \wedge (y_1 \vee y_2).$$

Il lettore provi a mettere in forma normale congiuntiva l'espressione

$$(x_1 \wedge y_1 \wedge z_1) \vee (x_2 \wedge y_2 \wedge z_2) \vee (x_3 \wedge y_3 \wedge z_3).$$

FINE RICHIAMI

Possiamo adesso elencare alcuni problemi paradigmatici per la classe $\mathcal{NP}$ e presentare delle funzioni di riduzione tra loro; poi faremo lo stesso per $\mathcal{P}$. Come già accennato, sono problemi ben noti in calcolo proposizionale e in teoria dei grafi.

Nel resto del capitolo, ci limiteremo a considerare solo espressioni booleane in forma normale congiuntiva. Inoltre rappresenteremo un grafo o come si fa di solito con un disegno, oppure come una coppia $(N, A \subseteq N \times N)$, dove con $i \in N$ $(1 \leqslant i \leqslant \#N = n)$ indicheremo i nodi e con la coppia (ordinata se il grafo è orientato) $(i, j) \in A$ l'arco dal nodo i al nodo j.

Problema SAT Il problema SAT, o di *soddisfacibilità*, consiste nel decidere se, data un'espressione booleana B esiste un assegnamento $\mathcal{V}$ tale che $\mathcal{V} \models B$ (a volte si dice semplicemente B è soddisfacibile).

Chiaramente SAT appartiene a $\mathcal{NP}$: basta scegliere a caso un assegnamento per le variabili di B e lasciar scegliere al non-determinismo la soluzione buona, se c'è, ovvero uno di quelli che rendono vera l'espressione; vedremo piú avanti, ma non è difficile farlo, che la *certificazione* ovvero che l'eventuale controllo che l'assegnamento proposto sia davvero una soluzione richiede tempo deterministico polinomiale: un algoritmo logaritmico in spazio scorre i congiunti uno alla volta e si tiene sul nastro di lavoro il risultato della sua valutazione.

Problema HAM Il problema HAM consiste nel decidere se in un grafo (orientato) c'è un cammino, detto *hamiltoniano*, che tocca tutti i nodi una e una sola volta.[22]

Esempio 3.5.8. Si consideri la figura 3.3. Per il grafo rappresentato nella parte (**a**) il problema HAM è risolto con successo. Infatti, il cammino $(4,1)(1,2)(2,5)(5,6)(6,3)$, o semplicemente $(4,1,2,5,6,3)$, è hamiltoniano. Se consideriamo invece il grafo nella parte (**b**), il problema HAM ha soluzione negativa, perché esso non ha alcun cammino hamiltoniano.

(a) **(b)**

Figura 3.3: *Il grafo nella parte* (**a**) *ha un cammino hamiltoniano, l'altro no.*

Finalmente vediamo che HAM si può ridurre in spazio logaritmico a SAT, cioè SAT è un problema almeno tanto difficile quanto lo è HAM, visto che la riduzione usata è efficiente.

Proprietà 3.5.9. $HAM \leqslant_{\text{logspace}} SAT$

Dimostrazione. Dato G, dobbiamo costruire $f \in$ LOGSPACE tale che G ha un cammino hamiltoniano se e solamente se $f(G)$ è soddisfacibile. Prima introdurremo un'espressione booleana in forma congiuntiva che "rappresenta" i cammini hamiltoniani (non nella forma piú economica: ci sono alcune ridondanze, infatti); poi facciamo vedere che questa è soddisfacibile se e solo se G ha un tale cammino; infine delineamo una macchina di Turing che costruisce a partire da G proprio la espressione booleana e che usa uno spazio di lavoro logaritmico rispetto al numero dei nodi di G.

Sotto l'ipotesi che G abbia n nodi, $f(G)$ ha n^2 variabili booleane $x_{i,j}$ con $1 \leqslant i, j \leqslant n$ (si noti che la coppia (i,j) è sufficiente a individuare la variabile $x_{i,j} \in X$). Ciascuna variabile è usata per "rappresentare" che "il nodo j è l'i-esimo di un cammino (hamiltoniano)." Poiché $f(G)$ sarà in forma congiuntiva,

[22]Se si volesse considerare il problema del ciclo hamiltoniano basterebbe richiedere la presenza di un arco dall'ultimo nodo del cammino hamiltoniano al primo, cosa che non avviene nell'esempio 3.5.8; la presenza di tale arco è necessaria anche nella dimostrazione della proprietà che segue: bisognerà allora introdurre nuovi congiunti a tale scopo.

basta costruire i congiunti che la compongono. Esprimiamo con ciascuna delle seguenti formule il fatto che la f è una permutazione sull'intervallo $[1..n]$ (i punti 1 e 2 dicono che f è davvero una funzione e che è definita su ogni elemento di $[1..n]$, il punto 3 che è surgettiva e il 4 che è iniettiva). Infine scriveremo un'espressione che risulta vera se e solamente se una sequenza di nodi (ovvero di numeri $1 \leq j \leq n$) rappresenta un cammino in G (punto 5).

1. lo stesso nodo j non può apparire in due posizioni diverse nello stesso cammino, cioè $\neg(x_{i,j} \wedge x_{k,j})$, ovvero applicando la legge di De Morgan:
$$(\neg x_{i,j} \vee \neg x_{k,j}) \;\; i \neq k$$

2. ogni nodo j deve apparire in un cammino
$$(x_{1,j} \vee x_{2,j} \vee \ldots \vee x_{n,j}), \;\; 1 \leq j \leq n$$

3. qualche nodo deve essere l'i-esimo di un cammino:
$$(x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,n}) \;\; 1 \leq i \leq n$$

4. due nodi non possono essere contemporaneamente l'i-esimo nello stesso cammino, cioé $\neg(x_{i,j} \wedge x_{i,k})$ e ancora De Morgan:
$$(\neg x_{i,j} \vee \neg x_{i,k}) \;\; j \neq k$$

5. Se (i, j) non è un arco di G, i e j non possono apparire in sequenza in un cammino (hamiltoniano o no):
$$(\neg x_{k,i} \vee \neg x_{k+1,j}) \;\; \forall k.\, 1 \leqslant k \leqslant n-1 \text{ e } \forall (i,j) \notin A$$

Adesso mostriamo che $f(G)$, la congiunzione delle formule ottenute in accordo con i punti 1-5, ha un assegnamento $\mathcal{V}$ tale che lo soddisfa se e solamente se G ha un cammino hamiltoniano.

Prima vediamo frettolosamente che G ha un cammino hamiltoniano se $\mathcal{V} \models f(G)$. In questo caso, per ogni j esiste unico i tale che $\mathcal{V}(x_{ij}) = tt$ altrimenti le clausole definite nei punti 1 e 2 non potrebbero essere soddisfatte entrambe (p.e., $\mathcal{V}(x_{1,1}) = tt$ e $\mathcal{V}(x_{2,1}) = tt$ fan sí che $(\neg x_{1,1} \vee \neg x_{2,1}) = ff$). Allo stesso modo per ogni i esiste unico j tale che $\mathcal{V}(x_{i,j}) = tt$ (p.e., $\mathcal{V}(x_{1,1}) = tt$ e $\mathcal{V}(x_{1,2}) = tt$ fan sí che $(\neg x_{1,1} \vee \neg x_{1,2}) = ff$).
Quindi $\mathcal{V}$ rappresenta una permutazione $\Pi(1) \ldots \Pi(n)$ dei nodi di G e le clausole definite nel punto 5 garantiscono che la permutazione è effettivamente un cammino. Del resto un cammino hamiltoniano non è altro che una permutazione dei nodi del grafo, a due a due connessi da un arco. Quindi l'espressione booleana $f(G)$ rappresenta un cammino hamiltoniano se è soddisfatta, il che era la nostra ipotesi.

Adesso sia $H = (\Pi(1), \ldots, \Pi(n))$ un cammino hamiltoniano. Allora è immediato verificare che

$$\mathcal{V}(x_{i,j}) = \begin{cases} tt & \text{se } \Pi(j) = i \\ ff & \text{se } \Pi(j) \neq i \end{cases}$$

soddisfa $f(G)$, perché il nodo j è l'i-esimo di un cammino hamiltoniano.

