Università di Pisa

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Triennale in Matematica

Tesi di Laurea Triennale

Alternanza, parallelismo e complessità

Candidato
Pietro Battiston

Relatore
Prof.
Alessandro Berarducci

Controrelatore
Prof.
Mauro Di Nasso

Anno Accademico 2007/2008

"Houston, we have a [PSPACE-complete] problem."
John Swigert, Jr.

Indice

Capitolo 1

Introduzione

Si definisce SAT come l'insieme delle formule booleane soddisfacibili, ovvero di quelle formule booleane che risultano vere per almeno un'assegnazione di valori alle variabili. Non è noto se SAT $\in$ P (=PTIME), ovvero se esistano algoritmi per stabilire in tempo polinomiale (rispetto alla lunghezza dell'input) se una data formula booleana appartenga a SAT. È invece semplice verificare che SAT $\in$ NP (l'insieme dei problemi decisionali algoritmicamente risolubili in tempo polinomiale con una macchina non deterministica).

Uno dei problemi aperti più importanti della teoria della complessità computazionale (l'unico, tra i 7 problemi del Millennium Prize del Clay Institute, ad essere nato in campo informatico) è se P = NP.

A questo riguardo un risultato fondamentale è il seguente:

Teorema 1. *(Cook, 1971 [3]): SAT è NP-completo.*

dove per "completo" si intende rispetto a riduzioni polinomiali. Di conseguenza, se SAT $\in$ P, allora P = NP,

La congettura più accreditata è che P $\neq$ NP; di conseguenza la NP-completezza di un problema è considerata un'indicazione del fatto che probabilmente il problema è intrattabile in tempo polinomiale. A partire dal classico risultato di Cook molti altri problemi sono stati dimostrati NP-completi tramite una riduzione a SAT ([6]).

Valgono le inclusioni

$$\text{P} \subset \text{NP} \subset \text{EXPTIME} \qquad (*)$$

dove EXPTIME è l'insieme dei problemi decisionali algoritmicamente risolubili in tempo esponenziale. Segue dal teorema di gerarchia ([4]) che

$$\text{P} \neq \text{EXPTIME}\ ,$$

e quindi almeno una delle due inclusioni in $(*)$ è stretta.

Se oltre ai connettivi booleani ammettiamo la possibilità di quantificare (universalmente ed esistenzialmente) su variabili proposizionali otteniamo l'insieme delle *formule booleane quantificate* e possiamo studiarne il sottoinsieme QSAT di quelle vere. Ad esempio la formula $\forall A(\neg A \rightarrow \exists B(A \rightarrow B))$ appartiene a QSAT. Il problema di stabilire se una formula appartiene a QSAT è, almeno

apparentemente, più complesso del problema analogo per SAT (che equivale al caso particolare di QSAT in cui tutti i quantificatori sono esistenziali e prenessi). Si pone quindi il problema, che affrontiamo in questa tesi, di studiare la complessità computazionale di QSAT.

La prima osservazione è che anche QSAT appartiene ad EXPTIME. Come vedremo però ci sono risultati che suggeriscono che QSAT $\notin$ NP (il risultato opposto confuterebbe alcune congetture generalmente ritenute vere). Per poter esporre questi risultati occorre introdurre delle classi di complessità che dipendono dallo spazio di memoria piuttosto che dal tempo di calcolo. Una delle più importanti di tali classi è PSPACE (l'insieme dei problemi decisionali risolubili in spazio polinomiale). Valgono le inclusioni

$$\mathrm{P} \subset \mathrm{NP} \subset \mathrm{PSPACE} \subset \mathrm{EXPTIME}$$

e si congettura che le inclusioni siano tutte strette. In questa tesi esporremo una dimostrazione del seguente:

Teorema 2. *(Stockmeyer, Meyer, 1973 [12]): QSAT è PSPACE-completo*

Ne segue che a meno che NP = PSPACE, QSAT $\notin$ NP, confermando l'intuizione che QSAT sia più complesso di SAT.

L'interesse della classe PSPACE è che essa può essere presa come possibile definizione della classe dei problemi decidibili in tempo polinomiale con un calcolatore capace di una forte forma di parallelismo. Vale infatti, a livello informale, il paradigma "Spazio deterministico = Tempo parallelo". Una delle forme in cui è possibile precisare questo paradigma è il seguente:

Teorema 3. *(Chandra, Kozen, Stockmeyer, 1981 [1]): PSPACE = APTIME*

dove ATIME è la classe dei problemi risolubili in tempo polinomiale con una macchina di Turing alternante (un modello di una certa forma di parallelismo). Le macchine di Turing alternanti sono una generalizzazione di quelle nondeterministiche: mentre le prime permettono di fare non-deterministicamente delle scelte prendendo l'OR dei risultati delle subcomputazioni, nelle macchine alternanti si ammette la possibilità di ramificazioni non-deterministiche sia di tipo OR che di tipo AND (ciò permette di simulare i quantificatori rispettivamente esistenziali e universali). L'utilità teorica delle macchine alternanti è che esse facilitano la progettazione di algoritmi per quei problemi nella cui definizione compaiono, implicitamente o esplicitamente, molti quantificatori alternati. Uno dei modi per dimostrare il teorema 3 è far vedere che QSAT è completo, oltre che per PSPACE, anche per APTIME, come vedremo in questa tesi.

QSAT risulta particolarmente interessante anche perché esso fornisce uno spunto per studiare la complessità di un'ampia classe di giochi ad informazione completa in cui due giocatori alternano le mosse. Si vedrà inoltre che il problema di stabilire chi abbia una strategia vincente nel gioco del GO (generalizzato a scacchiere $n \times n$) si dimostra essere PSPACE-completo dimostrandone l'equivalenza a QSAT ([7, 8]).

1.1 Terminologia e notazione

In questa esposizione, chiameremo *alfabeto* un qualsiasi insieme di simboli; in particolare, tutti gli alfabeti considerati saranno finiti. Nell'ambito delle macchine di Turing, daremo poi una definizione leggermente più restrittiva di tale

termine.

Una *stringa* sarà una sequenza finita di *simboli* in un dato *alfabeto*.

Per denotare una stringa, elencheremo, tra parentesi quadre, i simboli che vi appaiono:

$$[\sigma_1, \sigma_2, \sigma_3, \dots] ,$$

o più semplicemente, quando ciò non crei ambiguità:

$$[\sigma_1\sigma_2\sigma_3 \dots] .$$

Questa notazione ci permetterà anche di rappresentare in modo sintetico alcune semplici operazioni sulle stringhe:

- la concatenazione:
$$[0] + [1] = [0, 1]$$
(notare la differenza rispetto a $[0 + 1] = [1]$ e $[0, +, 1] = [0] + [+] + [1]$),
- la ripetizione:
$$[0]^5 = [00000]$$
(notare la differenza rispetto a $[0^5] = [0]$)
- e l'inversione:
$$[012]^{-1} = [210]$$

Dato un alfabeto Σ, indicheremo con Σ^* l'insieme di tutte le stringhe ottenibili dai suoi simboli; in altri termini,

$$\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^i$$

Indicheremo con il semplice $*$ un simbolo qualsiasi (solitamente in un alfabeto dato); ad esempio, se stiamo lavorando sull'alfabeto $\{a, b, c\}$ e diciamo che una certa proprietà vale per le stringhe della forma

$$[a * c] ,$$

intendiamo che vale per $[aac]$, $[abc]$ e $[acc]$.

Data una stringa s, scriveremo s_i per indicare l'i-esimo elemento di s.

Infine, dato un numero $n \in \mathbb{N}$, indicheremo con $bin(n)$ la sua rappresentazione binaria, che sarà una stringa nell'alfabeto $\{0, 1\}$; ad esempio:

$$bin(13) = [1101]$$

Capitolo 2

Macchine di Turing

La macchina di Turing è il più classico modello astratto di un calcolatore. Consiste in una macchina a stati finiti con un nastro di memoria infinito suddiviso in celle che un cursore può scorrere, leggere e scrivere; ad ogni "passo", in base allo stato attuale ed al simbolo che il cursore legge, una regola stabilisce in quale stato deve passare la macchina, quale simbolo deve scrivere il cursore e infine se ed eventualmente in quale direzione (destra/sinistra) deve poi spostarsi.

Sono state studiate altre rappresentazioni, come il λ-calcolo, le macchine a registri e le funzioni μ-ricorsive, che si sono viste essere equivalenti nella capacità espressiva formale; tuttavia, la macchina di Turing rimane quella più diffusa nello studio della complessità, perché piuttosto semplice[1] e perché particolarmente adatta alla classificazione dei problemi in base alla loro difficoltà (si veda più avanti le definizioni 16 e 17).

Fornire una definizione di cosa sia *calcolabile* è un compito assolutamente non banale e che forse esula dal semplice studio matematico e tecnico dei metodi di calcolo; per questo motivo, la cosiddetta *Tesi di Church-Turing*, ovvero:
"Tutto ciò che è calcolabile è calcolabile da una macchina di Turing",
si può considerare allo stesso tempo come una congettura e come una definizione, nel senso che essa è, come un assioma, alla base di tutta la teoria della computazione (ci riferiamo spesso, anche indirettamente, ad essa per stabilire se un certo problema è o no calcolabile) e l'impressione che essa rappresenti un'importante verità (anche se una di cui è difficile anche solo immaginare una dimostrazione) è in fondo il motivo per cui questa teoria viene considerata importante.

Questo status di rappresentazione canonica di un calcolatore si riflette nella terminologia standard: un qualsiasi sistema formale di calcolo si dice "*Turing equivalente*" se è possibile adattarvi una versione di qualsiasi algoritmo svolgibile con una macchina di Turing, ovvero, in sostanza, se viene ritenuto sufficientemente potente da risolvere ogni problema risolubile in modo meccanico[2]. Solitamente la dimostrazione della Turing-equivalenza passa attraverso

[1] Sono state create macchine di Turing addirittura sotto forma di configurazioni iniziali del celebre gioco Life di Conway (si veda [9]), ed è dimostrato che ciò è possibile anche all'interno della cosiddetta *Regola 110*, un semplice automa cellulare monodimensionale descritto da Stephen Wolfram [2].

[2] L'aggettivo "meccanico", utilizzato per designare il tipo di calcolo di cui è capace una

l'"emulazione" di una generica macchina di Turing, oppure di una cosiddetta "*macchina di Turing universale*", all'interno del nuovo sistema.

2.1 Macchine di Turing deterministiche

Definizione 1. *Nell'ambito delle macchine di Turing , definiamo* alfabeto *un qualsiasi insieme* Σ *finito contenente almeno i due simboli speciali* $\triangleright$ *("indicatore di inizio nastro") e* $\sqcup$ *("indicatore di cella vuota").*

Definizione 2. *Dati un alfabeto* Σ *ed un insieme* Q *(l'insieme degli* stati *che la macchina può assumere) contenente almeno un elemento* q_i *(lo* stato iniziale*) ed un elemento* q_f *(lo* stato finale*), sia data una funzione*

$$\delta : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \downarrow, \rightarrow\}$$

(detta funzione di transizione*) tale che*

$$\delta(\triangleleft, *) = (\triangleleft, *, \rightarrow) \ .$$

Allora si dice macchina di Turing *la quintupla*

$$(\Sigma, Q, q_i, q_f, \delta)$$

Per noi, il nastro sarà monodimensionale ed infinito *in una sola direzione*[3]; ha quindi una prima cella, in cui, come vedremo, comparirà il simbolo $\triangleright$; esso è un simbolo particolare: la condizione data sulla funzione di transizione fa sì che esso non possa essere sovrascritto e funga da limite al movimento del cursore.

L'esecuzione di una macchina di Turing è per sua natura legata al *tempo*[4] ed è fondata su tre concetti fondamentali: la *configurazione*, il *passo* e la *computazione*.

Definizione 3. *Si dice* configurazione *di una macchine di Turing* $(\Sigma, Q, q_i, q_f, \delta)$ *un terzetto* (s, q, a) *dove* $s \in \Sigma^*$ *è una stringa di simboli dell'alfabeto,* $q \in Q$ *è uno stato della macchina ed* a *è un numero naturale.*

Non è difficile immaginare l'interpretazione di questi simboli: a indica la posizione del cursore della macchina in un certo istante, q lo stato in cui si trova la macchina ed s descrive la stringa che si trova sul nastro: più precisamente, data $s = [s_1, s_2 \dots s_n]$, il nastro avrà i seguenti simboli memorizzati:

$$(\triangleright s_1 s_2 s_3 \dots s_n \sqcup \sqcup \sqcup \dots) \ .$$

Questi tre elementi sono effettivamente sufficienti a definire univocamente la configurazione. Evidentemente, per una macchina data non tutte le configurazioni sono *sensate*; in particolare, fissato un input ci saranno certe configurazioni

macchina, è piuttosto diffuso in letteratura e perciò verrà talvolta usato anche in questa tesi, nonostante al giorno d'oggi sarebbe forse più sensato qualificarlo come "elettronico".

[3] In letteratura spesso si suppone che il nastro di una macchina di Turing sia infinito in entrambe le direzioni; a noi ciò non serve e complicherebbe inutilmente alcune dimostrazioni. Sono state studiate anche macchine di Turing con nastri pluridimensionali, ma anche quelle esulano dall'ambito di questo lavoro.

[4] Sebbene ciò possa apparire scontato, è uno dei motivi per cui le macchine di Turing sono una formalizzazione "comoda" e simile ad un calcolatore reale.

irraggiungibili.

L'esecuzione di una qualsiasi macchina di Turing partirà da una *configurazione iniziale* della forma $(s, q_i, 1)$, ovvero con la macchina nello stato iniziale ed il cursore posizionato all'inizio del nastro (ed s una generica stringa in Σ^*, che viene considerata l'*input*).

Da una configurazione data, la macchina si potrà spostare in un'altra, purché le due configurazioni rispettino certi criteri: l'unico simbolo del nastro che può cambiare deve essere quello in corrispondenza del cursore ed il cambiamento di simbolo, stato e posizione del cursore deve avvenire come stabilito dalla legge di transizione. Tali criteri sono formalizzati dalla seguente:

Definizione 4. *Diremo che una macchina di Turing M* si sposta in un passo *da una configurazione (s, q, a) ad un'altra (s', q', a'), e scriveremo*

$$(s, q, a) \xrightarrow{M} (s', q', a')$$

(o, quando non c'è rischio di equivoco, semplicemente $(s, q, a) \to (s', q', a')$) se:

- $|a - a'| \leq 1$;
- $$\delta(s_a, q) = (s'_a, q', \alpha) \text{ con } \begin{cases} \alpha = \leftarrow & \text{se } a' - a = -1 \\ \alpha = \downarrow & \text{se } a' - a = 0 \\ \alpha = \rightarrow & \text{se } a' - a = 1 \end{cases}$$
- $\forall i \in \mathbb{N}$,
$$i \neq a \Rightarrow s_i = s'_i$$

Una *computazione* sarà un succedersi di diverse configurazioni, in cui la prima è una configurazione iniziale e le altre rispettano le condizioni appena fissate.

Definizione 5. *Si dice* computazione *di una macchina di Turing $(\Sigma, Q, q_i, q_f, \delta)$ una sequenza o successione di sue configurazioni $((s_j, q_j, a_j))_{j \geq 0}$ tale che:*

1. $q_0 = q_i$
2. $a_0 = 1$
3. $\forall j > 0$ *che compaia tra gli indici della successione,*
$$(s_{j-1}, q_{j-1}, a_{j-1}) \to (s_j, q_j, a_j)$$
4. *se esiste k tale che $q_k = q_f$, allora la sequenza sia composta da esattamente $k + 1$ configurazioni*[5]
5. *viceversa, se essa è una successione infinita, $q_j \neq q_f$ $\forall j \in \mathbb{N}$:*

Se si verifica il penultimo caso, diciamo che la macchina *accetta l'input s_0 in k passi.*

[5] Ovvero la configurazione avente come stato $q_k = q_f$ sia l'ultima raggiunta (in particolare stiamo chiedendo che k tale che $q_k = q_f$ sia unico).

2.2 Non-determinismo

Abbiamo detto che definire l'alfabeto, gli stati e la funzione di transizione significa determinare univocamente una particolare macchina di Turing. Più precisamente, quella data è la definizione di macchina di Turing *deterministica*, il tipo più semplice, oltre che più simile al comportamento dei calcolatori reali, di macchina di Turing.

Di seguito, definiamo altre due tipologie di macchine di Turing, fondamentali per il proseguimento dei nostri ragionamenti. Rispetto alla macchina di Turing deterministica, ricalcano meno la natura degli attuali calcolatori, ma vedremo più avanti che ci sono importanti motivi teorici e pratici per cui suscitano interesse:

Definizione 6. *Dati* Σ, Q, q_i *e* q_f *come nella definizione 2, sia definita una funzione di transizione:*

$$\delta : \Sigma \times Q \to \mathcal{P}(\Sigma \times Q \times \{\leftarrow, \downarrow, \rightarrow\})$$

(dove con $\mathcal{P}(X)$ *indichiamo l'*insieme delle parti*, o* insieme potenza *di un dato insieme* X*), tale che tutti gli elementi di* $\delta(\triangleleft, *)$ *siano della forma* $(\triangleleft, *, \rightarrow)$*.*

Allora si dice macchina di Turing nondeterministica *la quintupla*

$$(\Sigma, Q, q_i, q_f, \delta)$$

L'idea di fondo che ha portato alla definizione della macchina di Turing nondeterministica è la formalizzazione di un sistema di calcolo in cui la computazione si svolga contemporaneamente su più *rami paralleli*, come in una sorta di calcolatore multiprocessore (con un numero arbitrariamente grande di unità di calcolo): per questo motivo, ad ogni passo, la computazione può effettuare diverse operazioni *contemporaneamente*, ed ognuna di queste scelte si riflette poi in una *sub-computazione* indipendente dalle altre.

Osserviamo che una macchina deterministica si può vedere come un caso particolare di macchina nondeterministica, in cui l'immagine di δ contiene solo singoletti[6]

Diamo perciò una nuova definizione di vari concetti già introdotti nel caso deterministico:

Definizione 7. *Diremo che una macchina di Turing nondeterministica* M si sposta in un passo *da una configurazione* (s, q, a) *ad un'altra* (s', q', a')*, e scriveremo*

$$(s, q, a) \xrightarrow{M} (s', q', a')$$

(o semplicemente $(s, q, a) \to (s', q', a')$*) se:*

- $|a - a'| \leq 1$*;*

[6] La nostra definizione di macchina di Turing deterministica prevede che la legge di transizione abbia come immagine un sottoinsieme di $\Sigma \times Q \times \{\leftarrow, \downarrow, \rightarrow\}$, non del suo insieme potenza; tuttavia, questa è una differenza dalle conseguenze trascurabili.

- $$(s'_a, q', \alpha) \in \delta(s_a, q) \text{ con } \begin{cases} \alpha = \leftarrow & \text{se } a' - a = -1 \\ \alpha = \downarrow & \text{se } a' - a = 0 \\ \alpha = \rightarrow & \text{se } a' - a = 1 \end{cases}$$

- $\forall i \in \mathbb{N}$,
$$i \neq a \Rightarrow s_i = s'_i$$

Definizione 8. *Una* computazione *di una macchina di Turing nondeterministica M è un* albero orientato etichettato *(si veda l'appendice A.1.1), in cui ad ogni nodo è associata una configurazione e tale che*

- *la radice sia etichettata con una configurazione della forma $(s, q_0, 1)$,*
- *i figli di un nodo etichettato con una configurazione α siano etichettati con tutte e sole le configurazioni β tali che $\alpha \rightarrow \beta$*

(non richiediamo che i genitori di una configurazione β siano tutti gli α tali che $\alpha \rightarrow \beta$; ciò è inutile ai fini dello studio delle computazioni ed impossibile mantenendo la condizione che la computazione sia appunto un albero orientato).