Verifichiamo adesso che $f \in$ LOGSPACE, costruendo una MdT M di tipo I/O il cui alfabeto contiene l'insieme $\{tt, ff, \neg, \wedge, \vee, (,)\} \cup \{0, 1\}$, con l'obbiettivo di rappresentare in binario gli indici delle variabili $x_{i,j}$, quindi le variabili stesse, con $1 \leq i, j \leq n$. Descriviamo i passi di M raggruppandoli cosí:

i) M scrive il numero n dei nodi di G in binario su un nastro di lavoro;

ii) poi M scrive sul nastro di output le clausole definite nei punti da 1 a 4, che dipendono *solo* dal numero n dei nodi; per farlo si serve di 3 contatori che contengono i, j, k, rappresentati anch'essi in binario, i quali scorrono da 1 a n. Quindi quattro nastri di lavoro sono sufficienti — anzi, ne basterebbero tre perché i nastri possono venir inizializzati ad n per poi procedere a ritroso scalando il loro valore fino a 0.

iii) infine M genera le clausole $(\neg x_{k,i} \vee \neg x_{k+1,j})$ definite nel punto 5 una alla volta e le scrive sul nastro di output se $(i, j) \notin A$ (si ricordi che la descrizione degli archi è memorizzata sul nastro di input).

Tutto quello che ci serve sono quattro (o meglio tre) copie di n, il che richiede $\mathcal{O}(\log n)$ bit perché gli indici sono in binario. Allora $f \in$ LOGSPACE perché lo spazio necessario è la somma delle caselle visitate sui nastri di lavoro. □

Il risultato netto della funzione f costruita nella dimostrazione precedente è che HAM è stato "tradotto" in SAT — il risultato sorprendente è che il linguaggio logico del calcolo proposizionale, sia pur povero, è abbastanza potente da rappresentare un problema "difficile" come HAM e in un formalismo completamente diverso.

Vediamo adesso un altro problema e un'altra riduzione.

Problema CRICCA Il problema CRICCA consiste nel decidere se in un dato grafo (non orientato) $G = (N, A)$ esiste $C \subseteq N$, detto cricca (di grado k, funzione del numero dei nodi), tale che $\forall i, j \in C$, con $i \neq j$, l'arco $(i, j) \in A$.

Esempio 3.5.10. Dato il grafo nella figura 3.4, l'insieme $\{1, 2, 3\}$ è una cricca di grado 3, cosí come $\{1, 2, 4\}$; non vi sono invece cricche di grado 4, né ovviamente di grado maggiore a 4.

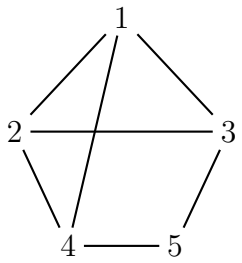

Figura 3.4: *Un grafo che ha due cricche di grado 3,* $\{1,2,3\}$ *e* $\{1,2,4\}$.

Ora vediamo che SAT si può ridurre in spazio logaritmico a CRICCA, cioè CRICCA è un problema almeno tanto difficile quanto lo è SAT.

Proprietà 3.5.11. $SAT \leqslant_{\text{logspace}} CRICCA$

Dimostrazione. Diamo solo l'idea della costruzione. Data un'espressione booleana $B = \bigwedge_{1 \leqslant k \leqslant n} C_k$, costruisci il grafo $f(B) = (N, A)$ così:

i) N è l'insieme delle *occorrenze* dei letterali in B;

ii) A è l'insieme $\{(i,j) \mid i \in C_k \Rightarrow (j \notin C_k \text{ e } i \neq \neg j)\}$

Essendo l'espressione B in forma normale congiuntiva, essa è soddisfacibile se e solamente se c'è almeno un letterale vero in ogni suo congiunto. Quindi, B è soddisfacibile se e solamente se $f(B)$ ha una cricca di ordine pari al numero di congiunti: basta attribuire valore tt ai letterali corrispondenti ai nodi della cricca. La costruzione infatti garantisce che un nodo, cioè un letterale, non può essere connesso da un arco al nodo originato dallo stesso letterale negato, né può esserlo a un nodo corrispondente a un letterale che compare nello stesso congiunto.
La riduzione è logaritmica in spazio perché basta mantenere sui nastri di lavoro due indici, rappresentati in binario, che scorrono i letterali. □

Esempio 3.5.12. Per esemplificare la dimostrazione della proprietà precedente, si consideri l'espressione booleana $(x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee \neg z)$ che dà origine al grafo rappresentato in figura 3.5. Esso ha tre archi uscenti e tre entranti in ciascun nodo. Una cricca di grado tre è l'insieme di nodi $\{\neg x, \neg y, z\}$ che corrisponde all'assegnamento $\mathcal{V}(x) = ff, \mathcal{V}(y) = ff, \mathcal{V}(z) = tt$.

In questo esempio tutti i letterali sono diversi, ma se volessimo aggiungere il congiunto $x \vee z$, la costruzione originerebbe due nuovi nodi, etichettati come quelli all'estrema destra e all'estrema sinistra nella figura.

Ci sono ancora un paio di problemi interessanti per gli sviluppi futuri. Per descriverli useremo una rappresentazione speciale delle funzioni booleane (totali) $f : \{ff, tt\}^n \to \{ff, tt\}$.

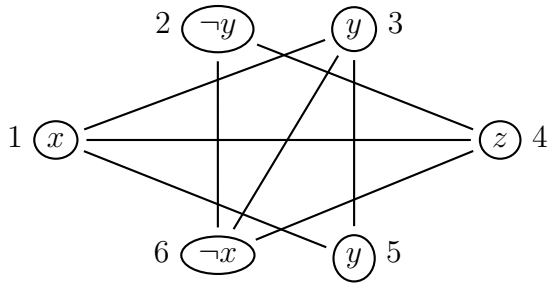

Figura 3.5: *Un grafo per l'espressione* $(x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee y)$*, che è soddisfacibile con la valutazione* $x, y \mapsto ff, z \mapsto tt$*, corrispondente alla cricca* $\{2, 4, 6\}$ *di grado 3, generata dall'insieme di letterali* $\{\neg y, z, \neg x\}$*. Si noti che i nodi 3 e 5 corrispondono a due distinte occorrenze del letterale* y*.*

Innanzitutto, dovrebbe già essere noto che tutte e sole le funzioni booleane sono rappresentabili da espressioni booleane (cosa non banale, ma ben dimostrata); per i nostri scopi, basta dire che l'espressione booleana con n variabili B rappresenta $f(x_1, \ldots, x_n)$ quando, dato l'assegnamento $\mathcal{V}(x_i) = t_i \in \{ff, tt\}$

$$f(t_1, \ldots, t_n) = tt \text{ se e solamente se } \mathcal{V} \models B.$$

Introduciamo adesso i circuiti booleani, che sono un altro modo ancora di rappresentare, o forse di realizzare le espressioni booleane e, per quanto appena detto, anche le funzioni booleane (di nuovo ci appoggiamo a cose note senza dimostrarle).

Definizione 3.5.13 (Circuito Booleano)**.** Un *circuito booleano* è un grafo diretto aciclico (N, A), i cui nodi $1, \ldots, n$ sono detti *porte*, e i cui archi sono al solito rappresentati come coppie ordinate: un arco orientato da i a j è rappresentato da (i, j).
Le porte hanno 0, 1 o 2 ingressi e sono di sorta $s(i) \in \{tt, ff, \neg, \vee, \wedge\} \cup X$, dove X è l'insieme delle *variabili*.
Gli *ingressi* i del circuito sono le porte di sorta $s(i) \in \{tt, ff\} \cup X$, e non hanno ingressi; *l'uscita* del circuito è, per convenzione, la porta n, senza uscite;[23] tutte le altre porte hanno una uscita e quando $s(i) = \neg$, la porta i ha un solo ingresso e quando $s(i) \in \{\vee, \wedge\}$ allora la porta i ha due ingressi.

Come visto prima, per rappresentare le funzioni booleane ci serve un assegnamento $\mathcal{V}$ di valori di verità alle variabili di un circuito C, che postuliamo essere buono, cioè tale da legare ogni variabile di C a un valore in $\{tt, ff\}$.

[23]La restrizione di avere una singola uscita si può facilmente rimuovere, permettendo cosí di avere circuiti con molte uscite.

Allora il valore di verità $[\![i]\!]_{\mathcal{V}}$ calcolato dalla porta i è definito induttivamente dalle seguenti clausole:

$$\begin{array}{lll}
[\![i]\!]_{\mathcal{V}} & = tt & \text{se } s(i) = tt \\
& = ff & \text{se } s(i) = ff \\
& = \mathcal{V}(x) & \text{se } s(i) = x \\
& = \text{not } [\![j]\!]_{\mathcal{V}} & \text{se } s(i) = \neg \text{ e } (j,i) \in A \\
& = [\![j]\!]_{\mathcal{V}} \text{ or } [\![k]\!]_{\mathcal{V}} & \text{se } s(i) = \vee \text{ e } (j,i),(k,i) \in A \\
& = [\![j]\!]_{\mathcal{V}} \text{ and } [\![k]\!]_{\mathcal{V}} & \text{se } s(i) = \wedge \text{ e } (j,i),(k,i) \in A
\end{array}$$

Infine il valore del circuito è $\mathcal{V}(C) = [\![n]\!]_{\mathcal{V}}$, quello della porta d'uscita.

Esempio 3.5.14. C'è un'immediata rappresentazione grafica di un circuito, che evidenzia quanto detto dianzi, cioè che esso sia interpretabile come un ordinamento parziale, i cui elementi sono le porte e l'ordinamento rappresenta il fluire del segnale nel circuito: le porte che appaiono piú in basso sono da considerare minori di quelle che appaiono piú in alto. Il grafo disegnato nella parte (**a**) della figura 3.6 rappresenta la seguente espressione booleana:

$$(x \vee (x \wedge y)) \vee ((x \wedge y) \wedge \neg(y \vee z)).$$

Nella parte (**b**), è rappresentato un circuito equivalente, nel senso che realizza la stessa espressione booleana e che si può ricavare dall'altro ottimizzando l'uso delle porte: nella figura della parte (**a**) vi sono due copie del circuito corrispondente all'espressione $(x \wedge y)$, nella parte (**b**) ce n'è una sola, condivisa dalle porte che la dominano nell'ordinamento parziale contravvenendo all'ipotesi che tutte le porte, ad eccezione di quelle di inrgesso, hanno una sola uscita (per esempio, nel circuito di sinistra le porte di sorta variabile sono condivise).

Possiamo ora descrivere un altro problema, e poi una sua variante, che verranno usati per caratterizzare le classi $\mathcal{NP}$ e $\mathcal{P}$.

Problema CIRCUIT SAT Il problema CIRCUIT SAT, o della *soddisfacibilità dei circuiti*, consiste nel decidere se esiste un assegnamento $\mathcal{V}$ tale che $\mathcal{V}(C) = tt$.

Un modo immediato per risolvere questo problema è di sostituire alle variabili $x, y, z, \ldots$ *tutte le possibili* combinazioni di tt, ff e di *controllare* una a

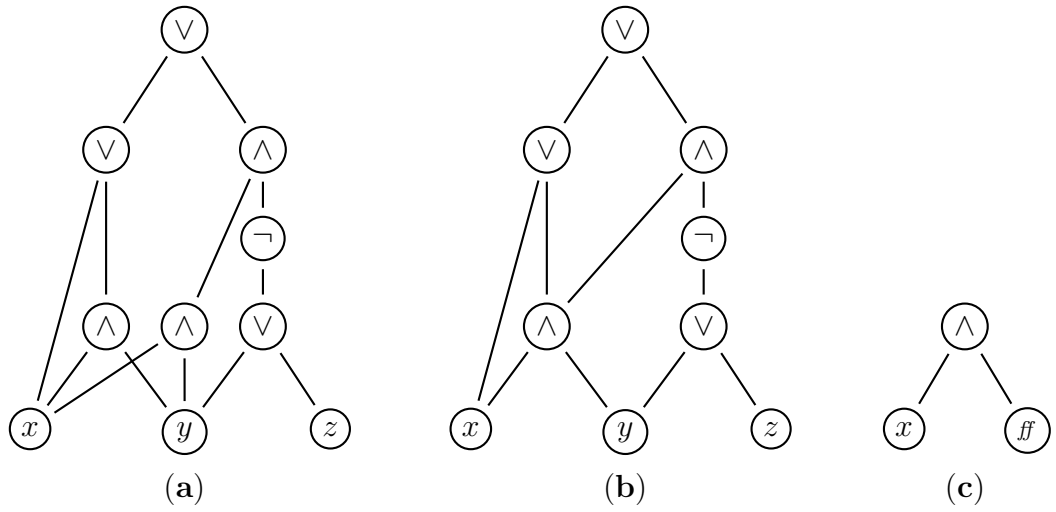

Figura 3.6: *Due circuiti booleani equivalenti in (**a**) e (**b**) soddisfacibili, e uno non soddisfacibile in (**c**).*

una le combinazioni per verificare se sono una soluzione. Questa è ancora una volta una ricerca esaustiva per forza bruta. La versione non deterministica mostra che CIRCUIT SAT $\in \mathcal{NP}$ e prevede due passi: nel primo si sceglie in tempo polinomiale non deterministico l'assegnamento "giusto" tra i 2^n possibili, se n sono le variabili; nel secondo, *polinomiale* nel numero delle porte del circuito, si *certifica* il risultato (v. subito sotto per una giustificazione intuitiva). Questo secondo passo può essere opportunamente visto come un problema di decisione, visto che risulterà essere particolarmente interessante.

Problema CIRCUIT VALUE Il problema CIRCUIT VALUE consiste nel calcolare il valore di un circuito *senza variabili*, ovvero in cui gli ingressi sono porte di sorta $s(i) \in \{tt, ff\}$ solamente.

È immediato vedere che questo problema appartiene a $\mathcal{P}$: basta "arrampicarsi" sull'ordinamento parziale a partire dalle porte di ingresso, valutando i valori di uscita delle n porte livello per livello; non ci si faccia confondere dal fatto che il numero totale delle porte può essere esponenziale rispetto la profondità del circuito, cioè il livello della porta di uscita: la taglia del circuito è data infatti proprio dal numero n delle sue porte.

Per calcolare il valore di un circuito è sufficiente allora al passo i-esimo memorizzare su un nastro di lavoro i valori delle porte di quel livello, usando i valori calcolati ai passi precedenti, ovvero quelli delle porte dei livelli inferiori, che sono stati via via memorizzati sul nastro di lavoro; i valori delle porte di ingresso stanno sul nastro di ingresso.

Esempio 3.5.15. Si consideri il circuito che si ottiene da quello rappresentato nella parte (**a**) della figura 3.6 sostituendo alle variabili x, y e z i valori tt, ff e tt nell'ordine. Si verifichi che esso produce come uscita tt, cosí come del resto quello della parte (**b**) della figura 3.6.

3.5.3 Problemi completi per $\mathcal{P}$ ed $\mathcal{NP}$

Iniziamo con alcune proprietà che ci saranno utili per presentare alcuni problemi e poi dimostrarli completi per $\mathcal{P}$ ed $\mathcal{NP}$.

Il seguente teorema è banale e illustra un tipo speciale di riduzione: quella per *generalizzazione*. Ogni caso particolare di un problema si riduce al problema stesso nella sua piena generalità attraverso la funzione identità.

Proprietà 3.5.16. *CIRCUIT VALUE* $\leqslant_{logspace}$ *CIRCUIT SAT.*

Correliamo adesso CIRCUIT SAT a un problema appena piú complesso e facciamo vedere che quando due problemi sono molto simili è facile trovare una riduzione tra essi (e chi l'avrebbe mai detto?); l'esempio che consideriamo è la riduzione di CIRCUIT SAT a SAT.

Proprietà 3.5.17.
CIRCUIT SAT $\leqslant_{\text{logspace}}$ *SAT*

Dimostrazione. Dato il circuito $C = (N, A)$ con variabili in X, definiamo una riduzione $f \in$ LOGSPACE tale che l'espressione booleana $f(C)$ sia soddisfacibile se e solamente se C lo è, in simboli tale che se $\exists \mathcal{V}.\ [\![C]\!]_{\mathcal{V}} = tt$ allora e solamente allora $\exists \mathcal{V}'$ tale che $\forall x \in X.\ \mathcal{V}'(x) = \mathcal{V}(x)$ e $\mathcal{V}' \models f(C)$. Il gioco è facile, perchè sia i circuiti che le espressioni booleane rappresentano funzioni booleane, ma non è affatto banale — l'idea è di Tseitin, S. Pietroburgo 1966.

i) Le variabili x di $f(C)$ sono quelle di C unite a un nuovo insieme che contiene una nuova variabile per ogni porta di C; per semplicità, diamo a queste nuove variabili lo stesso nome che hanno le porte, fidando nel fatto che il contesto ci dice quando sono le une e quando le altre;

ii) per ogni porta g di C costruiamo i congiunti di $f(C)$ cosí:
   - se g è la porta di uscita, allora genera il congiunto g; ciò garantisce che l'assegnamento $\mathcal{V}'$ manderà g in tt altrimenti l'intera formula $f(C)$ risulterebbe falsa;
   - se $s(g) = tt$ (oppure ff) allora genera g (oppure $\neg g$), e l'equivalenza è ovvia — si noti che se il circuito avesse solo una porta g di sorta ff la sua immagine sotto f sarebbe $g \wedge \neg g$ che non può essere soddisfacibile (il primo congiunto viene dalla prima condizione, il secondo da quella che stiamo considerando);

- se $s(g) = x \in X$ e quindi la porta g origina la variabile x, allora genera $(\neg g \vee x) \wedge (g \vee \neg x)$, ovvero $g \iff x$ (cioè g *se e solamente se* x, ricordando la definizione di $\iff$ in termini di $\neg, \vee$ e $\wedge$). Quindi entrambi i congiunti sono soddisfatti da $\mathcal{V}'(g) = \mathcal{V}'(x) = \mathcal{V}(x)$;
- se $s(g) = \neg$ e $(h, g) \in A$ allora genera $(\neg g \vee \neg h) \wedge (g \vee h)$, ovvero $g \iff \neg h$, e l'equivalenza tra le due formulazioni è ancora una volta ovvia;
- se $s(g) = \vee$ e $(h, g), (k, g) \in A$ allora genera $(\neg h \vee g) \wedge (\neg k \vee g) \wedge (h \vee k \vee \neg g)$, ovvero $g \iff (h \vee k)$, e l'equivalenza tra le due formulazioni è immediata;
- se $s(g) = \wedge$ e $(h, g), (k, g) \in A$ allora genera $(\neg g \vee h) \wedge (\neg g \vee k) \wedge (\neg h \vee \neg k \vee g)$, ovvero $g \iff (h \wedge k)$, e l'equivalenza tra le due formulazioni è immediata ancora una volta.