Nell'esecuzione di una macchina di Turing nondeterministica , ogni ramo può essere visto come una particolare computazione deterministica; in questa ottica, stabiliamo che la macchina accetta l'input se almeno una delle sub-computazioni che ne fanno parte accetta. Questo concetto è precisato nella seguente:

Definizione 9. *Diciamo che una macchina nondeterministica* accetta l'input in k passi *se a livello k nell'albero delle computazioni (e a nessun livello precedente) compare una configurazione accettante (con stato q_f).*

Si osservi che, mentre nel caso deterministico si poteva semplicemente dividere le possibili computazioni in due classi - quelle che terminano e accettano e quelle che non terminano e non accettano - potremo ora avere configurazioni che terminano e non accettano, qualora l'albero sia finito ma senza nessuna configurazione accettante.

Sebbene quello definito possa sembrare esattamente il comportamento di un'ipotetico computer con un numero infinito (o perlomeno arbitrariamente aumentabile) di unità di calcolo (ovvero processori, ognuno dotato di una propria memoria), la condizione di terminazione è piuttosto limitativa; la macchina accetta l'input se uno dei rami della computazione accetta, ma i rami sono veramente come macchine separate, mentre in un computer capace di parallelismo si suppone che le sub-computazioni possano interagire tra di loro. Anche per questo motivo ha senso una generalizzazione ulteriore.

2.3 Alternanza

Definizione 10. *Dati Σ, Q, q_i, q_f e δ come nella definizione precedente, sia $Q_\exists$ (l'insieme degli stati* esistenziali*) un sottoinsieme di Q. Allora si dice* macchina di Turing alternante *la sestupla*

$$(\Sigma, Q, q_i, q_f, Q_\exists, \delta)$$

e si indica con $Q_\forall$ l'insieme (degli stati universali*) $Q \setminus Q_\exists$.*

Diciamo una data configurazione (s, q, a) esistenziale *(rispettivamente* universale*) se q è esistenziale (rispettivamente universale).*

I concetti di *passo* e *computazione* sono nel caso alternante definiti in modo identico al caso nondeterministico. Prima di ridefinire anche cosa intendiamo per *accettazione*, diamone una nuova definizione nel caso nondeterministico:

Definizione 11. *Sia M una macchina di Turing nondeterministica e sia $\mathcal{A}_w$ la sua computazione su input w. Diciamo che:*

- *una configurazione in $\mathcal{A}_w$* accetta in 0 passi *se il suo stato è q_f*
- *una configurazione in $\mathcal{A}_w$* accetta in k passi *se almeno una delle configurazioni figlie accetta in $k-1$ passi, nessuna accetta in meno di $k-1$ passi e la configurazione stessa non ha stato q_f*
- *M accetta w in k passi se e solo se la configurazione associata alla radice di $\mathcal{A}_w$, ovvero $(w, q_i, 0)$, accetta in k passi*

Si verifica facilmente che questa definizione è equivalente alla 9. Se adesso la preferiamo, è perché è estremamente simile a quella che diamo nel caso alternante:

Definizione 12. *Sia M una macchina di Turing alternante e sia $\mathcal{A}_w$ la sua computazione su input w. Diciamo che:*

- *una configurazione in $\mathcal{A}_w$* accetta in 0 passi *se il suo stato è q_f o se è una configurazione universale senza configurazioni figlie*
- *una configurazione in $\mathcal{A}_w$* accetta in k passi *se non accetta in 0 passi e:*
  - *è universale e tutte le configurazioni figlie accettano in $k-1$ o meno passi (ed in particolare, almeno una accetta in esattamente $k-1$), oppure*
  - *è esistenziale ed almeno una delle configurazioni figlie accetta in $k-1$ (e nessuna delle altre accetta in meno di $k-1$ passi)*
- *M accetta w in k passi se e solo se la configurazione associata alla radice di $\mathcal{A}_w$, ovvero $(w, q_i, 0)$, accetta in k passi*

Diciamo ovviamente una configurazione *non accettante* se non esiste $k \in \mathbb{N}$ tale che essa accetti in k passi.

Confrontando le due definizioni appena date, si osserva che la macchina nondeterministica è un caso particolare di macchina alternante in cui tutte le configurazioni sono esistenziali. In effetti, vedremo più avanti che però questa semplice generalizzazione, formulata per la prima volta nel 1981 (si veda [1]), sembra aumentare in modo sostanziale la potenza di calcolo delle macchine.

Si può facilmente notare che lo stato q_f è conservato nelle macchina di Turing alternante semplicemente per coerenza con il caso deterministico e nondeterministico, in modo che ne sia una generalizzazione; infatti, supponendo che una

macchina ne sia priva non poniamo alcuna limitazione alle sue capacità di calcolo, dato che è possibile sostituirlo semplicemente con configurazioni universali senza figli. In questa tesi, questa osservazione ci permetterà di semplificare alcuni passaggi.

2.4 Funzioni di transizione e *regole*

La funzione di transizione ha sempre dominio finito e, data la struttura particolare del suo dominio, la si può rappresentare agevolmente come una tabella a 2 entrate, in cui ad esempio ad ogni simbolo corrisponde una riga e ad ogni stato una colonna; questa particolare tabella, che chiamiamo *tabella di transizione*, ci tornerà utile più avanti.

Chiameremo a volte "*regola*" una singola valutazione di una funzione di transizione δ su un particolare elemento di $\Sigma \times Q$ o su una particolare collezione di elementi: stilare una lista di regole è spesso il modo più comodo per definire la funzione di transizione δ di una macchina. Per semplificare ulteriormente questo compito, ometteremo le regole che non risultano *utili*, ovvero corrispondenti a coppie (σ, q) che non potranno mai verificarsi nella macchina (date le altre regole); δ potrà essere estesa a tutto $\Sigma \times Q$ in modo arbitrario, senza che le possibili scelte inficino i nostri ragionamenti.

Ci tornerà utile in seguito supporre che una macchine di Turing abbia uno *stato pozzo* $\bar{q}$ tale che, per ogni configurazione α avente stato $\bar{q}$,

$$\alpha \to \alpha$$

e, per ogni $\beta \neq \alpha$,

$$\alpha \not\to \beta \ .$$

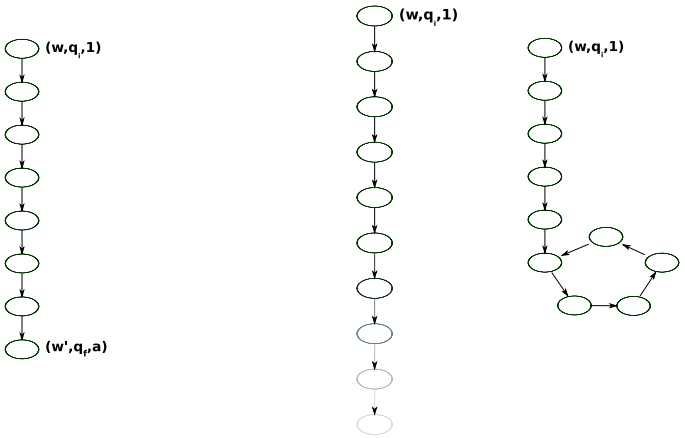
In altri termini, uno stato pozzo serve a garantirci che la macchina entri in un ciclo senza fine e quindi in particolare *non accetti*, e svolge quindi una funzione simmetrica a quella di q_f. Aggiungere ad una macchina uno stato pozzo $\bar{q}$ equivale semplicemente a stabilire che la funzione di transizione, una volta che la macchina entra nello stato $\bar{q}$, riscriva sempre i simboli letti, rimanga sempre in $\bar{q}$ e non muova i cursori.

2.5 Grafi delle configurazioni

Sebbene, come appena visto, le computazioni delle macchine di Turing siano definite come successioni e alberi etichettati, ci risulterà estremamente comodo rappresentarle come *grafi* in cui due "passaggi" per una identica configurazione sono indistinguibili. Tale concetto è formalizzato dalla seguente:

Definizione 13. *Si dice* grafo delle configurazioni *di una macchine di Turing (deterministica, nondeterministica o alternante) M con alfabeto Σ ed insieme degli stati Q il grafo $G = (V, E)$ dove:*

$$V = \Sigma^* \times Q \times \mathbb{N}$$



(a) Lineare finita (b) Lineare infinita (c) Ciclica

Figura 2.1: Le tre possibili conformazioni di un grafo delle configurazioni deterministico.

$$E = \left\{(c, c') \in V^2 | c \xrightarrow{M} c'\right\}$$

In altri termini, ogni vertice è associato - questa volta biunivocamente - ad una configurazione, e c'è un arco tra un vertice ad un altro se la macchina si sposta in un passo dalla configurazione del primo alla configurazione del secondo.

Osserviamo en passant che di un grafo delle configurazioni ci interesserà solitamente solo la parte raggiungibile da una certa configurazione (verosimilmente, quella iniziale).[7] Nel caso particolare delle macchine deterministiche, la forma di questo sottografo non riserverà grandi sorprese: esso sarà sempre lineare (finito o infinito) oppure lineare con l'aggiunta di un *ciclo* finale (Figura 2.1). Nei casi nondeterministico e alternante, invece, in cui ogni vertice può avere più di un arco uscente, la sua forma potrà variare.

2.6 Macchine multinastro

Una generalizzazione molto comune delle macchine di Turing è quella delle macchine di Turing *multinastro*, aventi appunto diversi nastri in cui leggere e scrivere simboli, e un cursore per nastro in grado di muoversi indipendentemente dagli altri. Il funzionamento di una macchina di Turing deterministica a k nastri sarà anologo a quello di una normale, ma la funzione di transizione non sarà più della forma:

$$\delta : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \downarrow, \rightarrow\} ,$$

ma:

$$\delta : \Sigma^k \times Q \to \Sigma^k \times Q \times \{\leftarrow, \downarrow, \rightarrow\}^k ,$$

[7] Si noti che questa parte può non essere un'intera componente connessa, dato che non ci interesseranno la configurazioni *da cui* si può raggiungere quella parte, ma solo quelle a cui si può *arrivare*. Ad esempio, la componente connessa di una configurazione iniziale può tranquillamente contenere configurazioni irraggiungibili per la macchina.

e descriverà contemporaneamente, in funzione dei k simboli letti e dello stato, i k simboli da scrivere e i k movimenti da effettuare.
Similmente, per una macchina di Turing a k nastri nondeterministica o alternante:

$$\delta : \Sigma^k \times Q \to \mathcal{P}\left(\Sigma^k \times Q \times \{\leftarrow, \downarrow, \rightarrow\}^k\right) .$$

All'inizio di una qualsiasi computazione, i nastri dovranno essere tutti vuoti a parte il primo, che sarà considerato il *nastro di input*.

Le macchina di Turing multinastro saranno per noi molto importanti nello studio della complessità, e non a caso ci si basa solitamente su di esse, in letteratura, per cercare di stabilire una classificazione dei problemi in base alla loro difficoltà; questi concetti saranno precisati nel capitolo successivo, e vedremo anche come (a differenza ad esempio della generalizzazione che porta dal paradigma deterministico a quello nondeterministico, o da quello nondeterministico a quello alternante) l'introduzione dei nastri aggiuntivi non cambi così drasticamente le capacità di calcolo, perlomeno nei casi che interesseranno a noi. Per questi motivi, nei capitoli successivi ci riferiremo sempre, quando non diversamente specificato, a macchine di Turing multinastro.

Osserviamo che una configurazione di una macchina a k nastri non sarà più un elemento di

$$\Sigma^* \times Q \times \mathbb{N}$$

ma di

$$(\Sigma^*)^k \times Q \times \mathbb{N}^k$$

Solitamente, nelle macchine multinastro ogni nastro ha uno scopo specifico che lo rende diverso dagli altri; per questo motivo ci risulterà utile supporre che una macchina multinastro abbia un alfabeto *per ogni nastro*, invece che uno solo globale. Ciò semplifica le dimostrazioni e non è affatto restrittivo, dato che una macchina con k nastri ed un alfabeto Σ_i diverso per ogni nastro si comporterà esattamente come una macchina avente alfabeto

$$\Sigma = \bigcup_{i \leq k} \Sigma_i$$

che si sposti in uno stato pozzo se nel nastro di input compare un simbolo non appartenente a Σ_1.

Capitolo 3

Problemi e classi di complessità

Sebbene in un contesto più ampio le macchine di Turing siano state immaginate come dei computer che, ricevendo un input sotto forma di una stringa, emettano un output (sotto forma di un'altra stringa che rimane su un nastro alla fine della computazione), noi ci limiteremo a studiare macchine di Turing prive di output, o meglio il cui solo output è dato dall'eventuale accettazione, a prescindere dallo stato finale del nastro. Ciò comporta che le possibili risposte sono solo due: "accetta" o "non accetta"; i problemi che tali macchine possono risolvere sono di tipo *decisionale*.

Più formalmente,

Definizione 14. *dato un alfabeto* Σ*, un* problema *(decisionale) è un sottoinsieme* S *di* $(\Sigma)^*$ *tale che esista una macchina* M *con alfabeto* $\Sigma \cup \{\triangleright, \sqcup\}$ *che ne accetti tutti e soli gli elementi. Si dice che una tale macchina* M risolve S.

Molti problemi matematici decisionali interessanti possono essere visti, modulo un'opportuna codifica, come "problemi" nel significato appena formalizzato del termine.

3.1 Classi di complessità

Sia dato un problema S ed una macchina di Turing che lo risolve. Se avviata con input $w \in S$, essa utilizzerà, nel corso della sua esecuzione, una certa quantità finita di "risorse", in termini di tempo (numero di passi prima dell'arresto) e spazio. Se il primo concetto è stato formalizzato, è opportuno invece precisare il secondo (dando una definizione relativa al paradigma alternante e - di conseguenza - a quello nondeterministico e a quello deterministico, che ne sono casi particolari). Quello che noi chiameremo "spazio occupato" sarà né più né meno che il numero di celle di memoria occupate, anche solo temporaneamente, sui nastri durante la computazione (osserviamo che se una macchina accetta in tempo k, ciò che succede dopo non è più interessante ai fini del problema che la macchina decide):

Definizione 15. *Data una macchina di Turing alternante M a k nastri ed un input w tale che M accetti w in p passi, diciamo che la computazione utilizza spazio m se nei primi p livelli dell'albero ogni configurazione*

$$((s_1, s_2 \dots s_k), q, (a_1, a_2 \dots a_k))$$

è tale che

$$\left| \sum_{i \leq k} s_i \right| \leq m \ ,$$

e ve ne è almeno una tale che

$$\left| \sum_{i \leq k} s_i \right| = m$$

.

Osserviamo che il concetto di spazio utilizzato *non traduce* integralmente la "quantità di informazione" contenuta in una configurazione, dato che trascura sia lo stato che la posizione dei cursori. Tuttavia, questo non deteriora la qualità della nostra analisi, che si concentrerà sul comportamento *asintotico* delle macchine (ovvero per un input di dimensione grande): da tale punto di vista, infatti, la "quantità di informazione" implicita nello stato è costante e quindi trascurabile, mentre la posizione del cursore è generalmente minore della lunghezza della stringa, e quindi il numero di bit necessari per memorizzarla esponenzialmente minore.[1]

L'impiego di spazio e tempo potrà variare in funzione di w; tuttavia, proprio generalizzando la dipendenza da w, possiamo classificare in base alla sua complessità il problema stesso. Non saremo interessati al costo che l'esecuzione di un algoritmo comporta a prescindere dall'input, ovvero ad eventuali costanti additive intrinsecamente associate alla macchina scelta. Data quindi una funzione $f : \mathbb{N} \to \mathbb{N}$,

Definizione 16. *Diciamo che un problema S sta nella classe $DTIME(f(n))$ (rispettivamente, $NTIME(f(n))$ o $ATIME(f(n))$) se esiste una macchina di Turing deterministica (rispettivamente, nondeterminstica o alternante) M ed un $\epsilon \in \mathbb{N}$ tali che M risolve S e su ogni input $w \in S$, essa accetta in al più $f(|w|) + \epsilon$ passi;*

Definizione 17. *Diciamo che un problema S sta nella classe $DSPACE(f(n))$ (rispettivamente, $NSPACE(f(n))$ o $ASPACE(f(n))$) se esiste una macchina di Turing deterministica (rispettivamente, nondeterminstica o alternante) M ed un $\epsilon \in \mathbb{N}$ tali che M risolve S e su ogni input $w \in S$, essa accetta utilizzando al più spazio $f(|w|) + \epsilon$.*

[1] È semplice fornire un caso patologico in cui questo è falso: è sufficiente prendere una macchina che abbia la sola regola di spostare continuamente a destra un suo cursore e non terminare mai. È possibile anche progettare algoritmi che terminino ed in cui un cursore si sposti a destra di un numero di passi di qualche ordine di grandezza maggiore del numero di celle occupate; possiamo però tranquillamente trascurare questi casi "estremi", che non hanno alcun interesse teorico né pratico nell'affrontare i problemi decisionali.

La costante ϵ che compare in queste definizioni serve a formalizzare il fatto che ci interessa il comportamento *asintotico* di un algoritmo, ovvero di una macchina. Molti algoritmi che hanno, per esempio, costo quadratico nella lunghezza dell'input saranno caratterizzati da alcune costanti che renderanno il costo leggermente più che quadratico (spesso solamente per input piccoli); in questo modo stabiliamo che questi casi non ci preoccupano.

Diremo che un problema è *risolvibile* in tempo (spazio) $f(n)$ se appartiene a $DTIME(f)$, $NTIME(f)$ o $ATIME(f)$ (rispettivamente $DTIME(f)$, $NTIME(f)$ o $ATIME(f)$). Spesso sarà chiaro dal contesto se stiamo considerando il caso deterministico, nondeterministico o alternante; in caso contrario, specificheremo.

Si osservi che queste definizioni sono piuttosto prive di senso se $|S| < \infty$; questo non ci preoccupa perché tutti i problemi di cui ci occuperemo sono infiniti. Si noti anche che automaticamente abbiamo che se una funzione è *definitivamente* maggiore di un altra, tutte le classi di complessità relative alla prima includono le rispettive classi relative alla seconda.