La dimostrazione che trasformazione richiede spazio logaritmico si basa sulle stesse osservazioni fatte per mostrare che HAM si riduce a SAT. (Si veda anche l'esercizio 1, parte sulla complessità.) □

Esempio 3.5.18. Per vedere come funziona la riduzione delineata nella dimostrazione precedente, si consideri il semplicissimo circuito C, illustrato in Figura 3.6, parte (**c**) e composto da tre porte g, h e k con g porta di uscita e $s(g) = \wedge$ e h, k porte di ingresso di sorta rispettivamente x e *ff*. Naturalmente C sempre ha valore *ff* e quindi ci aspettiamo un'espressione $f(C)$ non soddisfacibile. La porta di uscita genera i seguenti quattro congiunti, il primo per la prima condizione, gli altri tre per l'ultima: $g \wedge (g \vee \neg h \vee \neg k) \wedge (\neg g \vee h) \wedge (\neg g \vee k)$. La porta h genera i seguenti due congiunti: $(\neg h \vee x) \wedge (h \vee \neg x)$. Infine la porta k genera $\neg k$.

È immediato verificare che la congiunzione di tutte queste sette disgiunzioni non è soddisfacibile perché a g deve essere assegnato il valore *tt* e quindi $(\neg g \vee k)$ valuta a *tt* se e solamente se a k viene assegnato il valore *tt*, ma in questo caso l'ultimo congiunto $\neg k$ varrebbe *ff*.

Solo per ricordare che la classe LOGSPACE è chiusa per composizione e transitività (vedi il teorema 3.5.2) possiamo enunciare anche le seguenti proprietà.

Corollario 3.5.19.

$$\textit{CIRCUIT VALUE} \leqslant_{\text{logspace}} \textit{SAT}$$
$$\textit{CIRCUIT VALUE} \leqslant_{\text{logspace}} \textit{CRICCA}$$

Possiamo passare adesso a caratterizzare $\mathcal{P}$ e $\mathcal{NP}$ mediante uno o piú problemi completi. Ricordiamo anche che, data una classe di funzioni F tale che $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$ (con $\mathcal{D} \subseteq \mathcal{E}$), se un problema A è completo per $\mathcal{E}$ e $A \in \mathcal{D}$ allora $\mathcal{E} = \mathcal{D}$. Quindi studiare i problemi completi per una classe significa davvero caratterizzarla. Prima abbiamo po' di abbreviazioni e di ipotesi di lavoro, per semplificare le dimostrazioni e standardizzare la nozione di computazione. La piú importante riguarda un modo di rappresentare le computazioni delle macchine deterministiche come una successione di configurazioni opportunamente organizzate in una matrice. La definizione copre il caso di macchine a 1 nastro che operano in tempo deterministico oplinomiale, ma se la macchina fosse a k nastri la potremmo sempre ridurre in tempo polinomiale, grazie al teorema 3.2.6, a una macchina a 1 nastro, riportandoci al caso precedente.

Definizione 3.5.20 (Tabella di computazione)**.** Intuitivamente, la *tabella di computazione* T_M di una macchina di Turing M a 1 nastro deterministica, o semplicemente T quando non vi sia ambiguità, è una matrice quadrata il cui indice di riga i rappresenta l'i-esimo passo di computazione e il cui indice di colonna j rappresenta la j-esima posizione sul nastro. Di conseguenza, la riga i-esima rappresenta la configurazione di M dopo il passo $i-1$ e l'elemento $T(i,j)$ contiene il simbolo contenuto nella cella j-esima del nastro dopo $i-1$ passi di computazione.

Se M decide il problema I in tempo polinomiale deterministico $|x|^k$, la sua tabella di computazione (o semplicemente la sua computazione) su x ha al massimo $|x|^k$ righe e altrettante colonne.[24]

Per semplificarci la vita, facciamo alcune ulteriori standardizzazioni. Data una macchina M, conveniamo che:

1. il valore di k sia tale per cui la macchina M si arresta prima di $|x|^k - 2$ passi — quindi k deve essere abbastanza grande per garantirlo (v. anche il punto 5.) e ciò è facile quando $|x| \geqslant 2$; se invece $|x| \leqslant 1$ allora agisci localmente su questo caso particolare, ovvero fai come se non esistesse. (A dire il vero, sarebbe sufficiente considerare tabelle di dimensione $|x|^k + 2$, ma tale valore risulterebbe noioso da scrivere.)

2. Il nastro è riempito nelle posizioni non significative a destra con tanti # quanti ne servono per arrivare alla casella in colonna $|x|^k$: la testina di M non oltrepasserà mai tale posizione — in una macchina che decide un problema, il tempo limita lo spazio!

[24] Con queste dimensioni, una tabella di computazione ha ovviamente abbastanza righe, ed ha anche abbastanza colonne, perché non si può usare piú spazio che tempo.

3. Arricchiamo l'alfabeto di M in modo che la casella $T(i,j)$ contenga il nuovo simbolo σ_q, per registrare che nella configurazione i-ma la testina viene a trovarsi sulla j-esima casella, il simbolo letto è σ e lo stato corrente è q — per far ciò basta prendere $\Sigma \times Q$ nuovi simboli, che sono la congiunzione del simbolo σ e dello stato q.

4. La configurazione iniziale ha la testina sul carattere immediatamente a destra del simbolo di inizio nastro, cioè M inizia a calcolare da $\triangleright \underline{\sigma}_{q_0} w$ e non da $\trianglerighteq_{q_0} \sigma w$. Inoltre trasformiamo la funzione di transizione di M in modo che la testina non si posizioni mai sul simbolo di inizio stringa $\triangleright$ (quando ciò accadesse, basterebbe condensare in un solo passo i due compiuti dalla macchina M: lo spostamento a sinistra sulla casella contenente il simbolo $\triangleright$ e il successivo spostamento a destra). Questa ipotesi fa sí che $\triangleright$ si comporti come un *vero* respingente! c'è però l'eccezione riportata nel punto seguente.

5. Se $T(i,j) \in \{\sigma_{SI}, \sigma_{NO}\}$ allora "sposta" il cursore fino alla seconda colonna (con al massimo $\mathcal{O}(|x|^k)$ passi, introducendo uno stato ausiliario di finto arresto). Cioè lo stato di accettazione, se è raggiunto, sia sempre in $T(\ell, 2)$, per qualche $\ell \leq |x|^k$. Si noti che anche queste manovre possono influire sul valore di k, per esempio nel caso in cui M si arresti proprio all'estremità destra del nastro. Inoltre, come eccezione al punto precedente, si ammette che la testina si sposti sopra il simbolo $\triangleright$ quando lo stato sia q_{SI} (abbreviazione di $\sigma_{q_{SI}}$), con il vincolo che non debba *mai* toccare il simbolo $\triangleright$ piú a sinistra, che rappresenta l'inizio del nastro (v. ipotesi 4).

6. Se σ_{SI} oppure σ_{NO} appaiono nella riga $p < |x|^k$ e nella seconda colonna, allora tutte le righe di indice q, $p \leq q \leq |x|^k$, sono uguali alla p-esima.

Per finire, abbiamo l'ovvia condizione di terminazione con successo: M accetta x se e solamente se esiste i tale che $T(i,2) = \sigma_{SI} (= T(|x|^k, 2))$.

Esempio 3.5.21. Consideriamo la macchina di Turing M dell'esempio 1.2.6, che accetta la stringa palindroma in tempo n^2 e applichiamola a *abba*. In questo caso per il punto 1 della standardizzazione, la tabella di computazione ha $4^2 + 2 = 18$ righe e colonne. La disegnamo nella Figura 3.5.3, dove, a partire dalla settima colonna ci sono solo # e dalla riga 16 ci sono altre due righe uguali alla sedicesima; inoltre, abbiamo introdotto lo stato di accettazione ausiliario SI, lasciando al lettore il compito di apportare le necessarie modifiche alla macchina originale per ottenere il risultato descritto. Si noti che il passaggio dalla riga 8 alla 9 richiede alla macchina originale due passi, qui condensati in uno; lo stesso per il passaggio dalla riga 12 alla 13.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	$\triangleright$	a_{q_0}	b	b	a	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
2	$\triangleright$	$\triangleright$	b_{q_a}	b	a	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
3	$\triangleright$	$\triangleright$	b	b_{q_a}	a	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
4	$\triangleright$	$\triangleright$	b	b	a_{q_a}	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
5	$\triangleright$	$\triangleright$	b	b	a	$\#_{q_a}$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
6	$\triangleright$	$\triangleright$	b	b	$a_{q'_a}$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
7	$\triangleright$	$\triangleright$	b	b_{q_1}	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
8	$\triangleright$	$\triangleright$	b_{q_1}	b	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
9	$\triangleright$	$\triangleright$	b_{q_0}	b	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
10	$\triangleright$	$\triangleright$	$\triangleright$	b_{q_b}	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
11	$\triangleright$	$\triangleright$	$\triangleright$	b	$\#_{q_b}$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
12	$\triangleright$	$\triangleright$	$\triangleright$	$b_{q'_b}$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
13	$\triangleright$	$\triangleright$	$\triangleright$	$\#_{q_0}$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
14	$\triangleright$	$\triangleright$	$\triangleright$	$\#_{SI}$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
15	$\triangleright$	$\triangleright$	$\triangleright_{SI}$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
16	$\triangleright$	$\triangleright_{SI}$	$\triangleright$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
17	$\triangleright$	$\triangleright_{SI}$	$\triangleright$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$
18	$\triangleright$	$\triangleright_{SI}$	$\triangleright$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$	$\#$

Figura 3.7: *Il frammento iniziale della tabella di computazione della macchina di Turing dell'esempio 1.2.6.*

Se rimanessero dei dubbi su come costruire una macchina M' che obbedisce alle condizioni elencate sopra ed è equivalente a una macchina M che non le soddisfa, oltre a ricorrere alla tesi di Church-Turing, che tuttavia in sé non garantisce che M' impieghi tempo polinomiale, si può immaginare di procedere come segue. Si generino le configurazioni della computazione di $M(x)$, che sono in numero minore o uguale a $|x|^k$; poi si riempia per righe la tabella di computazione (che potrebbe avere rango $|x|^{k+1}$), ricopiando l'i-ma configurazione sull'i-ma riga avendo cura di corto-circuitare le configurazioni in cui il cursore si trova sul respingente (con le ovvie eccezioni), eventualmente riempiendo la riga con tanti $\#$ quanti servono; infine si ricopi la riga contenente la configurazione finale fino a completare la tabella. Il numero dei passi necessari a realizzare questa trasformazione è dell'ordine di $|x|^k + 2 \times (|x|^{k+1} \times |x|^{k+1})$: il primo addendo per il tempo speso da M, il secondo per leggere i simboli delle configurazioni e riempire tutte le caselle della tabella. Il tutto non ci fa uscire dalla classe $\mathcal{P}$ (né da LOGSPACE).

Facciamo adesso un'osservazione importante, che è alla base della dimostrazione dei due teoremi fondamentali di questo capitolo.

Osservazione 3.5.22. Sia M una macchina di Turing che decide I in $|x|^k$, e T la sua tabella di computazione su x. Abbiamo che

- la cella $T(1,2)$ contiene lo stato iniziale e il primo carattere di x;
  inoltre $\forall j.\ 2 \leq j \leq |x|+1$, la casella $T(1,j)$ contiene il $(j-1)$-esimo simbolo di x;
  infine $\forall j.\ |x|+2 \leq j \leq |x|^k$, la cella $T(1,j)$ contiene $\#$;

- $\forall i.\ T(i,1) = \triangleright$;

- $\forall i.\ T(i,|x|^k) = \#$ (si impiega meno spazio che tempo!)

Quindi la prima riga è completamente determinata dal dato di ingresso e la prima e l'ultima colonna sono fisse in *tutte* le tabelle di computazione.

La cosa importante adesso è capire come si determina $T(i,j)$ in tutti gli altri casi, una volta fissata la funzione di transizione δ; il suo valore *dipende solo da tre caselle*: quelle della riga precedente, nella stessa posizione o in quelle immediatamente ai suoi lati, cioè

il valore di $T(i,j)$ *è completamente determinato da*
quelli di $T(i-1,j-1), T(i-1,j), T(i-1,j+1)$

Per convincersene si esamini la figura 3.8 e si considerino i seguenti casi:

i) Se le tre caselle contengono $\sigma \in \Sigma$, cioè il cursore non si trova su alcune di esse, allora $T(i,j) = T(i-1,j)$;

ii) Se una di esse contiene σ_q, cioè è la casella su cui c'è il cursore, allora $T(i,j)$, che può essere diverso da $T(i-1,j)$ per effetto di uno spostamento e/o di una scrittura, è completamente determinato da $\delta(\sigma,q)$, e quindi da $T(i-1,j-1), T(i-1,j)$ e $T(i-1,j+1)$.

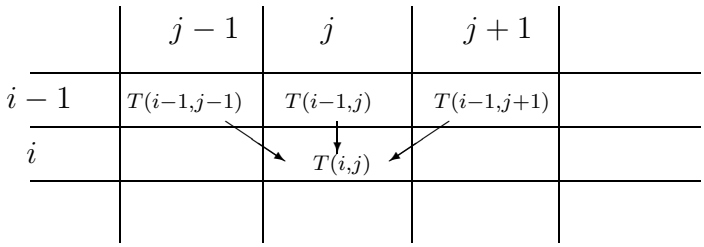

Figura 3.8: *Le tre caselle nella riga* $i-1$ *e nelle colonne* $j-1, j, j+1$ *(corrispondenti a quelle della configurazione precedente) determinano la casella* $T(i,j)$ *(corrispondente a quella della configurazione attuale).*

$\mathcal{P}$-completezza

Siamo pronti per presentare formalmente il primo problema $\mathcal{P}$-completo e a dimostrare che lo è davvero.

Teorema 3.5.23 (CIRCUIT VALUE è $\mathcal{P}$-completo)**.**
CIRCUIT VALUE è $\leqslant_{logspace}$-completo per $\mathcal{P}$.

Dimostrazione. Sappiamo già che CIRCUIT VALUE $\in \mathcal{P}$, quindi prendiamo un qualunque $I \in \mathcal{P}$ e facciamo vedere che c'è una riduzione $f \in LOGSPACE$ che lo trasforma in CIRCUIT VALUE; ovvero $x \in I$ se e solamente se $f(x)$ è un circuito *senza* variabili il cui valore è tt.

Sia M una macchina di Turing che decide I in $n^k - 2$, e T la sua tabella di computazione su x e inoltre sia Σ' l'alfabeto di M unito ai nuovi simboli $\sigma_q \in \Sigma \times (Q \cup \{\epsilon\})$ che si usano per costruire la tabella T.

Come primo passo della dimostrazione costruiamo un circuito a partire dalla tabella di computazione di M. Per far ciò codifichiamo ogni simbolo $\rho \in \Sigma'$ con una stringa di bit $(S_1 \dots S_m) \in \{tt, ff\}^m$, dove naturalmente $m = \lceil \log \#(\Sigma') \rceil$. Con questa rappresentazione, la tabella di computazione T risulta avere $|x|^k$ righe che sono stringhe di bit, ciascuna lunga $m \times |x|^k$. Possiamo allora rappresentare ciascun elemento della tabella con $S_{i,j,\ell}$, con $1 \leqslant i, j \leqslant |x|^k$ e $1 \leqslant \ell \leqslant m$.

Innanzitutto, $\forall i$ la m-upla $S_{i,1,1} \dots S_{i,1,m}$ codifica il simbolo $\triangleright$ e analogamente $\forall i$. $S_{i,|x|^k,1} \dots S_{i,|x|^k,m}$ codifica il simbolo $\#$.