Ciò detto, le classi di complessità che più ci interessano non sono definibili direttamente in questa forma e sono:

$$P = PTIME \stackrel{def}{=} \bigcup_{k\in\mathbb{N}} DTIME(n^k)$$

$$NP = NPTIME \stackrel{def}{=} \bigcup_{k\in\mathbb{N}} NTIME(n^k)$$

$$AP = APTIME \stackrel{def}{=} \bigcup_{k\in\mathbb{N}} ATIME(n^k)$$

e similmente:

$$PSPACE \stackrel{def}{=} \bigcup_{k\in\mathbb{N}} DSPACE(n^k)$$

$$NPSPACE \stackrel{def}{=} \bigcup_{k\in\mathbb{N}} NSPACE(n^k)$$

$$APSPACE \stackrel{def}{=} \bigcup_{k\in\mathbb{N}} ASPACE(n^k)\ .$$

Un'altra classe di complessità "celebre" è:

$$EXP \stackrel{def}{=} \bigcup_{k\in\mathbb{N}} DTIME(2^{n^k})$$

3.1.1 Relazioni tra le principali classi di complessità

Abbiamo presentato le macchine nondeterministiche come generalizzazioni di quelle deterministiche e quelle alternanti come generalizzazioni di quelle nondeterministiche. Ne deriva immediatamente che ogni classe deterministica è inclusa nella classe analoga nondeterministica, che a sua volta è inclusa nella classe analoga alternante; ad esempio:

$$PTIME \subset NPTIME \subset APTIME$$

$$PSPACE \subset NPSPACE \subset APSPACE$$

Supponiamo ora che una macchina di Turing risolva un problema in tempo $f(n)$: allora essa durante la sua esecuzione scriverà $f(n)$ simboli, e quindi utilizzerà al più $f(n)$ diverse celle di memoria.[2] Generalizzando questa analisi a tutte le macchina di Turing con questa proprietà, possiamo stabilire che

$$DTIME(f) \subset DSPACE(f) ,$$

$$NPTIME(f) \subset NPSPACE(f)$$

e

$$APTIME(f) \subset APSPACE(f) .$$

Queste relazioni si estendono naturalmente, modulo l'applicazione di semplici regole insiemistiche, alle classi di complessità che ci interessano maggiormente:

$$PTIME \subset PSPACE ,$$

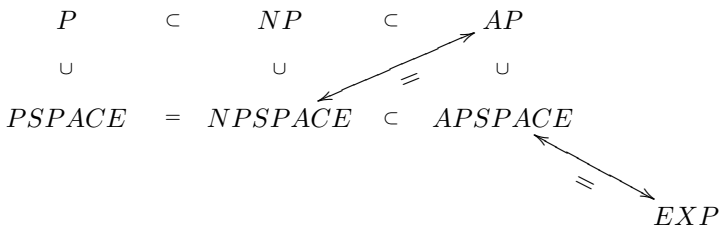
$$NPTIME \subset NPSPACE ,$$

$$APTIME \subset APSPACE .$$

A completare tale panorama, anticipiamo alcuni risultati noti:

- il capitolo 5 culminerà nel risultato finale $PSPACE = APTIME$,
- il *teorema di Savitch* (teorema 14) dimostra che $PSPACE = NPSPACE$, ovvero che il nondeterminismo risulta essere una generalizzazione che non aumenta drasticamente l'efficienza delle macchina di Turing per quel che riguarda lo spazio;
- il *teorema di gerarchia* (teorema 16) dimostra che $PTIME \subsetneq EXPTIME$;
- invece, si ha $APSPACE = EXPTIME$ (ed è probabilmente questo il risultato che mette più in evidenza la potenza del paradigma alternante; si dimostra solitamente trovando un problema completo per entrambe le classi).

Ne risulta la seguente situazione:



Questo schema potrebbe essere esteso a destra, trattando le classi esponenziali nondeterministiche, alternanti e quelle riguardanti lo spazio, e poi le classi doppiamente esponenziali e così via, ottenendo una struttura quasi periodica,

[2] Stiamo leggermente barando, dato che una macchina a k nastri potrà occupare $k * f(n)$ celle di memoria in tempo $f(n)$. Tuttavia, il teorema di accelerazione lineare (teorema 5) "legalizzerà" questa nostra trasgressione, dimostrando che nello studio delle classi le costanti moltiplicative sono trascurabili.

dato che molti risultati sulle classi polinomiali sono in realtà casi particolari di risultati più generali. Tuttavia, sulle classi che abbiamo scelto di considerare, lo schema rappresenta sostanzialmente *tutto ciò che si sa attualmente*. Infatti, della catena

$$P \subset NP \subset AP \subset EXP\ ,$$

è dimostrato solo, in virtù del teorema di gerarchia: $P \neq EXP$; si congettura che tutte le altre inclusioni siano strette, ma questo resta un problema aperto. In particolare, stabilire se $P = NP$ viene considerato il problema aperto più importante dell'intera teoria della complessità, tanto che è l'unico nato in campo informatico ad essere inserito tra i 7 "Millennium Problems", per la risoluzione di ognuno dei quali il Clay Institute di Cambridge ha offerto un premio di un milione di dollari.

3.1.2 Classi complementari

Dato ora un problema S in un alfabeto Σ, definiamo in modo ovvio il *complementare* di S:

$$S^C \stackrel{def}{=} \Sigma^* \setminus S$$

e finalmente, data una qualsiasi classe di complessità $\mathcal{C}$, definiamo la classe complementare:

$$co - \mathcal{C} \stackrel{def}{=} \left\{P | P^C \in \mathcal{C}\right\}$$

Vedremo più avanti che su alcune classi complementari esistono alcuni risultati interessanti.

3.2 Risultati accessori sulle macchine di Turing

Le definizioni 16 e 17, al variare della funzione considerata, forniscono una griglia in cui collocare i problemi decisionali. Siccome vogliamo considerare l'appartenenza ad una classe di complessità come una misura della difficoltà di risoluzione di un problema, sarà importante riuscire a stabilire un *ordine* su di esse; a tal fine, forniamo alcuni risultati generali che ci semplificheranno notevolmente l'opera.

Teorema 4. *Se un problema S è risolvibile da una macchina di Turing a k nastri M in tempo $f(n)$, allora esiste una macchina di Turing mononastro M' che lo risolve in tempo $f(n)^2 + n$ utilizzando lo stesso spazio.*

La dimostrazione di questo risultato è riportata in Appendice B.2.

Teorema 5 (Teorema di accelerazione lineare)**.** *Sia S un problema risolubile in tempo $c * f(n)$ e spazio $c * g(n)$ (per una certa $c \in \mathbb{N}$) da una certa macchina di Turing (in un dato paradigma: deterministico, nondeterministico o alternante). Allora esiste una macchina (nello stesso paradigma) che risolve S in tempo $f(n) + 2 * n$ e spazio $g(n) + n$.*

Dimostrazione. Sia

$$M = (\Sigma, Q, q_i, q_f, \delta)$$

una macchina di Turing deterministica a 1 nastro (la dimostrazione si applica con adattamenti minimi ai casi nondeterministico e alternante, nonché ad una

macchina multinastro) che risolve S in tempo $c * f(n)$ e spazio $c * g(n)$.

Vogliamo costruire una macchina $\bar{M}$ che effettui i passi di M a gruppi, in un solo passo. Vedremo che questa "emulazione" sarà strutturata in modo tale che per effettuare ogni gruppo di operazioni (di M) siano necessari 6 passi (di $\bar{M}$): faremo quindi in modo che ogni gruppo sia composto da $6 * c$ operazioni di M, ovvero che alla fine il tempo di esecuzione della nuova macchina sia $\frac{c*f(n)}{6*c} = f(n)$. Tuttavia, vedremo che questa nuova macchina avrà bisogno di un input codificato in modo particolare, e sarà proprio la ricodifica dell'input a richiedere tempo $2 * n$ e spazio n (più precisamente $\frac{n}{6*c}$), che andranno a sommarsi al costo generale.

Sia quindi $d = 6 * c$; $\bar{M}$ sarà una macchina a due nastri:

$$\bar{M} = (\bar{\Sigma}, \bar{Q}, \bar{q}_i, \bar{q}_f, \bar{\delta})$$

definita come segue:

$$\bar{\Sigma} = \Sigma \cup \Sigma^d$$

(in particolare ci risulterà comodo indicare con $\bar{\sqcup}$ il simbolo $(\underbrace{\sqcup, \cdots \sqcup}_{d})$

$$\bar{Q} = \left(\{\rightarrow\} \times \Sigma^{d-1}\right) \cup \{\leftarrow\} \cup (\{0, \dots 5\} \times Q \times \bar{\Sigma}^2 \times \{-d, -d+1 \dots 2d-1\}) \cup q_f$$

$$\bar{q}_i = (\rightarrow, \sqcup \cdots \sqcup)$$

$$\bar{q}_f = q_f$$

Il particolare insieme degli stati si spiega in base al fatto che $\bar{M}$ procederà in varie fasi;

- durante la codifica dell'input, i suoi stati apparterranno all'insieme

  $$Q_1 = \left(\{\rightarrow\} \times \Sigma^{d-1}\right)$$

  Ogni stato di questa forma è capace di "memorizzare" fino a $d-1$ simboli di Σ letti.

- dopo la codifica dell'input, la macchina entrerà nello stato $\leftarrow$ per riportare i cursori ad inizio nastro

- a questo punto, la macchina comincerà l'emulazione vera e propria, negli stati appartenenti a

  $$Q_2 = \{0, \dots 5\} \times Q \times \bar{\Sigma}^2 \times \{-d, \dots 2d-1\} \ .$$

  Ognuno degli stati di questa forma memorizza:

  1. un numero da 0 a 5 che scandirà i passaggi all'interno dell'esecuzione di un gruppo di istruzioni. Abbiamo detto infatti che ogni gruppo verrà eseguito in 6 passi: vedremo che ognuno compirà un'operazione diversa, e questo numero serve appunto ad indicare l'operazione da compiere

2. uno stato di M: siccome infatti $\bar{M}$ la emula, dovrà in particolare memorizzare lo stato in cui si trova
3. due simboli in $\bar{\Sigma}$, che serviranno per mantenere informazioni riguardanti le celle adiacenti a quella che sta leggendo il cursore
4. un numero da $-d$ a $2d-1$ che servirà a precisare la posizione del cursore

La codifica dell'input consisterà solo nel concentrare in un solo simbolo di $\bar{\Sigma}$ d simboli di Σ; negli stati in Q_1, la macchina si comporterà quindi come segue:

- legge un simbolo σ dal nastro di input
- se lo stato ha ancora "spazio", ovvero se vi figura almeno un $\sqcup$, passa in uno stato che aggiunga in memoria il simbolo letto; ad esempio, da uno stato
$$(\rightarrow, \sigma^1, \dots \sigma^5, \underbrace{\sqcup \cdots \sqcup}_{d-6}) ,$$
e leggendo un simbolo σ, si sposterà in uno stato
$$(\rightarrow \sigma^1, \dots \sigma^5, \sigma, \underbrace{\sqcup \cdots \sqcup}_{d-7}) .$$
- se invece lo stato è "pieno", ovvero vi figurano già $d-1$ simboli $\sigma^1 \dots \sigma^{d-1}$ e nessun $\sqcup$, scrive sul secondo nastro il simbolo
$$(\sigma^1, \dots \sigma^{d-1}, \sigma) ,$$
sposta a destra sia il primo che il secondo cursore e ricomincia questo processo, rientrando cioè nello stato $(\rightarrow, \underbrace{\sqcup \cdots \sqcup}_{d-1})$
- Se il cursore del primo nastro arriva ad una casella vuota, invece, esso scrive nel secondo nastro i simboli memorizzati nello stato, nella forma
$$(\sigma^1, \sigma^2 \dots \sigma^l, \sqcup, \cdots \sqcup)$$
ed entra nello stato $\leftarrow$, in cui la macchina semplicemente riporterà ad inizio stringa i due cursori mossi.

Da questo punto in avanti, la macchina lavorerà solo sul secondo nastro, contenente $\frac{n}{d}$ simboli, che rappresentano una codifica di tutto il contenuto del primo nastro e che fungeranno quindi da input.

Abbiamo detto che si tratterà di effettuare d passi di M in un solo passo di $\bar{M}$: per fare ciò, osserviamo che, in generale, ogni volta che di una macchina di Turing si conosce lo stato, la funzione di transizione, i d simboli a destra del cursore e i d (o meno, se si è vicini all'inizio del nastro) a sinistra, è possibile prevedere il comportamento della macchina per i d passi successivi, semplicemente componendo opportunamente δ per d volte di seguito.

È quindi possibile realizzare una tabella a due entrate che, dati $2d+1$ simboli (in Σ) ed uno stato (in Q), indichi lo stato in cui sarà M dopo d passi e

ovviamente i simboli che avrà scritto. È inoltre possibile modificare tale tabella in modo tale che, dati $3d$ simboli consecutivi, uno stato ed un numero α da 0 a $d-1$, essa indichi dove si troverà la macchina dopo d passi se avviata con il cursore in posizione $d+\alpha$ all'interno della stringa dei simboli conosciuti, nonché la nuova configurazione dei simboli (osserviamo che o i primi d o gli ultimi d saranno invariati).

Ciò detto quindi, non appena il secondo cursore sarà stato riportato ad inizio nastro, M' entrerà dallo stato $\leftarrow$ nello stato

$$(1, q_i, \bar{\sqcup}, \bar{\sqcup}, 1) ,$$

ed effettuerà quindi ripetutamente le seguenti operazioni

1. trovandosi in uno stato $(1, q, \bar{\sqcup}, \bar{\sqcup}, \alpha)$: si sposta a destra riscrivendo il simbolo che legge, entra nello stato $(2, q, \bar{\sqcup}, \bar{\sqcup}, \alpha)$

2. in uno stato $(2, q, \bar{\sqcup}, \bar{\sqcup}, \alpha)$: "memorizza" il simbolo σ che legge, entrando nello stato
$$(3, q, \sigma, \bar{\sqcup}, \alpha)$$
e spostandosi a sinistra. Questo passaggio è necessario perché nel corso dei d passi di M che vogliamo prevedere, il cursore può spostarsi a destra e quindi dobbiamo sapere che simboli leggerebbe.

3. in uno stato $(3, q, \sigma, \bar{\sqcup}, \alpha)$: si sposta ancora a sinistra, entrando nello stato $(4, q, \sigma, \bar{\sqcup}, \alpha)$

4. in uno stato $(4, q, \sigma, \bar{\sqcup}, \alpha)$: legge un simbolo σ', lo memorizza, si sposta a destra, entra in uno stato $(5, q, \sigma, \sigma', \alpha)$. Questo passaggio si giustifica in modo simile al 2: è necessario conoscere i simboli che il cursore di M leggerebbe se si spostasse di alcuni passi a sinistra. Notiamo che a questo punto il cursore è collocato nella stessa posizione in cui era collocato al passo 1.

5. qui avviene il nucleo della computazione: per stabilire la regola relativa agli stati della forma $(5, q, \sigma, \sigma', \alpha)$, è sufficiente consultare la tabella a due entrate descritta precedentemente. Lo stato contiene tutte le informazioni sui simboli letti a destra e a sinistra, per cui sarà possibile dedurne:

   - in quale stato q' passerebbe M in d passi
   - quale simbolo $\tau \in \Sigma^d$ scrivere (ovvero come M modificherebbe i d simboli adiacenti al cursore)
   - se è necessario modificare σ o σ' (i simboli a destra e a sinistra), e in tal caso
   - con quale simbolo τ' è necessario sovrascriverlo
   - di quanti passi β a destra (β potrà essere negativo) M sposterebbe il cursore;

   se è effettivamente necessario sovrascrivere σ, la macchina entrerà nello stato $(6, q', \tau', \bar{\sqcup}, \alpha + \beta - 6)$ e si sposterà a destra; simmetricamente, se è necessario sovrascrivere σ'; entrerà nello stato $(6, q', \tau', \bar{\sqcup}, \alpha + \beta + 6)$ e si sposterà a sinistra; altrimenti, entrerà nello stato $(6, q', \bar{\sqcup}, \alpha + \beta)$.

6. in uno stato $(6, q', \bar{\sqcup}, \bar{\sqcup}, \alpha + \beta)$, in cui tutti i simboli da scrivere saranno stati scritti, rimarrà soltanto da riposizionare il cursore sulla cella giusta: a tale scopo utilizzeremo le informazioni memorizzate nell'ultimo campo dello stato. Se $\alpha + \beta$ è minore di 0 o maggiore di 5, il cursore si sposterà rispettivamente a sinistra o a destra, mentre M' ed entrerà nello stato

$$(6, q', \bar{\sqcup}, \bar{\sqcup}, \alpha + \beta \ (mod\ 6)) \ .$$

   Se invece lo stato è della forma $(6, q', \tau', \bar{\sqcup}, \alpha + \beta)$, ovvero resta ancora un simbolo da scrivere, essa scriverà il simbolo τ', poi seguirà lo stesso criterio per spostarsi.

7. come unica eccezione al caso 5, se le condizioni sono tali che M, in 6 o meno passi, raggiunge q_f, allora anche M' entra direttamente nello stato q_f.

Ogni 6 passi, M' avrà effettuato d passi di M: in totale quindi, se M accetta in $c * f(n)$ passi, M' accetterà in $6 * \frac{c}{d} f(n) = f(n)$ passi. Per quanto riguarda lo spazio, abbiamo dimostrato un risultato ancora più forte di quello desiderato, dato che M' utilizza $\frac{c}{d} g(n) = \frac{g(n)}{6}$ celle di memoria.

La dimostrazione si riferisce solo al caso deterministico, ma si estende con modifiche minime ai casi nondeterministico ed alternante, ed è facilmente generalizzabile a macchine multinastro (nella dimostrazione, ad esempio, si può supporre di raddoppiare il numero di nastri. □

Un ultimo risultato che ci tornerà utile per semplificare le dimostrazioni è il seguente:

Teorema 6. *Ogni problema risolubile da una macchina M in tempo $f(n)$ e spazio $g(n)$ può essere codificato nell'alfabeto $\{0, 1\}$ e risolto in tempo $h * f(n)$ e spazio $h * g(n)$, data una opportuna costante h, da una macchina M' binaria, ovvero avente alfabeto $\{0, 1\}$.*

Questo teorema, che potremmo quasi chiamare di *decelerazione lineare*, può essere visto come un risultato in senso contrario a quello di accelerazione lineare: così come si può accelerare una macchina semplicemente espandendone l'alfabeto, è possibile ridurne l'alfabeto a costo di rallentarla di un fattore costante. La dimostrazione si basa sempre su una sorta di emulazione e perciò sarà analoga a quella appena vista.

Dimostrazione. Sia M la macchina data, e siano Σ il suo alfabeto e Q il suo insieme degli stati. Sia $k \in \mathbb{N}$ tale che $2^k > |\Sigma|$.
Costruiremo una macchina M' che in k passi emulerà un singolo passo di M. Possiamo associare biunivocamente ogni simbolo in Σ ad una stringa in $\{0, 1\}^*$, e quindi supporre che la stringa di input di M' identifichi un elemento di Σ^*.