Inoltre, poiché la funzione di transizione δ_M della macchina M è fissata, possiamo usare l'osservazione fatta appena sopra riguardo al valore della casella $T(i,j)$ nella tabella di computazione: in modo del tutto analogo il valore di $S_{i,j,\ell}$ dipende solo dal valore delle tre sequenze di m bit nella riga precedente con indice di colonna $j-1, j, j+1$, cioè dai $3m$ bit $S_{i-1,j-1,\ell'}$, $S_{i-1,j,\ell''}$, $S_{i-1,j+1,\ell'''}$ con $1 \leqslant \ell', \ell'', \ell''' \leqslant m$. Allora ci sono m funzioni booleane $F_1, \dots, F_m$, con $3m$ ingressi ciascuna, tali che $\forall i, j > 0$

$$\begin{aligned} S_{i,j,\ell} \;=\; F_\ell \;(&S_{i-1,j-1,1}, S_{i-1,j-1,2}, \dots, S_{i-1,j-1,m}, \\ &S_{i-j,j,1}, S_{i-1,j,2}, \dots, S_{i-1,j,m}, \\ &S_{i-1,j+1,1}, S_{i-1,j+1,2}, \dots, S_{i-1,j+1,m}) \end{aligned}$$

A costo di essere noiosi ricordiamo che ciascuna funzione F_ℓ dipende *solamente* dalla funzione di transizione δ_M.

Non è difficile adesso costruire, seguendo lo stesso schema, una funzione che "raggruppa" $F_1, \dots, F_m$ in una sola funzione che restituisce m valori a fronte degli stessi $3m$ ingressi. Per ogni funzione booleana, a uno o piú valori, esiste un circuito, con lo stesso numero di uscite, che la calcola (come

già accennato i circuiti booleani a m valori hanno come porte di uscita m porte, quelle senza archi uscenti). Per ora abbiamo detto come costruire il circuito booleano $\overline{C}$ con $3m$ ingressi e m uscite che codifica $T(i,j)$ dati $T(i-1,j-1), T(i-1,j)$ e $T(i-1,j+1)$; rappresentiamolo graficamente in figura 3.9, dove abbiamo decorato ciascuna copia di $\overline{C}$ con gli indici i e j per rendere chiaro il suo legame con la casella $T(i,j)$; si confronti questa con la figura 3.6, osservando che nelle nostre rappresentazioni le computazioni "procedono" verso il basso, i segnali "fluiscono" verso l'alto.
A questo punto facciamo un'osservazione molto importante: poiché il circuito $\overline{C}$ dipende *solo* dalla funzione di transizione δ_M di M, la sua taglia, comunque sia misurata, purché in modo "ragionevole," è *fissata* ed è *indipendente dall'input* x.

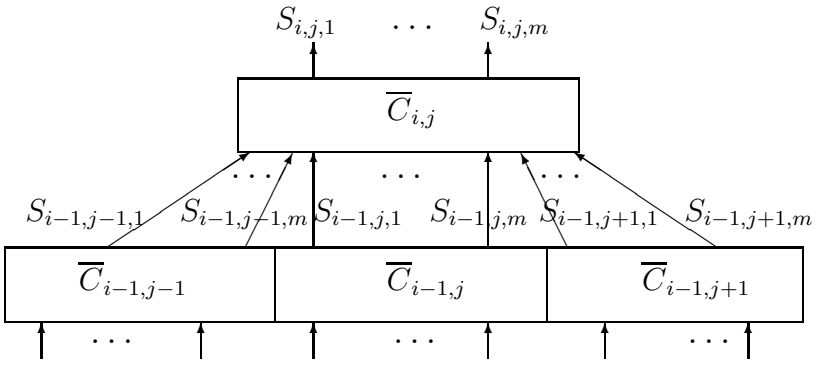

Figura 3.9: *Un circuito che calcola* $T(i,j)$ *dati* $T(i-1,j-1), T(i-1,j)$ *e* $T(i-1,j+1)$. *Ciascuno dei circuiti in basso ha* $3{\times}m$ *ingressi.*

Siamo pronti per definire la riduzione f da I a CIRCUIT VALUE. Sostanzialmente, si tratta di costruire un circuito C_I che sia la codifica della tabella di computazione T. Tale circuito C_I sarà composto da una copia di $\overline{C}$ per ogni $T(i,j)$, come rozzamente illustrato in figura 3.10. (Si noti che bastano $(|x|^k-1)\times(|x|^k-2)$ copie perchè, grazie alle ipotesi fatte sulla forma della tabella di computazione, abbiamo fissato una volta per tutte la prima riga ($\forall j \geqslant 2. T(1,j) = \sigma_{j-1}$ se il dato di ingresso è $\sigma_1, \ldots, \sigma_{|x|}$) e la prima e l'ultima colonna ($\forall i \geqslant 1. T(i,1) = \triangleright$ e $T(i,|x|^k) = \#$, perché la macchina si arresta prima di $|x|^k - 2$ passi). Quindi non c'è bisogno di costruire copie del circuito corrispondenti a tale riga e a tali colonne: basta prenderle dallo scaffale.) Le uscite di $\overline{C}_{i-1,j-1}, \overline{C}_{i-1,j}, \overline{C}_{i-1,j+1}$ sono gli ingressi di $\overline{C}_{ij}$. Gli ingressi di C_I sono quelli di $T(1,j), T(i,1), T(i,|x|^k)$, che sono determinati da x o sono fissi. Rappresentiamo adesso i simboli σ_{SI} e σ_{NO} corrispondenti all'accettazione o al rifiuto con stringhe composte rispettivamente da soli *tt* e da soli *ff*. Allora, l'uscita di C_I è una delle uscite di $\overline{C}_{|x|^k,2}$, poiché abbia-

mo convenuto che tali simboli appaiano sempre in posizione $T(|x|^k, 2)$ (cf. l'ultima ipotesi fatta sulla tabella T).

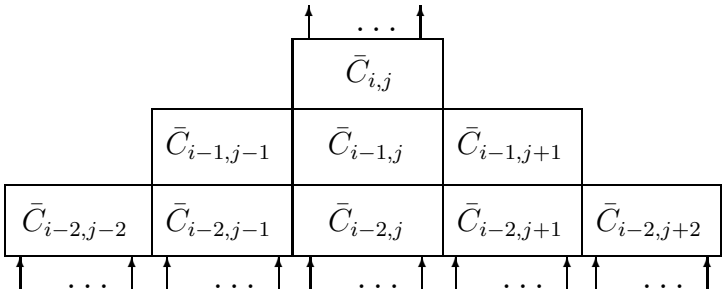

Figura 3.10: *Una parte del circuito C_I, composto da alcune copie di $\overline{C}$.*

Il prossimo passo consiste nel verificare che $C_I = f(x)$ ha valore tt se e solamente se $x \in I$. Prima di tutto ricordiamo che le uscite di $\overline{C}_{i,j}$ rappresentano in binario il valore della cella $T(i,j)$ della tabella della computazione di M su x. Delineiamo la dimostrazione che procede per induzione sul numero i dei passi della computazione (l'indice di riga di T).
Se $i = 1$ banale.
L'ipotesi induttiva è che il valore per $i-1$ sia ben calcolato (e dipenda solo dai tre precedenti), cioè che $\forall j$. $\overline{C}_{i-1,j}$ ha come uscite $R_1 \dots R_m$ se e solamente se $T(i-1,j) = \rho'$ e $\rho' \in \Sigma'$ è codificato proprio come $R_1 \dots R_m$. Dobbiamo dimostrare che $\overline{C}_{i,j}$ ha come uscite $S_1 \dots S_m$ se e solamente $T(i,j) = \rho$, la cui codifica è $S_1 \dots S_m$. Le uscite $S_1 \dots S_m$ sono state calcolate a partire da quelle di $\overline{C}_{i-1,j-1}$, $\overline{C}_{i-1,j}$ e $\overline{C}_{i-1,j+1}$ in pieno accordo con la funzione di transizione δ_M di M, esattamente allo stesso modo in cui il valore di $T(i,j)$ lo è stato in funzione dei valori di $T(i-1,j-1)$, $T(i-1,j)$ e $T(i-1,j+1)$. Allora basta applicare l'ipotesi induttiva.
A questo punto è immediato dedurre che $\overline{C}_{|x|^k,2} = tt \dots tt$ se e solamente se $T(|x|^k, 2) = \sigma_{SI}$, ovvero $x \in I$ se e solamente se $f(x) = tt \dots tt$ (il che significa che vale lo stesso per il caso $\overline{C}_{|x|^k,2} = ff \dots ff$ e σ_{NO}).