Abbiamo già visto nella dimostrazione precedente come un opportuno insieme degli stati permetta di "memorizzare" temporaneamente dei simboli; nel nostro caso, l'insieme degli stati Q' di M' sarà della forma:

$$Q \times \{l, s, m\} \times \{0, 1\}^{k-1} \times \{\rightarrow, \downarrow, \leftarrow\} \ ,$$

per memorizzare contemporaneamente lo stato di M che si sta emulando, la modalità di operazione (vedremo dopo cosa si intende), $k-1$ simboli binari ed un movimento da effettuare. Definiamo inoltre i due stati speciali:

$$q'_i = (q_i, l, \underbrace{0, \ldots 0}_{k-1})$$

$$q'_f = (q_f, s, \underbrace{0, \ldots 0}_{k-1}) \, .$$

La macchina effettuerà ciclicamente le seguenti operazioni:

1. immagineremo sempre che i nastri siano suddivisi in segmenti di k celle: all'inizio di ogni ciclo, ogni cursore si troverà all'inizio di un segmento e lo stato sarà
$$(q, l, \underbrace{0, \ldots 0}_{k-1}) \, ,$$

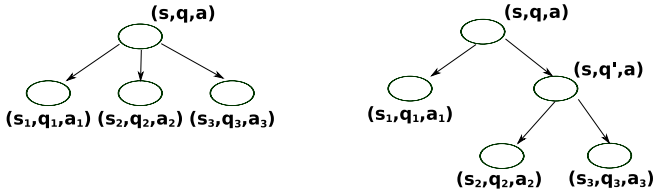
2. "l" sta per "lettura": i $k-1$ passi successivi consisteranno nel leggere e memorizzare nello stato $k-1$ simboli (spostandosi quindi ad ogni passo a destra): alla fine la macchina si troverà in uno stato:
$$(q, l, \sigma_1, \ldots \sigma_{k-1}) \, ,$$
con $\sigma_i \in \{0, 1\}$. L'interpretazione dei blocchi di k celle è che ognuno codifica un singolo simbolo di M: di conseguenza, a questo punto, leggendo un nuovo simbolo (l'ultimo del blocco), la macchina avrà sufficienti informazioni per emulare un passo di M (le regole esatte di M' saranno fissate in base ad una tabella simile a quella vista nella dimostrazione precedente). Individuato quindi il simbolo che M avrebbe scritto, M' dovrà procedere a scriverne la codifica; più precisamente, ne scriverà immediatamente l'ultimo simbolo e memorizzerà gli altri, entrando in uno stato:
$$(q, s, \sigma'_1, \ldots \sigma'_{k-1}, \alpha)$$
dove α è l'elemento di $\{\rightarrow, \downarrow, \leftarrow\}$ specificato dalla legge di transizione di M.

3. "s" sta per "scrittura": la macchina dovrà adesso tornare verso sinistra, scrivendo mano a mano i simboli memorizzati nello stato (e mano a mano "smemorizzandoli", in modo da tenere il conto di quelli già scritti)

4. finalmente, tornata all'inizio del segmento, rimarrà solo - eventualmente - da spostare il cursore a destra o sinistra, di k passi, per cui la macchina entrerà nella modalità "m" (per "movimento"): la direzione in cui muoversi è indicata da α (se è $\downarrow$, non sarà affatto necessario muovere il cursore), e per contare i k passi la macchina può ad esempio memorizzare uno alla volta i simboli letti, fermandosi quando "la memoria è piena"

5. per completare la descrizione della macchina, è sufficiente aggiungere la regola che da qualsiasi stato della forma
$$(q_f, *, \cdots *)$$
la macchina si sposti nello stato q'_f, ovvero accetti.

Per simulare un passo di M, M' ha effettuato $3k$ passi: abbiamo quindi dimostrato il teorema, per $h = 3k$. □



(a) Regola originale (b) Regola modificata

Figura 3.1: Si può ridurre un albero generico ad un albero binario aggiungendo di al più $|Q| \times \lfloor \log h \rfloor$ stati, dove h è il grado massimo dell'albero.

3.2.1 Criteri per macchine alternanti

Per il paradigma alternante, enunciamo di seguito ancora tre criteri che ci permettono di ricondurre le macchina di Turing ad una sorta di forma standard; tali criteri ci torneranno utili nel paragrafo 5.1.2.

Grafo binario

Data una macchina alternante M e il suo grafo delle configurazioni G, possiamo supporre senza perdita di generalità che ogni vertice di G abbia esattamente 2 o 0 archi uscenti. Infatti, ogni volta che l'immagine tramite δ di un certo elemento di $Q \times \Sigma$ ha cardinalità $h > 2$, è possibile, ampliando opportunamente l'insieme degli stati, ridistribuirla su più ramificazioni binarie (Figura 3.2.1).

Viceversa, se $|\delta((\bar{q}, \bar{\sigma}))| = 1$, possiamo considerare δ' definita come δ con la sola differenza che

$$\delta'((\bar{\sigma}, \bar{q})) = \delta((\bar{\sigma}, \bar{q})) \cup \{(\bar{\sigma}, \bar{q}, \downarrow)\};$$

ciò non risolve l'eventuale caso[3] in cui $\delta((\bar{\sigma}, \bar{q})) = \{(\bar{\sigma}, \bar{q}, \downarrow)\}$; possiamo quindi considerare un nuovo stato $\hat{q}$ [4] ed utilizzare una funzione di transizione modificata δ'' tali che

$$\forall \sigma \;\; \delta''((\sigma, \hat{q})) = \{(\sigma, \hat{q}, \downarrow), (\bar{\sigma}, \bar{q}, \downarrow)\}$$

$$\delta''((\bar{\sigma}, \bar{q})) = \delta'((\bar{\sigma}, \bar{q})) \cup \{(\sigma, \hat{q}, \downarrow)\} \;\; (= \delta''((\bar{\sigma}, \hat{q}))) \; .$$

Applicando queste modifiche alla macchina, la lunghezza di una computazione viene moltiplicata al più per

$$\bar{h} = \max_{(\sigma,q) \in \Sigma \times Q} |\delta((\sigma, q))| \leq |\Sigma \times Q| \; ,$$

ovvero per una costante che possiamo trascurare.

Osserviamo che questa ragionamento si applica anche alle macchine nondeterministiche.

[3] Stupido, ma possibile

[4] ne basta uno solo, che può essere utilizzato per tutti questi (σ, q) "patologici" e che punti ad uno qualsiasi di essi.

Configurazioni senza figli

Similmente, possiamo supporre che gli unici vertici senza archi uscenti siano le configurazioni accettanti, ovvero possiamo far sì che ogni qualvolta $\delta((\sigma, q)) = \emptyset$ e $q \in Q_\exists$, utilizziamo invece la funzione di transizione modificata[5]:

$$\delta'((\sigma, q)) = \begin{cases} \{(\sigma, q, \downarrow), (\sigma', q', \downarrow)\} \text{ se } (\sigma, q, \downarrow) \in \delta((\sigma', q')) \\ \{(\sigma, q, \downarrow), (\sigma', q', \leftarrow)\} \text{ se } (\sigma, q, \rightarrow) \in \delta((\sigma', q')) \\ \{(\sigma, q, \downarrow), (\sigma', q', \rightarrow)\} \text{ se } (\sigma, q, \leftarrow) \in \delta((\sigma', q')) \ . \end{cases}$$

Alternanza degli stati

Infine, possiamo supporre senza perdita di generalità che ogni arco colleghi una configurazione universale (ovvero avente stato universale) ad una esistenziale (viceversa). Infatti, a costo di raddoppiare l'alfabeto, possiamo facilmente sdoppiare ogni regola che non rispetti tale proprietà in due regole, aggiungendo un passaggio intermedio. L'esistenza di possibili configurazioni intermedie *non ambigue* (non ancora utilizzate dalla macchina) è garantita proprio dal raddoppiamento dell'alfabeto.

Ad esempio, se abbiamo uno stato esistenziale q ed un simbolo σ tale che:

$$\delta(\sigma, q) = (\sigma', q', \alpha)$$

e q' è ancora uno stato esistenziale, possiamo aggiungere all'insieme degli stati un $\bar{q}$ universale e stabilire che:

$$\delta(\sigma, q) = (\sigma', \bar{q}, \alpha)$$

$$\delta(\sigma', \bar{q}) = (\sigma', q', \downarrow) \ .$$

L'ordine in cui abbiamo enunciato questi tre criteri non è casuale, ma ci permette di concludere che, data una qualsiasi macchina di Turing alternante, possiamo supporre che il suo grafo delle configurazioni rispetti tutte tre le condizioni, semplicemente applicando in questo ordine le tre modifiche descritte.

3.3 Combinazioni di macchine di Turing

Costruire esplicitamente una macchina di Turing è un lavoro faticoso, e soprattutto può essere molto faticoso per un essere umano interpretare la descrizione formale di una macchina di Turing complessa; sebbene entrambe le operazioni siano facilitate da un'opportuna scelta dei simboli dell'alfabeto e di quelli che indicano gli stati, nonché dall'omissione delle regole inutili (come spiegato nel paragrafo 2.4), uno stratagemma più efficiente per semplificare tale descrizione è scomporla in operazioni più ridotte, eseguibili ognuna da una data macchina di Turing più semplice, e quindi combinarle tra di loro per costruire una macchina che svolga l'algoritmo desiderato. Nel corso di questa tesi, parleremo di

[5] Questa definizione è imprecisa: intanto può verificarsi più di una condizione (in tal caso, però, qualsiasi opzione va bene); inoltre, può non verificarsene nessuna (tuttavia, è il caso delle configurazioni irraggiungibili, per le quali non ci interessa affatto modificare δ)

concatenare due macchine di Turing e di *richiamare* una macchina all'interno di un'altra; diamo di seguito una descrizione di ciò che intendiamo ed un accenno di giustificazione formale.

3.3.1 Concatenazione

Supponiamo di dover costruire una macchina di Turing M che effettui due operazioni ben distinte A e B; se entrambe le operazioni sono effettuabili da due macchine M_A e M_B, con funzioni di transizione rispettivamente δ_A, δ_B e che supporremo avere alfabeti Σ_A, Σ_B disgiunti (ad eccezione dei simboli $\triangleright, \sqcup$) ed insiemi degli stati Q_a, Q_B disgiunti (ad eccezione dei simboli q_i e q_f), possiamo definire M semplicemente come una macchina deterministica, nondeterministica o alternante (a seconda di quale sia il paradigma più potente tra quello di M_A e quello di M_B) dalle seguenti caratteristiche:

- l'insieme degli stati sarà semplicemente l'unione degli insiemi Q_A e Q_B, con l'aggiunta di uno stato speciale q_i' che rappresenterà il punto di congiunzione tra M_A e M_B,
- similmente, l'alfabeto sarà $\Sigma_A \cup \Sigma_B$;
- la funzione di transizione δ sarà semplicemente l'unione delle funzioni di transizione δ_A e δ_B, "corretta" come segue:
  - se lo stato è q_i, δ si comporta come δ_A, non come δ_B;
  - tutte le regole di δ_A che portano la macchina nello stato q_f saranno sostituite con regole che la portano in q_i' e riportano i cursori all'inizio dei nastri; in tal modo, dopo avere eseguito l'operazione A la macchina non terminerà ma si posizionerà pronta per effettuare l'operazione B
  - δ si comporterà sullo stato q_i' come δ_B su q_i; ciò completa la concatenazione, stabilendo che l'operazione B deve essere effettuata dopo A e non all'inizio

Si osserva facilmente che se le operazioni A e B venivano effettuate in tempo $f_A(n), f_B(n)$ ed utilizzando $g_A(n), g_B(n)$ celle di memoria, la loro concatenazione così effettuata impiegherà tempo al più $f_A(n) + g_A(n) + f_B(g_A(n))$ e spazio al più $\max(g_A(n), g_B(g_A(n)))$.

Chiameremo M *concatenazione* o *composizione* di M_A ed M_B.

3.3.2 Innesto

Talvolta, avremo bisogno di compiere un'operazione B, effettuabile da una macchina di Turing M_B con insieme degli stati Q_B, numerose volte all'interno di un certo algoritmo, o comunque non solo alla fine; in particolare, non vorremo che la nostra macchina M *termini* dopo avere effettuato B, ma che ad esempio ripeta ciclicamente B ed un'altra operazione: diremo che M *richiama* M_B.

La formalizzazione di questa particolare combinazione è leggermente più complicata della semplice concatenazione: vorremo solitamente che, mentre effettua B, M *ricordi* il suo stato precedente, e per questo motivo l'insieme degli

spazi sarà un prodotto diretto di Q_B e dell'insieme degli altri stati necessari a M; le regole di δ_B saranno riportate in δ con le opportune modifiche, che consisteranno principalmente nel cambiamento degli stati (non più in Q_B ma nel prodotto diretto appena menzionato) e talvolta nel cambiamento del nastro o dei nastri su cui la macchina opera.

La concatenazione e l'innesto di varie macchine saranno talvolta sottolineati, più spesso utilizzati implicitamente: ci capiterà infatti di descrivere una macchina di Turing non come un semplice insieme di regole ma come una lista di operazioni che compie *in sequenza*, sfruttando il fatto che, in virtù delle caratteristiche della funzione di transizione stessa, la macchina potrà trovarsi in certi stati solo in una data fase dell'esecuzione, che potrà quindi essere vista come una macchina a parte.

Infine, ci succederà di descrivere dei *cicli*, in cui una serie di operazioni viene ripetuta più volte, e alla fine di ognuno dei quali si *ricomincia*; in termini formali, questo vuol dire solitamente che la macchina si riporterà nello stato in cui era all'inizio e di conseguenza si comporterà poi di nuovo in modo analogo.

3.4 Riducibilità e completezza

Nello studio delle classi di complessità, hanno un ruolo fondamentale quei problemi che si dimostrano essere *più complicati* di certi altri, o addirittura di tutti quelli che si trovano in una certa classe di complessità; con "più complicati", intendiamo che sapendo risolvere uno, sappiamo automaticamente risolvere anche l'altro. La formalizzazione di questo discorso si basa sul concetto di *riducibilità*:

Definizione 18. *Diciamo che un problema S in un alfabeto Σ è* polinomialmente riducibile *ad un altro problema T in un alfabeto Σ' se esiste una macchina di Turing deterministica M (nel cui alfabeto siano inclusi sia Σ che Σ') tale che, data in input una qualsiasi stringa $s \in \Sigma^*$, essa scriva in tempo polinomiale su un suo nastro una stringa $t \in \Sigma'$ in modo che:*

- $s \in S \leftrightarrow t \in T$
- $|t| \leq |s|^k$*, dove k è un parametro fissato relativo alla macchina*

In tal caso, scriveremo $S \leq T$.

In altre parole, S è riducibile a T se esiste una funzione, calcolabile in modo efficiente, da Σ in Σ' tale che l'immagine di S sia contenuta in T e l'immagine di S^c sia contenuta in T^c.
L'importanza di questa definizione risiede nel fatto che, se esiste una macchina N che risolva T in tempo polinomiale, allora creando una nuova macchina che prima si comporti come M e quindi come N, otteniamo una macchina che risolve S.

Esistono poi problemi a cui è polinomialmente riducibile ogni problema di una certa classe:

Definizione 19. *Data una classe di complessità $\mathcal{C}$, un problema T si dice* $\mathcal{C}$-hard*, se:*

$$\forall S \in \mathcal{C}\ S \leq T$$

Definizione 20. *Data una classe di complessità $\mathcal{C}$, un problema T si dice* $\mathcal{C}$-completo*, o* completo per $\mathcal{C}$*, se esso è $\mathcal{C}$-hard ed appartiene a $\mathcal{C}$.*

Vedremo che nel corso di questa esposizione i problemi completi risulteranno molto importanti; infatti, data una classe $\mathcal{C}$ sufficientemente ampia, è sufficiente fornire un problema S che sia $\mathcal{C}$-completo per caratterizzare univocamente $\mathcal{C}$ (come l'insieme dei problemi riconducibili a S); ciò non è vero invece per un qualsiasi $S \in \mathcal{C}$ (non tutti i problemi di $\mathcal{C}$ saranno riconducibili ad S), né per un qualsiasi S che sia $\mathcal{C}$-hard (infatti ad esso saranno riconducibili problemi non necessariamente in $\mathcal{C}$ - ad esempio S stesso).

"Sufficientemente ampia" significa in sostanza che la definizione è adatta alle classi polinomiali o più ampie ma non, ad esempio, a quelle logaritmiche (un problema di complessità più che logaritmica può essere riducibile polinomialmente ad un problema di complessità logaritmica); è possibile definire altri tipi di riducibilità, che non saranno trattati in questo lavoro; per maggiori informazioni, si veda [5].

3.5 Funzioni costruibili e classi complementari

Abbiamo definito nel capitolo 3 il concetto di *classe complementare*. Un problema fondamentale nella teoria della complessità è se

$$NP = co - NP\ .$$

L'interesse riposto in questa congettura deriva da quello che è probabilmente il più importante problema aperto nell'ambito della complessità:

$$P = NP?$$

P è infatti un sottoinsieme sia di NP che di $co - NP$, e si può dimostrare che se i due insiemi più grandi fossero identici, essi collasserebbero entrambi in P. Siccome la congettura largamente diffusa è che $P \neq NP$, si congettura di conseguenza che $NP \neq co - NP$.

3.5.1 Funzioni costruibili

Ci sono altre classi complementari per cui la situazione è molto meno misteriosa; prima di trattarle però, è necessario dare una definizione:

Definizione 21. *Una funzione $f : \mathbb{N} \to \mathbb{N}$ si dice* costruibile *se esiste una macchina di Turing deterministica che, ricevendo in input un numero m (rappresentato opportunamente, ad esempio con una sequenza di m simboli) sia in grado di scrivere su un altro nastro esattamente $f(m)$ simboli, in tempo lineare in $f(m)$.*

La funzione identità è banalmente costruibile. Si vede facilmente che è costruibile anche la funzione di moltiplicazione per una qualsiasi costante.
In realtà si può dimostrare che la stragrande maggioranza delle funzioni intere a cui si può pensare sono costruibili: se tuttavia diamo questa definizione, è perché effettivamente esistono casi patologici di funzioni calcolabili non costruibili.[6]

3.5.2 Classi complementari deterministiche

Il concetto di funzione costruibile ci risulta utile per quelle macchine che operano in tempo o spazio prefissato: è possibile infatti aggiungere degli "orologi" o dei "contatori" che controllino che il limite di spazio o tempo non venga superato, e fermino la macchina in caso contrario.

Più precisamente, sia data una macchina di Turing deterministica M che riconosce un problema S in tempo $f(n)$: se f è costruibile, possiamo progettare una macchina M' con un nastro aggiuntivo rispetto ad M che innanzitutto scriva su questo ultimo nastro $f(n)$ simboli, quindi si comporti come M ma spostando, ad ogni passo, l'ultimo cursore a sinistra; stabiliamo inoltre che nei casi in cui M si sposta nello stato q_f, M' si sposti in uno stato pozzo, mentre nel solo caso in cui l'ultimo cursore legge il simbolo $\triangleright$, ovvero torna all'inizio del nastro, M' accetti. Questa macchina riconoscerà esattamente il problema *complementare* a S, e lo farà in tempo lineare in $f(n)$, dato che il cursore dell'ultimo nastro effettua una sorta di "conto alla rovescia" che rende accettanti in tempo $f(n)$ tutte e sole le computazioni che non lo erano.

Similmente, se M riconosce S in spazio $g(n)$, con g costruibile, possiamo creare M' con un nastro aggiuntivo che innanzitutto scriva su questo ultimo nastro $g(n)$ simboli, quindi si comporti come M ma spostando l'ultimo cursore a sinistra ogni volta che un simbolo $\sqcup$ è sovrascritto con un simbolo diverso in uno degli altri nastri, e a destra ogni volta che un altro simbolo è sovrascritto con $\sqcup$, ovvero tenendo traccia dello spazio totale occupato. In realtà, se su più nastri viene contemporaneamente sovrascritto un $\sqcup$ noi vorremo spostare l'ultimo cursore a sinistra di diverse posizioni (e viceversa se più simboli $\sqcup$ vengono scritti contemporaneamente), ma questo si risolve facilmente aggiungendo alcuni stati accessori nei quali il cursore effettui uno spostamento alla volta (osserviamo infatti che in questo caso non ci interessa particolarmente il tempo di esecuzione).

Infine, non stiamo considerando il caso in cui un simbolo $\sqcup$ non sia finale, ovvero abbia alla sua destra simboli diversi da $\sqcup$; tuttavia, si verifica facilmente che restringendosi alle macchine in cui ciò non accade non si ha nessuna sostanziale limitazione sui problemi che si può risolvere, né su tempo e spazio impiegati per risolverli. Finalmente, possiamo quindi supporre, come sopra, che nei casi in cui M si sposta nello stato q_f, M' si sposti in uno stato pozzo, e se invece l'ultimo cursore di M' legge il simbolo $\triangleright$, M' accetti. Questa macchina, di nuovo, riconosce il problema *complementare* a S, e lo fa in spazio lineare in $g(n)$.

In virtù del teorema di accelerazione lineare, da questi due risultati possiamo dedurre in generale il seguente:

[6] Il controesempio classico è la *funzione di Ackermann*.

Teorema 7. *Sia f una funzione costruibile: allora*

$$TIME(f(n)) = co\text{-}TIME(f(n))$$

$$SPACE(f(n)) = co\text{-}SPACE(f(n))$$

Dimostrazione. Consideriamo un qualsiasi problema in una di queste classi: abbiamo visto che il suo complementare sarà risolubile utilizzando la stessa quantità di spazio e tempo, collocandosi quindi anche esso nella stessa classe. Dalla definizione di classe complementare, deriva direttamente il risultato cercato. □

Corollario 8.

$$P = co\text{-}P$$

$$PSPACE = co\text{-}PSPACE$$

Dimostrazione. P è definito come unione di una successione di classi, per ognuna delle quali vale il teorema 7; il complementare[7] di P non è altro che l'unione delle loro classi complementari, che però abbiamo visto essere uguali alle classi stesse. □

3.5.3 Classi complementari nondeterministiche ed alternanti

Al caso deterministico abbiamo già accennato, osservando che la congettura $NP =$ co-NP rappresenta un problema aperto. Ci limitiamo qui ad accennare che invece $NPSPACE = co - NPSPACE$, e di conseguenza l'analisi delle più importanti classi complementari nondeterministiche è esaurita.

Può essere utile però osservare *quale aspetto* della dimostrazione del teorema 7 non funziona per una macchina M nondeterministica: in effetti è possibile realizzare una macchina M' uguale ma con l'aggiunta di un contatore: in seguito, ogni ramo della computazione di M' su un dato input sarà per l'appunto scandito dal "conto alla rovescia". Tuttavia, se semplicemente scambiassimo lo stato q_f con uno stato pozzo, avremmo che *tutti* i rami accettanti diverrebbero non accettanti, e viceversa. M accetta se almeno un ramo accetta, mentre M' accetterebbe se almeno un *suo* ramo accettasse, ovvero se almeno un ramo di M *non* accettasse... ovvero M' non riconoscerebbe affatto il problema complementare.

Per trattare il caso alternante, invece, i contatori del caso deterministico ci tornano nuovamente utili.

Sia infatti M una macchina alternante che riconosca in spazio (tempo) $f(n)$ un problema S (supponiamo - abbiamo visto alla fine del paragrafo 2.3 che è lecito farlo - che essa non abbia alcuno stato q_f): possiamo aggiungere un contatore, come descritto nel caso deterministico, che tenga traccia dello spazio (del tempo) e quindi operare come segue:

- innanzitutto, sfruttando il meccanismo dei contatori, trasformare M in una macchina M' in cui ogni ramo che duri più tempo (occupi più spazio) di $f(n)$ venga inviato in uno stato pozzo (esistenziale e senza figli) $\bar{q}$;

[7] Ricordiamo che il significato di "complementare" nel caso delle classi non è quello puramente insiemistico ma quello definito in 3.1.2.

- da M' otteniamo quindi una macchina M'' semplicemente scambiando l'insieme degli stati universali con quello degli stati esistenziali.

Osserviamo che, dato un qualsiasi input w, l'albero A_w che descrive la computazione di M' su w è finito. Allora si può verificare per induzione che M'' effettivamente accetta tutti e soli gli input non accettati da M': la dimostrazione consiste nel mostrare che *ogni* configurazione accettante diventa non accettante, e viceversa:

- sia α una configurazione senza figli: se α era universale, cioé accettante, in M', in M'' è esistenziale e quindi non accettante, e viceversa
- l'ipotesi induttiva è che se una configurazione α ha rami uscenti di lunghezza al più $k-1$, allora essa è accettante in M'' se e solo se non lo è in M': il passo induttivo è una semplice applicazione delle leggi di De Morgan (se una configurazione β ha rami uscenti di lunghezza al più k, ogni configurazione figlia avrà rami uscenti di lunghezza al più $k-1$ e su di essa quindi varrà l'ipotesi).

In particolare, la configurazione associata alla radice di A_w sarà accettante in M' se e solo se non lo è in M'', ovvero M'' riconoscerà il problema complementare a quello riconosciuto da M'. Ma M' riconosce esattamente lo stesso problema di M, e di conseguenza abbiamo creato una macchina che riconosce S^c.

Siccome questo vale per ogni macchina alternante M, abbiamo, come nel caso deterministico, il seguente:

Teorema 9. *Sia f una funzione costruibile: allora*

$$ATIME(f(n)) = co\text{-}ATIME(f(n))$$

$$ASPACE(f(n)) = co\text{-}ASPACE(f(n))$$

Dimostrazione. Identica alla dimostrazione del teorema 7. □

Corollario 10.

$$AP = co\text{-}AP$$

$$APSPACE = co\text{-}APSPACE$$

Dimostrazione. Identica alla dimostrazione del corollario 8. □

Capitolo 4

SAT e QSAT

4.1 Formule booleane

La logica booleana è un sistema formale caratterizzato da un insieme di simboli di variabile (A, B, y, x_1 ...), alcuni operatori ($\neg, \wedge, \vee, \rightarrow \ldots$) e le parentesi, che servono a determinare la precedenza degli operatori. Definire in modo rigoroso la logica proposizionale equivale semplicemente a definire l'insieme delle *formule ben formate*:

1. i simboli di variabile sono formule ben formate, così come i valori di verità V e F;
2. se φ e ψ sono formule ben formata, lo sono anche $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$ e (φ);
3. tutto ciò che non ricade nei casi 1 e 2 non è una formula ben formata.

Intrinsecamente legato al concetto di formula booleana è quello della sua *valutazione*: attribuendo un valore in $\{0, 1\}$ ad ogni variabile in gioco, e seguendo le regole stabilite nelle tavole di verità degli operatori:

$\wedge$	V	F
V	V	F
F	F	F

$\vee$	V	F
V	V	V
F	V	F

$\rightarrow$	V	F
V	V	F
F	V	V

$\neg$	
V	F
F	V

ad ogni possibile formula viene associato, per induzione sulla sua complessità, un valore di verità.

4.1.1 Forme normali

Si dice *letterale* una singola variabile o una singola costante, eventualmente preceduta da una negazione.

Si dice *clausola* una sequenza di letterali collegati tra loro da operatori $\vee$.
Si dice *implicante* una sequenza di letterali collegati tra loro da operatori $\wedge$.
Una formula si dice in *forma normale congiunta* se è una congiunzione di clausole, e in *forma normale disgiunta* se è una disgiunzione di implicanti. *Ogni* formula può essere portata in forma normale congiunta ed in forma normale disgiunta seguendo alcune semplici regole (definite per induzione sulla struttura):

- se la formula è una singola variabile, è già in forma normale congiunta e disgiunta
- se la formula è della forma $\neg\varphi$, per metterla in forma normale congiunta è sufficiente prendere φ, trasformarla in forma normale disgiunta e sostituire ogni $\vee$ con un $\wedge$, ogni $\wedge$ con un $\vee$, ogni letterale negato con un letterale non negato e viceversa; il caso della forma normale disgiunta è perfettamente simmetrico
- se la formula è della forma $\varphi \vee \psi$,
  - per portare la formula in forma normale disgiunta, è sufficiente portarvi φ e ψ,
  - per portarla in forma normale congiunta, supponiamo che φ e ψ lo siano già:
    $$\varphi = \bigwedge_i \varphi_i \ , \quad \psi = \bigwedge_j \psi_j \ ,$$
    allora la formula ricercata è
    $$\bigwedge_{i,j} \varphi_i \vee \psi_j \ ;$$
- il caso $\varphi \wedge \psi$ è perfettamente simmetrico,
- il caso $\varphi \to \psi$ è riconducibile a quelli precedenti.

Osserviamo che la forma normale congiunta di una formula composta dalla disgiunzione di n implicanti conterrà 2^{2n-2} clausole, e similmente la forma normale disgiunta della congiunzione di n clausole.

4.2 Valutazione di una formula booleana

Si verifica facilmente che stabilire se una data stringa di simboli in

$$\{\vee, \wedge, \to, \neg, (,), 0, 1\}$$

sia una formula ben formata è un problema piuttosto semplice: si tratta in sostanza di verificare che le parentesi siano ben bilanciate, che tra due "sottoformule ben formate" vi sia sempre un operatore binario, che ogni operatore binario si collochi tra due sottoformule ben formate e che l'operatore $\neg$ si collochi prima, e non dopo, di una sottoformula ben formata: tutte queste verifiche si possono effettuare in tempo lineare.

Per la nostra analisi, ci interessa particolarmente l'aspetto della *valutazione* di una formula booleana. In particolare, si osserva che la valutazione di una

formula booleana senza variabili (contenenti solo le costanti V e F, gli operatori e le parentesi) si può effettuare in tempo quadratico nella lunghezza della formula. È sufficiente infatti procedere effettuando alcune sostituzioni:

- tutte le stringhe della forma:

$$[V]^i + [\vee] + [F]^j$$

$$[F]^i + [\vee] + [V]^j$$

$$[V]^i + [\vee] + [V]^j$$

$$[V]^i + [\wedge] + [V]^j$$

$$[V]^i + [\rightarrow] + [V]^j$$

$$[F]^i + [\rightarrow] + [V]^j$$

$$[F]^i + [\rightarrow] + [F]^j$$

$$[\neg] + [F]^i$$

$$[(] + [V]^i + [)]$$

  verranno sostituite con una stringa di V, e viceversa tutte le stringhe della forma:

$$[V]^i + [\wedge] + [F]^j$$

$$[F]^i + [\wedge] + [V]^j$$

$$[F]^i + [\wedge] + [F]^j$$

$$[F]^i + [\vee] + [F]^j$$

$$[V]^i + [\rightarrow] + [F]^j$$

$$[\neg] + [V]^i$$

$$[(] + [F]^i + [)]$$

  con una stringa di F.

- tale sostituzione verrà ripetuta fintanto che nel nastro non comparirà una stringa di sole V; se ciò accadrà, la macchina accetterà.

Si dimostra facilmente che saranno necessarie al più n scansioni e che ogni scansione ha costo lineare in tempo; di conseguenza il costo globale dell'algoritmo è quadratico.

4.3 SAT

L'aspetto della valutazione è fondamentale nel problema SAT (dall'inglese "*Boolean satisfiability*", ovvero "*soddisfacibilità booleana*"): data una formula booleana, è possibile assegnare ad ogni variabile che vi compare un valore di verità tale che la formula nella sua interità sia vera?

Per rappresentare in modo più sintetico il problema, utilizzeremo una notazione particolare, in cui indicheremo le variabili con dei numeri scritti in binario e racchiusi tra parentesi; ad esempio, la formula

$$(A \vee B) \rightarrow C$$

(che - osserviamo en passant - è soddisfacibile[1]) sarà rappresentabile come

$$((0) \vee (1) \rightarrow (10))$$

Possiamo a questo punto dare la seguente:

Definizione 22. *Dato l'insieme di simboli*

$$\Sigma = \{\vee, \wedge, \rightarrow, \neg, (,), 0, 1\} ,$$

chiamiamo SAT il sottoinsieme di Σ^* contenente tutte e sole le formule ben formate tali che assegnando un opportuno valore alle variabili esse risultino vere.

SAT è un problema fondamentale nella teoria della complessità, principalmente in virtù del teorema che segue, dimostrato da Cook nel 1971:

Teorema 11 (Teorema di Cook [3])**.** *SAT è un problema completo per NP*

Dimostrazione. Dimostriamo che SAT $\in$ NP: definiremo una macchina M a 3 nastri, di alfabeto rispettivamente:

$$\{\exists, \forall, \neg, (,), \vee, \wedge, V, F, 0, 1\}$$

$$\{0, 1\}$$

$$\{V, F\}$$

Supponiamo che nel primo nastro, quello di input, le formule vengano immesse nel formato descritto sopra con la sola differenza che le stringhe binarie siano scritte al contrario: ad esempio, la formula

$$(A \vee B) \rightarrow C$$

sarà ora scritta come

$$((0) \vee (1) \rightarrow (01)) .$$

Innanzitutto la macchina controllerà che la formula sia ben formata: abbiamo visto che questa operazione può essere effettuata anche da una macchina deterministica in tempo quadratico. Se la formula *non* è ben formata, la macchina rifiuterà l'input (si sposterà in un pozzo).
Altrimenti, effettuerà le seguenti operazioni:

[1] Non fosse altro che per il fatto che *ogni formula* in cui ogni variabile occorre al più una volta è soddisfacibile.

1. sul secondo nastro, scriverà inizialmente [0] ed incrementerà poi sempre il numero, scritto in forma binaria invertita: esso indicherà la variabile che stiamo sostituendo

2. ogni volta che incrementa il contenuto del secondo nastro, scriverà non deterministicamente (in due rami separati), sul terzo nastro, V ed F

3. andrà quindi a sostituire, nel primo nastro, ogni stringa della forma (α), dove α è il contenuto del secondo nastro, con una stringa della forma F^i se sul terzo nastro compare una F e V^i se vi compare una V; così facendo, avrà effettivamente attribuito una valutazione ad una particolare variabile: ricomincerà quindi dal punto 1

4. quando non verrà effettuata nessuna sostituzione, ovvero quando ad ogni variabile sarà stato attribuito un valore, avvierà la valutazione della formula scritta sul primo nastro. Osserviamo che essa non sarà più strettamente una formula ben formata, dato che avrà delle sottostringhe della forma F^i o V^i; tuttavia, sarà chiaramente assimilabile ad una formula booleana ben formata, ed in particolare si presterà perfettamente ad essere valutata secondo l'algoritmo descritto in 4.2, operante in tempo quadratico

Si osserva facilmente che nella fase finale di esecuzione di questa macchina, la formula booleana iniziale viene valutata, nei vari rami della computazione (che saranno 2^m per la valutazione di m variabili), con tutte le combinazioni di valori attribuibili alle variabili. Se almeno in uno di questi casi la valutazione restituirà una stringa di V, la macchina accetterà. Ogni ciclo di questa macchina prenderà sostanzialmente il tempo di una sostituzione (tempo lineare), i cicli saranno meno di n e la valutazione finale avrà costo quadratico, per un costo totale ancora quadratico. Quindi SAT $\in$ NP.

Resta da dimostrare che $NP \leq SAT$. Questa parte della dimostrazione non viene riportata, perché è piuttosto lunga, nonché un caso particolare di quella che vedremo in 5.2.2. □

Nella teoria della complessità, il problema SAT ha acquisito un ruolo fondamentale non solo perché è stato il primo ad essere dimostrato NP-completo, ma anche perché molti altri importanti problemi in NP sono stati dimostrati completi tramite la riducibilità di SAT (o della sua versione semplificata $SAT-3$) ad essi [6].

4.4 QSAT

Si indica con la sigla QSAT (per *Quantified Boolean Satisfiability* [8]), o talvolta $TQBF$ (per *True Quantified Boolean Formulae*; ma nei primi articoli che trattano l'argomento si trova invece IEQ [12]), il problema SAT modificato quantificando, esistenzialmente ed universalmente, sulle variabili booleane in gioco (in modo che non rimanga alcuna variabile libera). Un esempio di formula booleana quantificata è:

$$\forall A(\neg A \rightarrow \exists B(A \vee B)) \; . \qquad (*)$$

Per formalizzare questo problema, ci atterremo sempre alla notazione utilizzata per il problema SAT, giungendo quindi alla seguente:

Definizione 23. *Dato l'insieme di simboli:*

$$\Sigma = \{\forall, \exists, \vee, \wedge, \leftarrow, \neg, (,), 0, 1\} ,$$

chiamiamo QSAT il sottoinsieme di Σ^* *contenente tutte e sole le formule ben formate e vere.*

È interessante innanzitutto individuare due casi particolari del problema:

- se la formula è in forma prenessa e tutti i quantificatori sono universali, stabilire se essa è in QSAT equivale a determinare se il suo corpo è una tautologia,
- se la formula è in forma prenessa e tutti i quantificatori sono esistenziali, stabilire se essa è in QSAT significa invece equivale a determinare se il suo corpo è in SAT.

Da questa ultima osservazione, ricaviamo subito un risultato interessante: siccome è semplice verificare se una formula ha solo quantificatori esistenziali prenessi, il problema QSAT sarà *più difficile* del problema SAT.

4.4.1 Una forma normale per QSAT

Così come abbiamo fatto con le macchina di Turing , anche nel problema QSAT cercheremo di stabilire delle restrizioni sull'insieme delle formule da considerare, tali però che i nostri ragionamenti a riguardo non perdano di generalità. Utilizzeremo la forma normale disgiunta, aggiungendo però la condizione che i quantificatori siano prenessi alla formula ed alternati (tra universali ed esistenziali), ed in particolare siano in numero pari ed il primo sia esistenziale.

Per portare i quantificatori in forma prenessa, esistono delle semplici regole (riportate in appendice A.2), mentre l'ipotesi sulla loro alternanza è soddisfacibile semplicemente aggiungendo quantificatori su nuove variabili che non appaiono nella formula; è quindi possibile utilizzare le regole descritte in 4.1.1. Ad esempio, per portare la formula $(*)$ in forma normale si possono compiere i seguenti passaggi:

$$\forall A \ \ (\neg A \rightarrow \exists B \ \ (A \vee B)) \qquad (*)$$

$$\forall A \ \exists B \ \ (\neg(\neg A) \vee (A \vee B))$$

$$\forall A \ \exists B \ \ (A \vee A \vee B)$$

$$\forall A \ \exists B \ \ (A \vee B)$$

$$\exists C \ \forall A \ \exists B \ \forall D \ \ (A \vee B)$$

Questa forma normale risulterà utile perché, una volta stabilito un ordine sulle variabili ed assumendo che il primo quantificatore sia esistenziale, è sufficiente, per descrivere una formula, riportarne il corpo (quantificatori esclusi, quindi). Tuttavia, a tale scopo è necessario tenere traccia delle variabili aggiunte; questo può essere fatto facilmente inserendole nella formula senza cambiarne

il valore, aggiungendo un'implicazione sempre falsa; ad esempio quella sopra può diventare:

$$\exists C \forall A \exists B \forall D [A \vee B \vee (C \wedge \neg C \wedge D)]$$

Fissate queste convenzioni, e stabilito a priori un ordine sui nomi di variabile, sapremo che la formula quantificata associata a

$$[A \vee B \vee (C \wedge \neg C \wedge D)] \ ,$$

e proprio quella in cui A e C sono quantificate esistenzialmente, mentre B e D sono quantificate universalmente.