Vediamo infine che $f \in$ LOGSPACE, cioè che la riduzione può essere calcolata in spazio $\mathcal{O}(\log |x|)$, stimando lo spazio necessario ai nastri di lavoro. Per calcolare f dobbiamo costruire e connettere opportunamente tra loro

i) le porte di ingresso — facile, esamina x e conta fino a $|x|^k$, ricordandosi k in base 2 su un nastro di lavoro, scrivendo la codifica di x (e di # finché serve);

ii) gli elementi della prima e dell'ultima colonna che sono $2 \times |x|^k$ circuiti costanti;

iii) $(|x|^k - 1) \times (|x|^k - 2)$ copie del circuito $\overline{C}$ (che dipende, ricordiamolo, *solo* da M e ha costo *fisso*). A ciascuna copia associamo gli appropriati indici, per metterla in relazione con la casella di computazione corrispondente; tali indici sono tutti minori di $|x|^k$, e averli rappresenti in binario ci consente di manipolarli in spazio $\mathcal{O}(\log |x|)$. □

Ritorniamo adesso rapidamente sulla distinzione tra approcci hardware e software ai modelli di calcolo fatta nel capitolo 1.5. Per quelli hardware, si suggeriva di vedere ciascun algoritmo come una macchina, le cui dimensioni potevano addirittura crescere al crescere dei dati di ingresso. Tale osservazione trova giustificazione nel circuito C_I costruito nella dimostrazione precedente: al crescere del dato x cresce la tabella di computazione della macchina di Turing M e cosí crescono le *dimensioni* del circuito corrispondente, ma non cambiano i suoi componenti $\overline{C}$ — né cambia, per cosí dire, la sua forma.

Vediamo adesso una variante di CIRCUIT VALUE, chiamato MONOTONE CIRCUIT VALUE: il problema di verificare se un circuito chiuso e *senza* $\neg$ calcoli il valore *tt*. Si sa bene che le espressioni booleane e quindi i circuiti con soli $\vee, \wedge$ sono meno espressivi di quelli che hanno anche il $\neg$, tuttavia *la difficoltà nel valutare gli uni e gli altri è la stessa*! Infatti non è difficile costruire una riduzione facile da CIRCUIT VALUE a MONOTONE CIRCUIT VALUE.

Corollario 3.5.24. *MONOTONE CIRCUIT VALUE è $\mathcal{P}$-completo*

Dimostrazione. Facciamo vedere che CIRCUIT VALUE $\leqslant_{\text{logspace}}$ MONOTONE CIRCUIT VALUE, cioè trasformiamo un circuito qualunque (con ingressi assegnati) in uno monotono equivalente. Ciò vien fatto applicando le regole di De Morgan partendo dalle porte di uscita a scendere fino alle porte di ingresso. Le porte di ingresso *tt* rimpiazzano se necessario quelle *ff* e viceversa, e se ne aggiungono di nuove se servissero. Tutto questo si fa ovviamente in LOGSPACE: basta visitare una sola volta le porte, rappresentate come coppie (i, j), dove i e j sono indici di livello e di "colonna", rappresentati a loro volta in binario. □

Come esempio della trasformazione delineata nella dimostrazione appena vista si consideri il circuito senza variabili nella parte (**a**) della figura 3.11 che viene trasformato in quello privo di negazione raffigurato nella parte (**b**); si noti la duplicazione della porta di ingresso centrale.

Ecco un paio di problemi $\mathcal{P}$-completi, l'uno preso dalla teoria dei linguaggi formali e l'altro dalla logica; poiché appartengono allo stesso grado, vi sono trasformazioni sia dall'uno all'altro che da e in (MONOTONE) CIRCUIT VALUE.

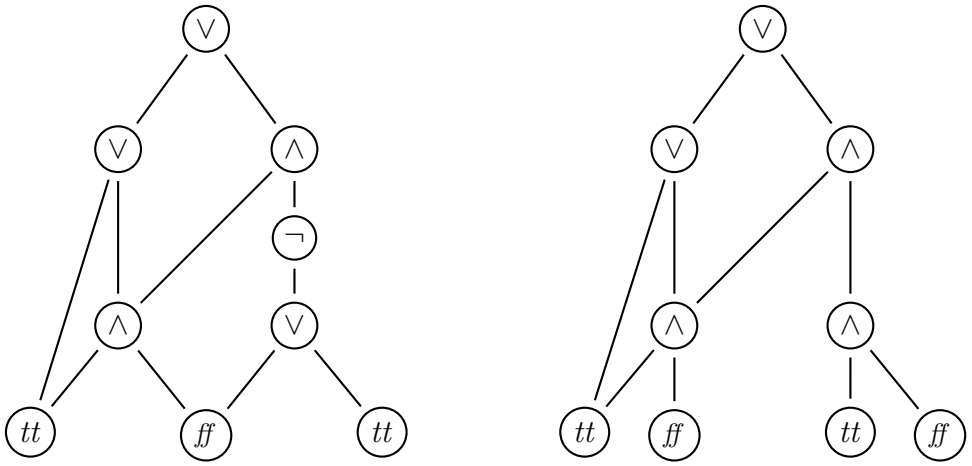

Figura 3.11: *Due circuiti costanti che producono tt; quello in parte (**b**) è monotono.*

Esempio 3.5.25.

- CFL $\neq \emptyset$ è il problema di verificare se il linguaggio generato da una data grammatica libera da contesto [25] è vuoto, cioè
$$\{L(G) \neq \emptyset \mid G \text{ è una grammatica libera dal contesto}\}.$$
- dato un insieme di variabili X e una congiunzione di un numero finito di formule di Horn $\mathtt{H}$,[26] HORN è il problema di decidere se una variabile $x \in X$ è deducibile da $\mathtt{H}$, cioè
$$\text{HORN} = \{\mathtt{H}, x \mid \mathtt{H} \vdash x\}.$$

Non è difficile vedere che entrambi i problemi citati stanno in $\mathcal{P}$; il lettore è invitato a ridurre in spazio logaritmico HORN a CIRCUIT VALUE, sfruttando la similarità che deriva dalla loro comune natura logica, e poi a ridurre CFL a HORN, sfruttando la similarità tra "deduzioni" e "derivazioni."

[25] Una *grammatica libera* è una quadrupla $G = \langle N, \Sigma, S, P\rangle$ dove
- $N \ni A, B, ...$ è l'alfabeto dei simboli *non terminali*;
- $\Sigma \ni a, b, ...$ è l'alfabeto dei simboli *terminali*;
- $S \in N$ è chiamato simbolo *distinto*;
- $P \subseteq (N \times (N \cup \Sigma)^+)$ è l'insieme *finito* delle *produzioni*, usualmente rappresentate come $A \to \alpha$.

Il linguaggio generato da G è $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$ dove $xAy \Rightarrow_G x\alpha y$ sse $A \to \alpha \in P$, per $x, y \in \Sigma^*$ e $\Rightarrow_G^*$ è la chiusura riflessiva e transitiva di $\Rightarrow_G$.

[26] Una formula di Horn è un'espressione booleana nella forma
$$x_1 \wedge x_2 \wedge \ldots \wedge x_k \implies y,\ y, x_i \in X,\ i \geq 0$$
Inoltre, diciamo che x è deducibile da una congiunzione di formule di Horn $\mathtt{H}$, in simboli $\mathtt{H} \vdash x$, se e solamente se
- $\implies x \in \mathtt{H}$ oppure
- $x_1 \wedge x_2 \wedge \ldots \wedge x_k \implies x$ e $\forall i.\ \mathtt{H} \vdash x_i$

$\mathcal{NP}$-completezza

Finalmente veniamo ai problemi intrattabili $\mathcal{NP}$ e al suo massimo campione, SAT, il problema della soddisfacibilità delle espressioni booleane. Questo risultato risale all'inizio degli anni '70 e fu dimostrato indipendentemente da Cook e Levin, da cui il nome.

Teorema 3.5.26 (Cook-Levin)**.**
SAT è $\mathcal{NP}$-completo

Dimostrazione. Sappiamo già che SAT $\in \mathcal{NP}$, perché abbiamo visto una procedura non deterministica che realizza una ricerca esaustiva, seppur implicita e lo decide: assegna *a caso* un valore di verità alle variabili dell'espressione booleana (fase che richiede tempo *non deterministico* polinomiale) e valuta l'espressione (o il circuito) booleano chiuso che si ottiene (fase che richiede tempo *deterministico* polinomiale): il problema è risolto positivamente se esiste un'espressione chiusa che valuta a tt.

Poichè CIRCUIT SAT $\leqslant_{\text{logspace}}$ SAT per la proprietà 3.5.17, ci basta dimostrare che CIRCUIT SAT è $\mathcal{NP}$-completo, ovvero che $\forall I \in \mathcal{NP}$ si ha che $I \leqslant_{\text{logspace}}$ CIRCUIT SAT.