In 4.1.1 abbiamo menzionato il fatto che la forma normale disgiunta di una formula può essere *esponenzialmente* più lunga della formula stessa: questo è un effetto che vogliamo evitare, perciò sfrutteremo la peculiarità delle formule booleane quantificate per dimostrare il

Teorema 12. *Una formula booleana quantificata può essere portata in forma normale aumentando al più quadraticamente la sua lunghezza.*

Dimostrazione. Effettivamente, le regole descritte nel caso di formule senza quantificatori erano corrette, nel senso che l'"esplosione" esponenziale non è evitabile. Per dimostrare questo teorema dovremo sfruttare la possibilità di quantificare sulle variabili.
Ricordiamo che il problema principale si poneva nel momento in cui si portava in forma disgiunta una formula della forma $\varphi \wedge \psi$; infatti la sua conversione comportava un aumento quadratico della lunghezza della formula, e la ripetizione di questo procedimento portava appunto al suo aumento esponenziale. Diamo quindi una nuova regola per affrontare questo caso: supponendo sempre per induzione che φ e ψ siano già in forma normale disgiunta, sia X una qualsiasi variabile che non compare in nessuna delle due; allora possiamo scrivere la seguente formula equivalente:

$$\forall X \ (\varphi \wedge X) \vee (\psi \wedge \neg X) \ .$$

È facile osservare che questa formula è davvero equivalente a $\varphi \wedge \psi$: meno immediato può risultare il fatto che portare tale formula in forma disgiunta è molto più semplice: X e $\neg X$ non sono due formule qualunque, ma due letterali, che possono essere fatti "scivolare" rispettivamente all'interno di ogni clausola di φ e ψ, ottenendo una formula

$$\forall X \ \varphi^X \vee \psi^{\neg X}$$

che in più rispetto alla formula iniziale avrà soltanto la quantificazione ed un letterale per ogni clausola: è evidente che ciò farà al più raddoppiare la sua lunghezza.
I passaggi successivi (portare i quantificatori in forma prenessa ed alternarli) saranno effettuati come riportato sopra, e la formula finale sarà, nel caso pessimo, in cui la formula era composta di n clausole molto brevi, lunga comunque meno di n^2.

□

Capitolo 5

$PSPACE = APTIME$

In questo capitolo, ci proponiamo di dimostrare l'equivalenza tra le due classi di complessità riconducendole entrambe ad uno stesso problema: $QSAT$.

Nella notazione delle macchine di Turing, utilizzeremo talvolta $-1, 0, 1$ al posto dei simboli $\leftarrow, \downarrow, \rightarrow$, in modo da potere usarli in operazioni senza dovere esplicitare una loro conversione.

5.1 $QSAT \in PSPACE - C$

5.1.1 $QSAT \in PSPACE$

Sappiamo (da 4.2) che la valutazione di una formula booleana chiusa può essere effettuata da una macchina di Turing deterministica $\mathcal{N}$ operante in tempo quadratico; possiamo supporre (modulo l'aggiunta di un orologio, come visto nella sezione 3.5.2), che su tutti gli input che non rappresentano una formula booleana chiusa vera essa si sposti in uno stato pozzo, e se si sposta in tale stato pozzo diciamo che essa *rifiuta*. Cercheremo ora di realizzare una macchina deterministica M che valuti una formula booleana quantificata utilizzando spazio polinomiale.

Il principio della macchina sarà il seguente: essa riceverà in input $\varphi(x_1, \dots x_m)$, una formula booleana, e dovrà calcolare il valore di:

$$\exists x_1 \forall x_2 \exists x_3 \dots \forall x_m \ \varphi(x_1 \dots x_m) \ .$$

Per fare ciò, essa scriverà su di un nastro, una dopo l'altra, tutte le possibili valutazioni delle variabili della formula, sotto forma di stringhe binarie di lunghezza m: a tal fine, essa scriverà inizialmente la stringa $[0]^m$ e quindi incrementerà sempre di 1 il numero rappresentato in forma binaria.
Questo modo di procedere non ci garantisce solo che effettivamente tutte le possibili stringhe saranno prese in considerazione, ma anche che esse saranno scritte in un ordine particolare: se supponiamo che ognuna di queste stringhe individui un ramo di un albero binario completo avente m livelli, nella nostra scansione i rami di un qualsiasi sottoalbero, ovvero discendenti da un qualsiasi nodo, saranno considerati consecutivamente.[1] Tale albero indicherà per noi le

[1] Compiendo una cosiddetta *visita in profondità* dell'albero

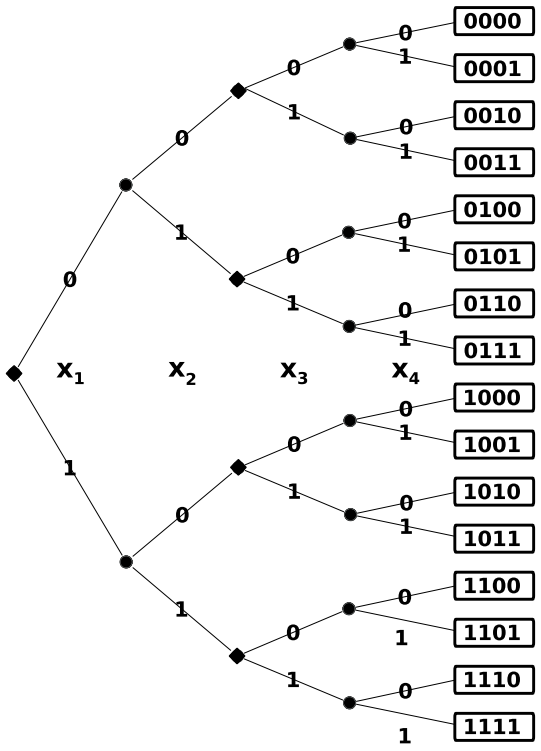


Figura 5.1: L'albero delle possibili stringhe binarie ($m = 4$, i quadrati corrispondono ai nodi esistenziali).

possibili valutazioni di φ.

Fissata una tale stringa s, avremo quindi stabilito il valore di $\varphi(s)$; potremo quindi copiare su un nuovo nastro φ con le costanti $s_1 \dots s_m$ sostituite alle variabili $x_1 \dots x_m$, e quindi valutarla (in tempo quadratico) e memorizzare il suo valore di verità nella n-esima cella di un quarto nastro. Supponiamo che $\varphi(s)$ sia falsa; allora, lo è anche:

$$\forall x_m \; \varphi(s_1, s_2 \dots s_{m-1}, x_m) \; ,$$

se invece $\varphi(s)$ è vera, sarà necessario considerare la valutazione $s_1, \dots s_{m-1}, \neg s_m$ per potere concludere; potremo quindi annotare anche questo risultato, nella $n-1$-esima cella del quarto nastro. In seguito, cercheremo di stabilire se sia vera la formula

$$\exists x_{m-1} \forall x_m \; \varphi(s_1, s_2 \dots s_{m-2}, x_{m-1}, x_m) \; ,$$

ma per fare ciò ci servirà conoscere il valore non solo di

$$\forall x_m \; \varphi(s_1, s_2 \dots s_{m-1}, x_m) \; ,$$

ma anche di

$$\forall x_m \; \varphi(s_1, s_2 \dots \neg s_{m-1}, x_m) \; ,$$

ovvero non dovremo più analizzare solo 2, ma 4 diverse stringhe, che però saranno sempre consecutive. Il procedimento per analizzare tutte le 2^m stringhe è piuttosto semplice; meno banale è il problema di assegnare, mano a mano, valori di verità ad ogni nodo dell'albero; infatti il numero dei nodi è $2^{m+1} - 1$, e non possiamo permetterci di assegnare una cella di memoria ad ognuno. Posto che, come accennato in questo esempio, ad ogni nodo dell'albero è associata una ed una sola formula avente alcune variabili quantificate ed altre con valore fissato (chiameremo "veri" i nodi associati a formule vere, e viceversa), utilizzeremo quindi la seguente strategia:

- un nodo ad un livello dispari (associato ad un quantificatore universale) sarà considerato vero *fino a prova contraria*, ovvero fino a che un suo nodo figlio non risulti falso;
- un nodo ad un livello pari (associato ad un quantificatore esistenziale) sarà considerato *falso* fino a prova contraria, ovvero fino a che un suo figlio non risulti vero;
- siccome per stabilire il valore di verità di un nodo è sufficiente conoscere il valore di quelli sottostanti, potremo procedere un sottoalbero alla volta; consideriamo ad esempio un nodo α a livello j pari: una volta che avremo studiato il sottoalbero discendente da α, utilizzando $j-1$ celle di memoria, memorizzeremo il valore di verità di α: vero, se ha almeno un figlio vero, falso altrimenti.
  Se risulta falso, ciò rende falso anche il suo nodo padre, che è a livello dispari; altrimenti, le $j-1$ celle di memoria potranno ora essere riutilizzate per calcolare il valore di verità di β, fratello di α, ed una volta che i valori di verità di α e β saranno noti, potremo calcolare con certezza il valore del loro nodo padre, memorizzandolo e liberando questa volta j celle di memoria, dato che il valore di α e di β non ci interesserà più

Tale ragionamento si può formalizzare per induzione su $m - j$.

L'idea di verità o falsità "fino a prova contraria" verrà implementata inizializzando il quarto nastro con una stringa lunga m e data dall'alternanza di due simboli, indicanti rispettivamente i nodi universali e falsi ed i nodi esistenziali veri. I due simboli potranno essere cambiati, nel corso dell'esecuzione, in altri due simboli indicanti rispettivamente i nodi universali veri ed i nodi esistenziali falsi. Ogni volta che si esaurirà lo studio di un sottoalbero alto j, sarà importante riportare gli ultimi j simboli della stringa nella situazione iniziale.

M avrà quindi 5 nastri, di alfabeto rispettivamente:

$$\{\exists, \forall, \neg, (,), \vee, \wedge, V, F, 0, 1\}$$

$$\{\exists, \forall, \neg, (,), \vee, \wedge, V, F, 0, 1\}$$

$$\{0, 1\}$$

$$\{\exists, \forall\} \times \{V, F\}$$

$$\{0, 1\}$$

Il primo nastro sarà utilizzato esclusivamente per leggere l'input (in cui le variabili, numerate da 1 ad m, saranno identificate con il loro numero scritto, tra

parentesi, in forma binaria invertita - si veda la dimostrazione del teorema 11). Questa è una summa delle regole di M:

1. inizialmente, essa scrive sul terzo nastro (che dovrà contenere sempre le varie stringhe binarie menzionate sopra sotto il nome di s) la stringa $[0]^n$,

2. scrive quindi sul quarto nastro (utilizzato per memorizzare le condizioni dei nodi) la stringa $[(\forall, V)]^n$ e sostituisce quindi un simbolo ogni due (partendo dal primo) con $(\exists, F)$, lasciando poi il cursore alla fine e riportando invece il terzo cursore all'inizio;

3. copia nel secondo nastro φ, ovvero il contenuto del primo,

4. scrive nel quinto nastro (che indicherà ogni volta la variabile che stiamo valutando in φ) [0], ovvero la forma binaria invertita di 0

   (a) nel secondo nastro, sostituisce ogni istanza di $[(] + \eta + [)]$, dove η è il contenuto del quindi nastro, con il simbolo F se il terzo cursore sta leggendo 0, V se sta leggendo un 1; in altri termini, valuta la variabile indicata da η; se però non viene effettuata alcuna sostituzione (ovvero se η non appare nella formula - ricordiamo che le variabili sono solo m, non n), passa al punto 5;

   (b) incrementa il contenuto del quinto nastro di 1[2] e sposta il terzo cursore a destra, preparandosi quindi a considerare la variabile successiva;

   (c) ricomincia da (a);

5. esegue $\mathcal{N}$ sul secondo nastro, ovvero valuta la formula chiusa; entra quindi in uno stato q_V se essa accetta, q_F se rifiuta. Questi due stati fungono da "messaggeri" tra i nodi figli ed i nodi padri, ed hanno una funzione simile a quelle che può avere, in un computer capace di sommare numeri interi binari, un registro adibito a memorizzare il riporto:

   (a) nello stato q_F: se il quarto cursore legge $(\forall, V)$, scrive $(\forall, F)$, altrimenti riscrive il simbolo letto,

   (b) nello stato q_V: se il quarto cursore legge $(\exists, F)$, scrive $(\exists, V)$, altrimenti riscrive il simbolo letto,

   (c) in ognuno dei due casi: se il simbolo scritto indica un nodo vero, ovvero è della forma $(*, V)$ entra nello stato q_V, altrimenti q_F; sposta poi il quarto cursore a sinistra, ovvero inizia a considerare il nodo padre;

   (d) se nello stato q_F la macchina legge (sempre nel quarto nastro) $\triangleright$, passa a 6 (non abbiamo concluso niente studiando la metà sinistra dell'albero, è necessario ora studiare la metà destra),

   (e) se nello stato q_V la macchina legge (sempre nel quarto nastro) $\triangleright$, entra in q_f (ovvero accetta, in quanto ha verificato che il nodo radice, associato alla la formula con tutte le variabili quantificate è vero).

[2] Anche l'"incrementare" va inteso nella notazione binaria invertita

6. incrementa il contenuto del terzo nastro (visto come numero binario) di 1, ovvero passa alla stringa s successiva, e reinizializza (vi riscrive i simboli alternati scritti all'inizio) gli ultimi i simboli della stringa contenuta nel quarto nastro, dove i è il numero di simboli cambiati nel terzo (ovvero il numero di riporti causati dall'incremento di 1).

I cicli sono due: uno interno che cicla sulle diverse variabili per sostituirle nella formula (e si ripete quindi m volte) ed uno esterno che cicla sulle diverse configurazioni di variabili (e si ripete quindi 2^m volte): M opera in tempo esponenziale. Tuttavia, su ogni stringa essa scrive al più n simboli, per cui essa opera in spazio non solo polinomiale ma addirittura lineare; ciò dimostra che $QSAT \in PSPACE$.

5.1.2 $PSPACE \leq QSAT$

Sia M una macchina di Turing deterministica mononastro operante in spazio al più n^k su input di lunghezza n: cerchiamo una formula booleana quantificata di lunghezza polinomiale in n ed avente m variabili libere che sia vera (valutata in un input w di lunghezza n opportunamente codificato in valori booleani) se e solo se M accetta w. La formula utilizzata varierà al variare di M ed w, ma il risultato globale sarà che $QSAT$, la collezione di tutte queste formule, è un problema $PSPACE$-hard.

Possiamo assumere, senza perdita di generalità, che M sia una macchina binaria; siano inoltre

$$k_q = \lceil \log_2 |Q| \rceil$$

e

$$k_n = k * \lceil \log_2 n \rceil \ .$$

Sia G il grafo delle configurazioni di M e, data una stringa di input w, sia G_w il sottografo composto dalle sole configurazioni (s, q, a) tali che $|s| < |w|^k$ (ovvero delle configurazioni che ci interessano, data la limitazione sullo spazio utilizzato da M).

Supponendo di scrivere q e a sempre in rappresentazione binaria di rispettivamente k_q e k_n cifre (ovvero aggiungendo zeri qualora i numeri siano piccoli), possiamo vedere ogni configurazione come una stringa binaria di lunghezza

$$N \stackrel{def}{=} n^k + k_q + k_n = O(n^k),$$

ovvero come una sequenza di N variabili booleane.

I vertici di G_w, ovvero le possibili sequenze di N valori booleane, sono quindi $2^N \sim 2^{(n^k)}$. È evidente che M accetta w se e solo se in G_w esiste un percorso da w a f. D'altronde, se questo percorso esiste è ovviamente lungo al più 2^N. Quindi possiamo ricondurci al seguente problema:

in G_w, esiste un percorso lungo al più 2^N tra w ed f?

Per comodità di notazione, indicheremo questo problema come[3]

$$REACH_{2^N}(w, f) \ .$$

Data la codifica molto particolare di G_w, che deriva dalla descrizione di M, possiamo trasformare questo problema in una formula booleana; per far questo però saranno necessari alcuni passaggi.

Al variare di due configurazioni $A = a_0, \dots a_N$ e $B = b_0, \dots b_N$, creiamo:

1. una formula $\varphi_{eq_N}(x_0, \dots x_N, y_0, \dots y_N)$ tale che $\varphi(A, B)$ valga se e solo se A e B sono la stessa configurazione.

   Per fare ciò, iniziamo con l'osservare che l'equivalenza tra due variabili booleane x, y si può esprimere con la semplice formula:

   $$\varphi_{eq}(x, y) \stackrel{def}{=} (x \wedge y) \vee (\neg x \wedge \neg y) \ .$$

   Ciò detto, è possibile costruire una formula che esprima l'identicità di due configurazioni (per noi una configurazione sarà sempre un vettore di N variabili booleane):

   $$\begin{aligned} \varphi_{EQ_N}(X, Y) \equiv \varphi_{EQ}(x_1, \dots x_N, y_1 \dots y_N) &\stackrel{def}{=} \bigwedge_{i<N} \varphi_{eq}(x_i, y_i) \\ &\equiv \bigwedge_{i<N} (x_i \wedge y_i) \vee (\neg x_i \wedge \neg y_i) \end{aligned}$$

2. per ogni coppia $(\sigma, q) \in \Sigma \times Q$ $(= \{0,1\} \times \{0,1\}^{k_q})$, una formula

   $$\varphi_{\sigma,q}(a, \sigma', q', a') \ ,$$

   o più precisamente:

   $$\varphi_{\sigma,q}(x_{a1}, \dots x_{ak_n}, y_\sigma, y_{q1}, \dots y_{qk_q}, y_{a1}, \dots y_{ak_n}) \ ,$$

   che valga se e solo se:

   $$\delta(\sigma, q) = (\sigma', q', a' - a) \ ,$$

   ovvero che verifichi la corretta applicazione di una regola di δ.

   Per la sua costruzione, utilizziamo alcune semplici sottoformule:

   (a) per ogni $\sigma \in \Sigma = \{0,1\}$, φ_{eq_σ} tale che:

   $$\varphi_{eq_\sigma}(\sigma') \equiv \varphi_{eq}(\sigma', \sigma) \ ,$$

   che si definisce semplicemente come:

   $$\varphi_\sigma(\varphi') \stackrel{def}{=} \begin{cases} \sigma' & \text{se } \sigma = V \\ \neg\sigma' & \text{se } \sigma = F \end{cases} \ ;$$

[3] Il problema REACH è un classico della teoria della complessità, e consiste appunto nella ricerca di un percorso tra due nodi di un grafo; tuttavia, nella sua formulazione solita l'input rappresenta la codifica di un generico grafo: la nostra situazione, in cui l'input codifica una macchina di Turing , è piuttosto diversa, ma l'algoritmo che utilizzeremo è sostanzialmente l'adattamento di un algoritmo per il REACH originale.

(b) per ogni $q \in Q$,

$$\varphi_{EQ_q}(q') \equiv \varphi_{EQ_q}(x_1 \dots x_{k_q})$$

tale che:

$$\varphi_{EQ_q}(q') \equiv \varphi_{EQ_{k_q}}(q, q') \ ,$$

definita, in linea con il caso precedente, come:

$$\varphi_{EQ_q}(q') \stackrel{def}{=} \bigwedge_{i \leq k_q} \neg^{q_i} x_i \ ,$$

dove con $\neg^{q_i} x$ indichiamo x se $q_i = F$, $\neg x$ altrimenti;

(c) per ogni $d \in \{-1, 0, 1\}$,

$$\varphi_d(a, a') \equiv \varphi_d(x_1 \dots x_{k_n}, y_1 \dots y_{k_n}) \ ,$$

che valga se e solo se $a+d = a'$, la cui costruzione è piuttosto semplice.

È facile verificare che allora, data una regola $\delta(\sigma, q) = (\sigma_\delta, q_\delta, d_\delta)$,

$$\varphi_{\sigma,q}(a, \sigma', q', a') \stackrel{def}{=} \varphi_{\sigma_\delta}(\sigma') \wedge \varphi_{q_\delta}(q') \wedge \varphi_{d_\delta}(a, a')$$

è esattamente la formula che cerchiamo, ovvero vale se e solo se

$$\delta(\sigma, q) = (\sigma', q', a' - a) \ .$$

3. una formula $\varphi_{passo}(x_0 \dots x_N, y_0 \dots y_N)$ tale che $\varphi(A, B)$ valga se e solo se la macchina M si sposta (in esattamente un passo) dalla configurazione A nella configurazione B. Definiamo innanzitutto, ad imitazione delle precedenti, una sequenza di sottoformule $\varphi_{bin_i}(a_1 \dots a_{k_n})$, ognuna delle quali valga se e solo se $a_1, a_2 \dots a_{k_n}$ sono i coefficienti della raffigurazione binaria di i, ovvero se $\sum_{j=1}^{k_n} a_j 2^j = i$.
Possiamo quindi definire:

$$\varphi_{passo_i}(X, Y) \equiv \varphi_{passo_i}(s, q, a, s', q', a') \stackrel{def}{=}$$

$$\bigvee_{(\sigma,\sigma') \in \{0,1\}^2} \bigvee_{\bar{q} \in Q} \varphi_{bin_i}(a) \wedge \varphi_\sigma(s_i) \wedge \varphi_{\sigma'}(s'_i) \wedge \varphi_{\bar{q}}(q) \wedge \varphi_{\sigma,\bar{q}}(a, \sigma', q', a') \wedge$$

$$\varphi_{EQ_{n^k-1}}(s_1 \dots s_{i-1}, s_{i+1} \dots s_{n_k}, s'_1 \dots s'_{i-1}, s'_{i+1} \dots s'_{n_k}) \ ,$$

il cui significato è "il cursore è in posizione i, il cambiamento di stato e del simbolo scritto nella cella i seguono la regola (data da δ), tutte le altre celle restano inalterate" (osserviamo che soltanto uno dei termini di questa lunga disgiunzione potrà essere vero: quello corrispondente a σ, σ' e $\bar{q}$ dati da s, s' e q). Finalmente, possiamo quindi definire:

$$\varphi_{passo}(X, Y) \stackrel{def}{=} \bigvee_{1 \leq i \leq n^k} \varphi_{passo_i}(X, Y) \ ,$$

interpretabile come

"per un certo i, il cursore si trova in posizione i ed M si sposta in un passo da X ad Y" ,

o più semplicemente *"M si sposta in un passo da X ad Y"*.

Si verifica facilmente che φ_{passo} ha lunghezza polinomiale in N e quindi in n.

Il nostro scopo è ora definire, per ogni $i \in \mathbb{N}$, una formula $\psi_i(X, Y)$ che formalizzi l'affermazione "la macchina M si sposta da X a Y in al più 2^i passi". Lo faremo per induzione e, sfruttando gli strumenti già costruiti, il caso base è banale:

$$\psi_0(X, Y) \stackrel{def}{=} \varphi_{EQ_N}(X, Y) \vee \varphi_{passo}(X, Y) .$$

Per il passo induttivo, osserviamo che:

$$\psi_i(X, Y) \leftrightarrow \exists Z(\psi_{i-1}(X, Z) \wedge \psi_{i-1}(Z, Y)) ,$$

o più precisamente:

$$\psi_i(X, Y) \leftrightarrow \exists z_0 \ldots \exists z_N(\psi_{i-1}(X, z_0 \ldots z_N) \wedge \psi_{i-1}(z_0 \ldots z_N, Y)) .$$

Non possiamo però definire ψ_i semplicemente come il membro destro della doppia implicazione; infatti si vede facilmente che in tal modo φ_{i+1} avrebbe lunghezza almeno doppia di φ_i, e quindi per induzione φ_N (che è la formula che esprime $REACH_{2^N}$) sarebbe lunga almeno

$$n^k * 2^N \sim n^k * 2^{(n^k)} .$$

Possiamo però osservare (imitando la dimostrazione del teorema 12) che:

$$\begin{array}{c} \exists Z(\psi_{i-1}(X, Z) \wedge \psi_{i-1}(Z, Y)) \\ \updownarrow \\ \exists Z \forall T \forall U \left[((T = X \wedge U = Z) \vee (T = Z \wedge U = Y)) \rightarrow \psi_i(T, U)\right] \end{array}$$

o più formalmente:

$$\exists Z \forall T \forall U \left[((\varphi(T, X) \wedge \varphi(U, Z)) \vee (\varphi(T, Z) \wedge \varphi(U, Y))) \rightarrow \psi_i(T, U)\right]$$

dove con φ indichiamo per brevità φ_{EQ_N}.

Tale formula ha finalmente lunghezza $O(N^{2k})$ ed è vera se e solo se M accetta w. È facile verificare che la costruzione di tale formula chiede tempo polinomiale, e che quindi quella descritta è effettivamente una riduzione polinomiale di $PSPACE$ a $QSAT$.

La dimostrazione fornita si applica ad una macchina avente un solo nastro; essa si generalizza ad una macchina multinastro in virtù del teorema B.2.

5.2 $QSAT \in APTIME - C$

5.2.1 $QSAT \in APTIME$

Costruiremo una macchina di Turing alternante Mche risolva il problema $QSAT$, ovvero si comporti come quella costruita per dimostrare che $QSAT \in PSPACE$; ovviamente però sfrutteremo l'alternanza per fare in modo che essa, a differenza

di quella vista allora, lavori in tempo polinomiale.

Le convenzioni assunte saranno le stesse: M riceverà la codifica di una formula φ avente m variabili libere $x_1.x_2 \dots x_m$, e dovrà accettare se e solo se la formula

$$\exists x_1 \forall x_2 \exists x_3 \dots \forall x_m \, \varphi(x_1, x_2, x_3 \dots x_m)$$

è vera.

Se nel paragrafo 5.1.1 la situazione era resa complessa dalla necessità di concatenare temporalmente il controllo di tutte possibili configurazioni di valori e di memorizzare in modo furbo i risultati intermedi, con il paradigma alternante invece semplificheremo nettamente la situazione, dando un assaggio del fatto che esso si presta naturalmente ad affrontare problemi nella cui formulazione si trovano, impliciti o espliciti, molti quantificatori sia esistenziali che universali.

Adesso, infatti, le 2^m possibili assegnazioni di valori per le variabili verranno generate tutte contemporaneamente, in modo nondeterministico, su di un nastro: i valori delle variabili quantificate universalmente verranno scritti in stati universali, e viceversa, in modo tale che lo stato accettante o meno delle foglie dell'albero delle computazioni, ovvero la verità o falsità della formula valutata nelle varie possibili assegnazioni di valori, ricada automaticamente, con un effetto a cascata, sull'accettazione delle configurazioni precedenti.

La parte finale dell'esecuzione della macchina, che consiste nel sostituire ogni istanza di una variabile con il valore attribuitole e valutare la formula chiusa risultante, sarà identica a quella vista in 5.1.1.

M avrà 3 nastri, di alfabeto rispettivamente:

$$\{\exists, \forall, \neg, (,), \vee, \wedge, 0, 1\}$$

$$\{0, 1\}$$

$$\{V, F\} \, ;$$

infatti, siccome in questo caso non abbiamo necessità di effettuare molte copie successive della formula iniziale (le sue diverse valutazioni si svolgono in parallelo), potremo operare direttamente nel nastro di input. M avrà insieme degli stati

$$\{\forall, \exists\} \cup (\{\mathcal{N}\} \times Q_{\mathcal{N}}) \, ,$$

dove $\mathcal{N}$ è la macchina deterministica che valuta in tempo quadratico un'espressione booleana chiusa, e gli stati $\forall$ ed $\exists$ sono (prevedibilmente) rispettivamente uno stato universale ed uno esistenziale; stabiliamo che M riceva nel primo nastro l'input, consistente nella sola descrizione di φ in termini di simboli logici e di variabili scritte nella forma

$$[(] + bin(i)^{-1} + [)]$$

(si veda 5.1.1), e operi nel seguente modo (avendo come stato iniziale $\forall$):

1. scriva sul secondo nastro 0, ovvero $bin(0)^{-1}$,

2. scriva sul terzo, non deterministicamente, i simboli V (in un ramo della computazione) e F (in un altro),

3. posto s il contenuto del secondo nastro (s sarà sempre la forma binaria invertita di un numero tra 1 ed m), sostituisca ogni istanza di $[(] + s + [)]$ che trova nel primo nastro con il simbolo contenuto in M_3; se nessuna sostituzione viene effettuata (ovvero se la trasformazione della formula in una formula chiusa è giunta al termine), cancelli dal nastro 2 ogni simbolo (un modo per segnalare che la sostituzione è giunta al termine), altrimenti incrementi di 1 il contenuto del secondo nastro,

4. se il secondo nastro contiene almeno un simbolo, ovvero se c'è ancora qualche variabile libera da valutare, ricominci dal punto 2,

5. se invece il secondo nastro è vuoto, avvii $\mathcal{N}$ (ovviamente modificata in modo da operare con gli stati $\{\mathcal{N}\} \times Q_{\mathcal{N}}$ invece che $Q_{\mathcal{N}}$) sul primo nastro. Osserviamo che, essendo $\mathcal{N}$ deterministica, è indifferente stabilire se i suoi stati sono universali o esistenziali; l'unica eccezione è data da $(\mathcal{N}, q_f)$ (esso non ha rami uscenti, e deve essere considerato universale - ovvero accettante).

Si verifica facilmente che tutte queste operazioni sono sufficientemente semplici, e perciò M opererà in tempo polinomiale: ciò completa la dimostrazione.

5.2.2 $APTIME \leq QSAT$

La dimostrazione di questo ultimo risultato prenderà le mosse da quella di $PSPACE \leq QSAT$, ed in particolare si baserà sempre sull'individuazione di una formula booleana quantificata, associata ad una macchina mononastro M ed un input w, che sia vera se e solo se M accetta w.

Siano quindi M una generica macchina in $ATIME(n^k)$ (con alfabeto Σ ed insieme degli stati Q), G il suo grafo delle configurazioni e G_w il sottografo composto dalle configurazioni (s, q, a) tali che $|s| \leq n^k$.

Come dimostrato in 3.2.1, possiamo supporre che M sia binaria, alterni configurazioni universali ed esistenziali e che tutte quelle terminali siano accettanti.

Come al punto 5.1.2, consideriamo ogni configurazione come una sequenza di N variabili booleane. Avevamo definito, per ogni coppia (q, σ), una formula:

$$\varphi_{\sigma,q}(x_{a1}, \dots x_{ak_n}, y_\sigma, y_{q1}, \dots y_{qk_q}, y_{a1}, \dots y_{ak_n}) \ ,$$

che esprimeva la seguente affermazione:

$$\delta(\sigma, q) = (\sigma', q', a' - a) \ .$$

Adesso che però la nostra δ non assume più valori in

$$\Sigma \times Q \times \{\leftarrow, \rightarrow, \downarrow\} \ ,$$

ma in

$$\mathcal{P}(\Sigma \times Q \times \{\leftarrow, \rightarrow, \downarrow\}) \ ,$$

e più precisamente in

$$(\Sigma \times Q \times \{\leftarrow, \rightarrow, \downarrow\})^2 ,$$

dovremo darne una forma modificata. Stabiliamo innanzitutto un ordine su $\Sigma, Q, \{\leftarrow, \rightarrow, \downarrow\}$ e di conseguenza[4] su $(\Sigma \times Q \times \{\leftarrow, \rightarrow, \downarrow\})^2$, e per ogni coppia (q, σ) definiamo:

$$\delta^0((q,\sigma)) \stackrel{def}{=} \min(\delta((q,\sigma)))$$

e

$$\delta^1((q,\sigma)) \stackrel{def}{=} \max(\delta((q,\sigma))) .$$

Possiamo considerare δ^0 e δ^1 due funzioni di transizione di due macchine deterministiche, e quindi associare loro due formule $\varphi^0_{\sigma,q}$ e $\varphi^1_{\sigma,q}$ così come a δ avevamo associato $\varphi_{\sigma,q}$ nella dimostrazione di $PSPACE \leq QSAT$. Ridefiniamo quindi:

$$\varphi_{\sigma,q}(a,\sigma',q',a') \stackrel{def}{=} \exists c\, (\neg c \wedge \varphi^0_{\sigma,q}(a,\sigma',q',a')) \vee (c \wedge \varphi^1_{\sigma,q}(a,\sigma',q',a')) ,$$

Utilizzando queste nuove $\varphi_{\sigma,q}$, ridefiniamo $\varphi_{passo_i}(X,Y)$ e $\varphi_{passo}(X,Y)$.

Ridefiniamo quindi una formula ψ_i al variare di i, ma con un significato diverso da quello dato in precedenza: $\psi_i(X)$ vorrà dire "dalla configurazione X la macchina accetta dopo al più i passi" (osserviamo che, grazie al vincolo sulla lunghezza delle computazioni, la macchina accetterà se e solo se vale $\psi_{n^k}(X_w)$, dove X_w è la configurazione con stato iniziale, cursore ad inizio nastro e input w). In particolare, poiché le configurazioni accettanti in 0 passi sono tutte e sole quelle senza figli, avremo

$$\psi_0(X) \stackrel{def}{=} \forall Z(\neg\varphi_{passo}(X,Z))$$

e per $i \neq 0$ definiremo $\psi_i(X)$ come:

$$\exists Z_i(\varphi_{passo}(X,Z_i) \rightarrow \psi_{i-1}(Z_i)) \vee \psi_{i-2}(X) \text{ se } i \text{ è dispari}$$
$$\forall Z_i(\varphi_{passo}(X,Z_i) \rightarrow \psi_{i-1}(Z_i)) \vee \psi_{i-2}(X) \text{ se } i \text{ è pari} .$$

Il termine sinistro di ognuna di queste formule si può interpretare (sempre grazie alla nostra ipotesi sull'alternanza di configurazioni esistenziali ed universali) come *"X accetta in i passi"*; il termine destro serve invece a "saltare i passi", ovvero a permettere che $\psi_i(X)$ sia vera anche se X accetta in *meno* di i passi.

A fugare qualche possibile dubbio sulla correttezza di queste formule, un'osservazione: quando diciamo che M accetta in n^k passi, intendiamo dire che essa accetta in *al più* n^k passi; tuttavia, c'è una restrizione ben precisa che le condizioni imposte su M (l'alternanza di stati universali ed esistenziali ed il fatto che ogni configurazione esistenziale ha figlie) stabiliscono: il numero di passi impiegati sarà pari se e solo se lo stato iniziale è universale.

Finalmente, si vede facilmente che $\psi_{n^k}(X_w)$ può essere costruita in tempo polinomiale, e che la lunghezza è dell'ordine di n^{2k}; abbiamo quindi dimostrato che $APTIME \leq QSAT$ (di nuovo, la generalizzazione a macchine multinastro è fornita dal teorema B.2).

[4] Ad esempio con l'ordine lessicografico

Ricordiamo ora che avevamo lasciato in sospeso la dimostrazione del Teorema di Cook (11) accampando la scusa che essa era il caso particolare di quest'ultima. In effetti, supponiamo che M abbia solo stati esistenziali, ovvero sia nondeterministica: rinunciando ovviamente all'ipotesi che i quantificatori siano alternati, possiamo sopprimere la definizione di ψ_i nei casi di i pari, ed utilizzare sempre l'altra.[5] La formula risultante contiene sì dei quantificatori, ma sono tutti esistenziali, e non preceduti da negazioni; è possibile quindi portarli in forma prenessa, e a quel punto studiare il valore di verità di ψ_{n^k} equivale a studiare la soddisfacibilità del suo corpo.

5.3 Conclusione

La conclusione è abbastanza evidente: esiste un problema che è completo sia per $PSPACE$ che per $APTIME$; di conseguenza entrambi le classi possono essere ridefinite come "*l'insieme dei problemi riconducibili a* $QSAT$", e sono quindi coincidenti.

[5] In particolare, decideremo di non considerare più le configurazioni accettanti in un passo come universali senza figli ma come aventi stato q_f, per cui ψ_0 esprimerà tale fatto.

Capitolo 6

Giochi ed alternanza

∃rnesto ed ∀ntonio, due amici, hanno inventato un gioco: essi scelgono a caso una formula booleana avente variabili libere $x_1, x_2 \dots x_n$, poi ∃rnesto comincia la partita stabilendo il valore (V o F) di x_1; ∀ntonio sceglie quindi quello di x_2, ∃rnesto quello di x_3 e così via, fino a che ogni variabile non è determinata. A quel punto, essi calcolano il valore di verità complessivo che la formula acquista con tale valutazione: se essa risulta vera vince ∃rnesto, altrimenti vince ∀ntonio.

∃rnesto vorrebbe cercare di capire *quali sono* le formule φ per cui ha una strategia vincente, ovvero una sequenza di scelte, dipendente dalla formula ma anche dalle scelte di ∀ntonio, che gli permetta di vincere con certezza, facendo risultare vera φ. Egli ragiona come segue:

"Data una certa φ, potrò scegliere un valore di x_1 per cui, qualunque valore di x_2 ∀ntonio scelga, io possa scegliere un valore di x_3 per cui, qualunque valore di x_4 egli scelga ... φ risulti vera?"

Tutt'a un tratto, ∃rnesto si rende conto che la sua domanda equivale a chiedersi se la formula quantificata

$$\exists x_1 \forall x_2 \exists x_3 \dots \varphi(x_1, x_2 \dots x_n)$$

appartiene o meno a $QSAT$ (e che tra l'altro tale formula è nella forma normale per le formule booleane quantificate descritta al punto 4.4.1).

6.1 Giochi in PSPACE

In effetti, trovare strategie vincenti per il gioco di ∃rnesto ed ∀ntonio è un problema $PSPACE$-completo, e capire se una data posizione è una posizione vincente per un giocatore (ovvero una posizione in cui tale giocatore ha una strategia vincente) ne è la versione decisionale; sarà interessante osservare che tra i vari problemi in $PSPACE$, che quindi si riconducono ad esso, ci sono le ricerche di posizioni vincenti per numerosi altri giochi a due giocatori. Riportiamo innanzitutto una definizione:

Definizione 24. *Si dice* ad informazione perfetta[1] *un gioco tale che, al momento di effettuare una mossa, un giocatore possa sempre conoscere esattamente la situazione attuale di ogni componente del gioco, nonché quella subito successiva alla sua mossa.*

Alcuni esempi di giochi ad informazione perfetta sono il tris, gli scacchi, la dama e il go, così come alcuni classici giochi matematici (ad esempio il Chomp e il Nim), ma anche alcuni giochi a più di 2 giocatori.
Alcuni esempi di giochi *non* ad informazione perfetta sono il poker e la briscola (perché un giocatore non conosce le carte in mano ad un altro giocatore), la morra e la morra cinese (perché un giocatore non può prevedere quale simbolo o numero "butterà" il suo avversario) e i giochi di dadi (perché un giocatore non può prevedere che numero uscirà da un lancio).