Sia allora $I \in \mathcal{NP}$. Costruiamo $f \in$ LOGSPACE tale che $x \in I$ se e solamente se $f(x)$ è soddisfacibile. Per ipotesi c'è una macchina di Turing non deterministica N che decide I in tempo n^k. In altri termini, dato x esiste una computazione di lunghezza minore o uguale a $|x|^k$ che accetta x se e solamente se $x \in I$; ora, questa computazione può essere vista come una successione di scelte non deterministiche, anch'essa di lunghezza minore o uguale a $|x|^k$. Per semplicità conveniamo che il grado di non determinismo sia *esattamente* uguale a 2, cioè che ad ogni passo vi siano sempre 2 scelte possibili.[27] Codifichiamo la prima scelta con il bit ff e la seconda con il bit tt. Una computazione è allora una successione di bit della forma

$$B = b_0 b_1 \dots b_{|x|^k - 1}, \text{ con } b_i \in \{tt, ff\}.$$

Ovviamente nel caso della macchina non deterministica non abbiamo una tabella della computazione che rappresenti l'albero della computazione non-deterministica; tuttavia, se fissiamo una successione di scelte B sappiamo costruire la tabella T in funzione della macchina N, del dato x e *anche* della

[27] Possiamo sempre ricondurci a questa situazione: se la macchina N ha

(i) solo una scelta, consideriamo due scelte coincidenti, il che richiede uno stato ausiliario;

(ii) $m > 2$ scelte, si sceglie la prima o le rimanenti $m - 1$ e si ripete il procedimento $m - 2$ volte, il che richiede di aggiungere $m - 2$ nuovi stati.

successione di scelte B. Come fatto per CIRCUIT VALUE, la prima riga e la prima e l'ultima colonna sono fissate. Inoltre, $T(i,j)$ dipende, oltre che da $T(i-1,j-1), T(i-1,j)$ e da $T(i-1,j+1)$ *anche* da b_{i-1}, cioè dalla scelta fatta al passo precedente. Quindi il circuito $\overline{C_N}$ che possiamo costruire in perfetta analogia a quanto fatto nella dimostrazione della $\mathcal{P}$-completezza di CIRCUIT VALUE (teorema 3.5.23), ha $3m+1$ ingressi e m uscite. Di nuovo possiamo costruire in LOGSPACE un circuito $f(x)$ che ha come porte di ingresso le scelte non deterministiche codificate nel vettore B e i valori della riga $T(1,j)$ ottenuti dal dato di ingresso x.

Infine, la dimostrazione che $f(x)$ è soddisfacibile se e solamente se $x \in I$ è del tutto simile a quella del teorema 3.5.23. □

Vediamo alcuni esempi di altri problemi che sono $\mathcal{NP}$-completi, cioè tali per cui vi sono trasformazioni dall'uno all'altro e da e in SAT. La letteratura riporta migliaia di problemi di questo tipo, la gran parte dei quali è significativa dal punto di vista computazionale — quindi la classe $\mathcal{NP}$ è importante; per una discussione sulla sua rilevanza e della gran quantità di problemi completi per essa si veda l'articolo di Papadimitriou [ICALP'97].

Esempio 3.5.27 (Problemi $\mathcal{NP}$-completi)**.**
I seguenti problemi sono $\leqslant_{\text{logspace}}$-completi per $\mathcal{NP}$:

- HAM;
- CRICCA;
- Problema del Commesso Viaggiatore;
- Programmazione Intera: dato un sistema di disequazioni a coefficienti interi in n variabili trovare una soluzione intera.[28]

[28] Si ricordi che se si tralascia la parola "intero" il problema è in $\mathcal{P}$: basta usare il metodo elissoide o quello dei punti interni.

Abbiamo (quasi) finito – BOZZA!!

Ci fermiamo qui con lo studio delle classi di complessità e della loro gerarchia; si potrebbe continuare per mesi e anni, sia nello studio di classi $\mathcal{C} \subseteq \mathcal{NP}$, sia di classi $\mathcal{C} \subseteq \mathcal{P}$ (o $\subseteq$ LOGSPACE).

Qualcosa verrà detto nel prossimo capitolo, che prima introdurrà una classe di problemi "più trattabili" di quelli in $\mathcal{P}$: $\mathcal{NC}$ (Nick's Class). Essa consiste dei problemi risolvibili *in tempo* $\mathcal{O}(\log^k |x|)$ con un *numero di processori polinomiale* in $|x|$ che procedono in parallelo scambiandosi messaggi *istantaneamente*. Si suppone che essa sia la classe dei problemi veramente trattabili anche se

1. il numero dei processori di solito è prefissato
2. scambiarsi messaggi è un'attività costosa.

Un problema aperto è se $\mathcal{NC} = \mathcal{P}$; al momento pare che non sia cosí: ci sarebbero problemi polinomiali che non possono essere risolti efficentemente ricorrendo a un parallelismo piú che massiccio. Questo è un sintomo che via via che le capacità di calcolo meccanico aumentano, si va restringendo la classe dei problemi che consideriamo davvero trattabili.

Nel prossimo capitolo esamineremo di sfuggita anche le due estensioni seguenti agli algoritmi che ci permettono di aver ragione delle difficoltà insite nei problemi $\mathcal{NP}$-completi.

Algoritmi probabilistici. Durante una computazione è consentito scegliere a caso tra le varie opzioni non-deterministiche. Se l'algoritmo è furbo e la scelta veramente casuale (ma quanto è realistica quest'ultima opzione?) la probabilità di raggiungere una configurazione di accettazione può essere mantenuta minore di un dato ϵ. Il classico esempio è dato dall'algoritmo probabilistico per verificare se un numero è primo, il quale può avere un margine di errore minore della probabilità che si verifichi un guasto hardware durante la sua esecuzione [Rabin]. Lo stesso Rabin tuttavia affermò alcuni anni fa che se c'è un algoritmo probabilistico che risolve un problema in tempo polinomiale, ce n'è uno deterministico equivalente, ancora in tempo polinomiale. Questa sua intuizione è stata confermata proprio sul problema della primalità, che risulta essere in $\mathcal{P}$ [Agrawal-Kayal-Saxena, 2002].

In tempi piú recenti è stata introdotta la classe $\mathcal{BPP}$ (Bounded-error Probabilistic Polynomial-time) che contiene quei problemi che possono essere risolti da algoritmi probabilistici in tempo polinomiale e con una probabilità di errore minore di $\frac{1}{3}$. Alcuni autori ritengono che questi siano i problemi che ammettono una soluzione efficiente (si noti che $\mathcal{P} \subseteq \mathcal{BPP}$).

Algoritmi approssimati. Questo è un approccio leggermente diverso (e più classico): cerchiamo un algoritmo (euristico) che funzioni "bene" nella

maggior parte dei casi, cioè sia polinomiale anche se il problema è $\mathcal{NP}$-completo e che restituisca un risultato vicino a quello esatto per una misura ϵ prefissata. Per esempio, nel caso dei problemi di ottimizzazione, quale quello del Commesso viaggiatore, si ritiene accettabile una soluzione che non sia troppo lontana da quella ottima e si garantisce che l'algoritmo approssimato fornisce sempre soluzioni accettabili.

Con questo approccio si possono definire delle *soglie di approssimabilità*, attraverso le quali si riescono a distinguere tra loro problemi $\mathcal{NP}$-completi, suddividendo ulteriormente tale classe e raffinando la nozione di "difficoltà" (attenzione, però: le riduzioni viste *non* conservano le soglie di approssimabilità!)

Notiamo di sfuggita che già Turing trovava inadeguata la nozione di algoritmo che lui stesso aveva contribuito a definire, tanto da estenderla cosí:

Algoritmi con oracolo. Durante una computazione la macchina pone una domanda a un oracolo, e questi è sempre in grado di rispondere; per esempio l'oracolo decide se $x \in \overline{K}$, in barba alla non ricorsiva enumerabilità di tale insieme; oppure decide *immediatamente* se un espressione è soddisfacibile o no. Il lettore rifletta su quanto siano realizzabili questi algoritmi.[29]

Ci dobbiamo forse considerare sconfitti dato che

- esistono funzioni non calcolabili?
- esistono problemi "troppo" difficili eppure interessanti?

NO, certamente no. Intanto abbiamo capito i nostri limiti o piuttosto quelli che ci siamo posti con l'idea intuitiva di algoritmo che ci siamo dati. Inoltre, i vari risultati negativi ci stimolano a trovare modi e maniere di aggirare le difficoltà, modificando addirittura la nozione stessa di algoritmo, come visto sopra. Tutto questo è stato possibile perché, seguendo il precetto galileano, abbiamo fatto lo sforzo di formalizzare la nostra intuizione e, come dice Italo Calvino:

> "la fantasia senza regole esatte è come la marmellata senza una fetta di pane su cui spalmarla."

[29] Una variante delle macchine con oracolo è quella delle macchine interattive: un utente (lo pseudo-oracolo) fornisce a tempi successivi parte degli ingressi modificando *durante il calcolo* i dati su cui la macchina opera. (Naturalmente se la combinazione del sistema interattivo con l'utente risulta in un sistema che non accetta altri ingressi, si ottiene di nuovo un algoritmo "classico.") Questo modello ricorda da vicino quelli per i sistemi "reattivi" o "concorrenti."