Per studiare asintoticamente la complessità di un gioco, sarà necessario supporre che la *dimensione* n del gioco possa variare a piacere: il concetto di dimensione trova un significato naturale nel Chomp e nel Nim, in cui n sarà semplicemente il numero di quadretti/monete iniziale, e nel gioco di ∃rnesto ed ∀ntonio, in cui n potrà essere la lunghezza della formula. Altri giochi, come il go, possono essere *adattati* a dimensioni diverse da quelle standard: ad esempio nel go si suppone che utilizzare goban (scacchiere) di dimensioni maggiori delle 19×19 regolamentari non alteri sostanzialmente il gioco in sé, per cui possiamo considerare n il lato della scacchiera.
Per altri giochi, come il tris, gli scacchi e la dama, in cui tattiche e strategie sono strettamente condizionate dalle dimensioni, dal numero ed eventualmente dal tipo dei pezzi, una tale generalizzazione perde invece di senso; tutt'al più fornisce un limite superiore (costante) alla difficoltà del problema (limite che comunque nel caso degli scacchi è assolutamente inabbordabile per un qualsiasi computer esistente, ma non nel caso del tris, e nemmeno in quello della dama, in cui è stato dimostrato per esaustione che né il bianco né il nero ha una strategia vincente[2]).

Teorema 13. *Siano G un gioco ad informazione perfetta e k una costante tali che, in una partita di dimensione n:*

1. *il numero di mosse che una partita può durare sia limitato da n^k,*
2. *quando è il suo turno di giocare, il numero di mosse tra cui un giocatore può scegliere sia limitato da n^k, ed egli abbia un algoritmo R, con tempo di esecuzione al più n^k, per sapere se una sua mossa è lecita,*

[1] Tali giochi vengono talvolta detti *ad informazione completa*, ma altri adoperano questa qualifica per i giochi in cui un giocatore conosce la situazione attuale ma non necessariamente quella successiva alla sua mossa, ad esempio se i giocatori muovono contemporaneamente. In realtà, il ragionamento che segue si può estendere ad una categoria molto più ampia di giochi, contenente (non solo) quelli ad informazione completa (in questo senso più esteso), mediante il semplice espediente di considerare il "fato" come un ulteriore giocatore da sconfiggere; tuttavia, questo approccio rende poco interessante l'analisi, dato che sono pochi i giochi con una componente aleatoria in cui un giocatore veramente sfortunato possa comunque avere una strategia vincente.

[2] Nel senso che una partita giocata ottimamente da entrambe le parti conduce necessariamente ad una patta; l'analisi completa è stata compiuta - utilizzando le regole inglesi, simili a quelle italiane ma con l'aggiunta della possibilità per una pedina di mangiare una dama - dal gruppo Chinook [11]; iniziata nel 1989, è terminata nel luglio 2007 ed ha richiesto l'utilizzo quasi costante di 12 computer periodicamente sostituiti da modelli sempre più all'avanguardia.

3. *i turni dei giocatori siano fissati in anticipo,*

4. *al termine della partita, le regole per stabilire chi ha vinto siano calcolabili in tempo n^k;*

allora la ricerca di strategie vincenti in una partita a G di dimensione n è un problema in $PSPACE$.

Nel caso del Nim e del Chomp, si vede facilmente che $k = 1$. Nel caso del go, perché esso rientri nelle ipotesi del teorema sarà necessario forzare nuovamente le regole, stabilendo ad esempio che il gioco termina dopo al più n^2 mosse; sebbene ciò sia probabilmente vero in qualsiasi partita mai giocata, il fatto che un giocatore possa passare il suo turno quando vuole rende questa ipotesi, appunto, una forzatura, che però di nuovo non sembra alterarne sostanzialmente la giocabilità.

Stilare una dimostrazione rigorosa del teorema dato richiederebbe la definizione rigorosa di cos'è un gioco, e ciò esula dal presente lavoro; tuttavia, possiamo fornirne un'idea:

Dimostrazione. Supponiamo per semplicità che i giocatori siano 2 ed i loro turni si alternino in modo ciclico. Assegniamo un numero ad ogni possibile mossa (ad esempio nel caso del Chomp è sufficiente numerare i quadretti, nel caso del Nim le monete, nel caso del go le intersezioni del goban), e sia R un algoritmo per stabilire in tempo n^k se una data mossa è lecita.

Costruiamo una macchina di Turing alternante che:

1. scriva non deterministicamente su di un nastro tutte le possibili stringhe di n^k numeri da 1 ad n^k, alternando stati esistenziali ed universali (per cui è in uno stato esistenziale quando scrive il primo numero, universale quando scrive il secondo e così via),

2. interpreti i numeri scritti (o la prima parte dei numeri scritti) come la successione di mosse di una partita, e verifichi prima di tutto che la partita sia valida, simulandone lo svolgimento e richiamando una macchina che implementi R su ogni mossa;

3. se la partita è valida, ne calcoli l'esito, ed accetti se e solo se essa risulta vincente per il primo giocatore.

Il primo passaggio potrà essere effettuato in tempo $n^k log(n)$, il secondo in tempo $n^k \times n^k = n^{2k}$, il terzo in tempo n^k, quindi la macchina opererà globalmente in tempo polinomiale, ed accetterà se e solo se esiste una strategia vincente per il primo giocatore.
Generalizzare questa costruzione ad un qualsiasi numero di giocatori ed una qualsiasi sequenza dei turni di gioco significa semplicemente cambiare la partizione degli stati utilizzati nella prima fase tra esistenziali ed universali; ad esempio, se si alternano ciclicamente 3 giocatori, sarà sufficiente stabilire che solo uno stato ogni 3 sia esistenziale. □

Sebbene quanto visto esaurisca l'analisi del problema decisionale legato all'invincibilità, è interessante osservare che una computazione accettante di una

tale macchina descrive un'effettiva strategia vincente.

Questa dimostrazione, sebbene soltanto abbozzata, permette di apprezzare la particolare efficacia che il paradigma alternante risulta avere nello studio dei giochi; la classe $PSPACE$ è si identica a $APTIME$, ma descrivere un risultato come quello dato restando nel paradigma deterministico sarebbe stato sensibilmente più faticoso. In effetti, questa è la stessa osservazione che era stata fatta confrontando la macchina di Turing descritta nella sezione 5.2.1 con quella descritta nella sezione 5.1.1: l'alternanza si presta naturalmente allo studio dei problemi in cui vi siano molte possibilità da sondare e molti quantificatori più o meno espliciti sulle possibilità, mentre realizzare algoritmi deterministici che risolvano gli stessi problemi rispettando limiti di spazio significa non soltanto (ovviamente) ammettere la possibilità di tempo esponenziale, ma anche essere costretti ad affrontare eventuali fastidiosi "dettagli di progettazione" che fanno spesso sembrare gli algoritmi, appunto, una sorta di emulazione del paradigma alternante in sé.

6.2 Il caso del go

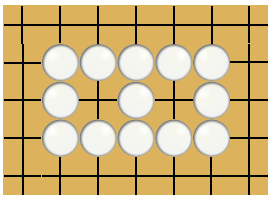
I 3 giochi riportati sinora come esempio sono drasticamente diversi dal punto di vista della risolubilità:

- il Nim è sostanzialmente un gioco risolto: un semplice ragionamento, basato sulla rappresentazione binaria delle posizioni, ne individua le condizioni iniziali in cui il primo giocatore ha una strategia vincente e suggerisce anche le mosse che deve fare per vincere,
- la situazione del Chomp è meno chiara: si dimostra, grazie ad un *argomentazione a "furto di strategia"*, che nella posizione iniziale il primo giocatore ha una strategia vincente (si dimostra infatti che se il secondo l'avesse, l'avrebbe anche il primo); tuttavia, tale ragionamento per assurdo non aiuta effettivamente ad individuare una buona strategia, e sopra ad una certa dimensione il calcolo esaustivo delle possibili partite è impraticabile per un calcolatore,
- nel caso del go, non si sa in generale (ma neanche nel caso particolare con $n = 19$) se nella posizione iniziale ci sia una strategia vincente per un giocatore o per l'altro, o per nessuno dei due.

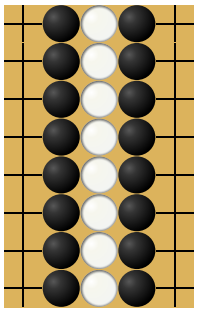
In realtà, un'analisi più approfondita del go rivela un risultato molto interessante: esso, come il gioco di ∃rnesto ed ∀ntonio, è un gioco PSPACE-completo; più precisamente, è PSPACE-completo il problema di stabilire se una generica disposizione di pietre sul goban costituisca o meno una posizione vincente per il giocatore a cui tocca muovere.

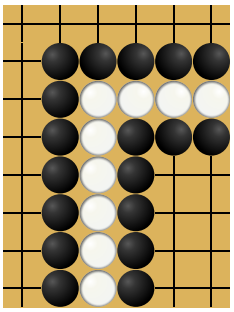
La dimostrazione di questo risultato, che qui riportiamo solo a grandi linee, è stata effettuata David Lichtenstein e Michael Sipser ([7]), dimostrando che al problema descritto può essere ricondotto polinomialmente QSAT: in sostanza, essa sfrutta particolari posizioni del GO che disegnano sul goban una sorta di circuito, disponendo le pietre in configurazioni che agiscono come delle porte

logiche e dei canali di comunicazione tra di esse; il giocatore bianco vince se e solo se riesce a collegare, tramite tale canale, un assembramento di sue pietre (sufficientemente vasto da garantirgli la vittoria) ad una zona del goban già definitivamente conquistata, come può essere la seguente:

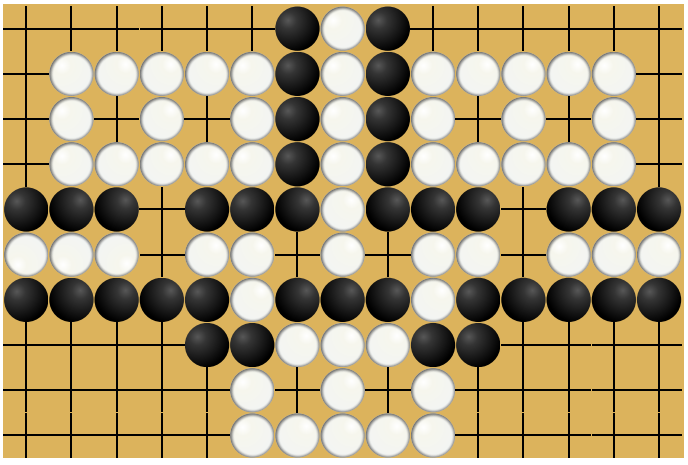


. Un canale potrà avere ad esempio le seguenti forme:





mentre a titolo di esempio riportiamo una raffigurazione della "porta logica" associata al quantificatore esistenziale: si assume che tocchi al giocatore bianco giocare e che il canale superiore sia collegato all'assembramento di pietre da salvare; chi conosce le regole del go verificherà facilmente che il nero ha una strategia per chiudere il ramo destro o quello sinistro, ma il bianco ha la scelta riguardo a quale salvare.



La dimostrazione si completa mostrando che ogni formula booleana quantificata può essere associata ad un opportuno circuito *planare* raffigurabile con tali "porte".

Appendice A

Definizioni e risultati intermedi

A.1 Alcune definizioni

A.1.1 Grafi ed alberi

Albero orientato

In letteratura, un albero viene spesso definito semplicemente come un grafo non orientato aciclico. Tuttavia, un grafo non orientato si sposa particolarmente male con una relazione, come quella di derivazione in un passo della definizione 4, intrinsecamente orientata. È conveniente quindi introdurre la nozione di albero orientato, che può essere definito come

"un grafo orientato aciclico tale che ogni vertice sia raggiungibile dalla radice"

o più semplicemente come

"un grafo orientato tale che ogni vertice sia raggiungibile in uno ed un solo modo dalla radice".

Albero etichettato

Nella definizione di un grafo, solitamente si parte da un insieme V che si sta studiando e che rappresenterà l'insieme dei vertici, e si stabilisce una certa relazione binaria R, che fisserà gli archi, su tale insieme. Tuttavia, nel caso delle computazioni di macchine alternanti e deterministiche, vogliamo potere *distinguere* una configurazione anche in base al procedimento che ha portato ad essa; per fare ciò, sfruttiamo la nozione di *albero etichettato*: un albero sarà un oggetto

$$(n, f, X, R) ,$$

dove f è una funzione da $\{1 \dots n\}$ in X, n indica il numero dei nodi e

- ogni nodo è identificato con un numero da 1 ad n,
- ad ogni nodo i è assegnato, in modo univoco ma *non bigettivo*, un elemento $f(i) \in X$,

- c'è un arco tra un nodo ed un altro se i rispettivi elementi di X sono in relazione.

A.2 Regole per portare le formule quantificate in forma prenessa

Data una formula booleana quantificata φ, dove le variabili quantificate sono $x_1 \dots x_m$, è possibile portarla in forma prenessa:

$$\mathcal{Q}_1 x_1 \mathcal{Q}_2 x_2 \dots \mathcal{Q}_m x_m \; \varphi' \; ,$$

dove $\mathcal{Q}_i \in \{\forall, \exists\}$ e φ' non contiene alcun quantificatore.

Per dimostrare ciò per induzione sulla struttura della formula, diamo delle opportune regole:

- sia
$$\varphi \equiv \varphi_1 \; \ominus \; \mathcal{Q} y \; \varphi_2 \; ,$$
dove φ_1, φ_2 sono già in forma prenessa, con $\mathcal{Q}$ indichiamo al solito un generico quantificatore e con $\ominus$ indichiamo un generico operatore binario. È sufficiente scegliere una nuova variabile y' che non appaia né in φ_1 né in φ_2, e sostituire in φ_2 (*non* in φ_1) ogni istanza di y con y', ottenendo una formula φ_2'; allora la forma prenessa di φ sarà:
$$\mathcal{Q} y' (\varphi_1 \ominus \varphi_2')$$

- sia
$$\varphi \equiv \neg \exists y \; \varphi_1 \; ;$$
allora la forma prenessa di φ sarà
$$\varphi \equiv \forall y \; \neg \varphi_1 \; ,$$

- viceversa, se
$$\varphi \equiv \neg \forall y \; \varphi_1 \; ,$$
la forma prenessa sarà
$$\varphi \equiv \exists y \; \neg \varphi_1 \; .$$

Si osserva banalmente che l'applicazione per induzione di tali regole non aumenta la lunghezza di una formula.

Appendice B

Alcuni teoremi noti

B.1 Risultati classici

B.1.1 Teorema di Savitch

Teorema 14 (Teorema di Savitch, 1970 [10])**.** *Sia* $f : \mathbb{N} \to \mathbb{N}$ *una funzione definitivamente maggiore di* $\log n$*: allora*

$$NSPACE(f(n)) \subset DSPACE(f(n)^2) \ .$$

La dimostrazione di tale risultato è omessa, ma ci interessa il suo:

Corollario 15.

$$NPSPACE = PSPACE \ ,$$

che si deriva dal fatto che $DSPACE(f(n))$ è banalmente inclusa in $NSPACE(f(n))$, e da semplici ragionamenti insiemistici.

B.1.2 Teorema di gerarchia

Teorema 16 (Teorema di gerarchia per tempo deterministico: Hartmanis e Stearns, 1965 [4])**.** *Sia* $f : \mathbb{N} \to \mathbb{N}$ *una funzione costruibile; allora esiste un problema decisionale* S *tale che*

$$S \notin DTIME(f(n))$$

ma

$$S \in DTIME(f(n)^2) \ ;$$

in altri termini, il teorema dice che

$$DTIME(f(n)) \subsetneq DTIME(f(n)^2) \ .$$

Anche di questo teorema non forniamo la dimostrazione, ma ne citiamo la conseguenza che più ci interessa:

Corollario 17.

$$P \subsetneq EXP \ ,$$

la cui dimostrazione deriva dal fatto che esiste un problema risolvibile in tempo 2^{2n} ma non in tempo 2^n, ovvero appartenente ad EXP ma certamente non a P.

Esistono alcuni teoremi di gerarchia anche per classi di complessità legate allo spazio, e ai paradigmi non deterministici.

B.2 Emulazione di macchine multinastro

Teorema 18. *Se un problema S è risolvibile da una macchina di Turing a k nastri M in tempo $f(n)$, allora esiste una macchine di Turing mononastro M' che lo risolve in tempo $f(n)^2 + n$ utilizzando lo stesso spazio.*

Dimostrazione. Sia

$$M(\Sigma, Q, q_i, q_f, \delta) \ ;$$

creeremo una macchina M' che, grazie ad un alfabeto più grande di quello di M, sia capace di contenere in una sola cella le informazioni di k celle, ovvero in un nastro le informazioni di k nastri. M' deve però anche emulare i cursori di M, che sono k. La posizione di un cursore, e a maggior ragione di k cursori, non è un'informazione di tipo finito, dato che può crescere arbitrariamente; non potrà quindi essere mantenuta negli stati, ma nel nastro, modificando opportunamente l'alfabeto in modo tale che ogni cella "sappia" se e quali cursori la stanno puntando.
A causa della possibilità che hanno k cursori di muoversi discordemente, dovremo modificare pesantemente anche le regole della macchina: per effettuare un singolo passo di M sarà necessario scandire tutta la stringa di lavoro di M' alla ricerca dei cursori, ovvero delle celle da considerare; questa sarà l'origine della complessità quadratica.
$M' = (\bar{\Sigma}, \bar{Q}, \bar{q}_i, \bar{q}_f, \bar{\delta})$ sarà definita come segue:

$$\bar{\Sigma} = \Sigma^k \times \mathcal{P}(\{\uparrow_1, \dots \uparrow_k\}) \cup \{\sqcup, \triangleright\}$$

(come detto, ogni cella rappresenta k simboli più l'informazione sugli eventuali cursori che la stanno puntando) e

$$\bar{Q} = \{q_i, \dashleftarrow, \dashrightarrow, q_f\} \cup \left(\{\dashleftarrow, \dashrightarrow\} \times \Sigma^k \times Q \times Q_\uparrow\right) \ ,$$

dove:

$$Q_\uparrow = \{\downarrow\} \cup \left(\{\overleftrightarrow{\rightarrow}\} \times \mathcal{P}(\{1 \dots k\}) \times \{\overleftrightarrow{\leftarrow}\} \times \mathcal{P}(\{1 \dots k\})\right) \ ,$$

ovvero, a parte 4 stati accessori necessari in fasi particolari dell'esecuzione, ogni stato contiene informazioni riguardo a:

1. una direzione (ma anche una modalità di operazione: per emulare ogni passo di M, M' si sposterà prima a destra lungo il nastro, in modalità "lettura"; giunta in fondo al contenuto del nastro, avrà acquisito abbastanza informazioni da poter emulare il passo di M', e si muoverà quindi a sinistra, in modalità "scrittura"),

2. una particolare k-upla di simboli in Σ, che M' collezionerà nella modalità "lettura" (saranno i simboli puntati dai cursori di M), cambierà in funzione di δ e quindi scriverà nella modalità "scrittura",

3. uno stato di M (M' deve emularla, quindi in particolare deve conoscere lo stato in cui sarebbe),

4. un elemento di $Q_\uparrow$, in cui $\downarrow$ non porta nessuna particolare informazione, mentre gli altri indicano i cursori che vanno riportati a sinistra e quelli che vanno invece spostati a destra.

La funzione di transizione sarà quindi definita in base alle seguenti regole:

- inizialmente, sarà necessario predisporre il nastro in modo che esso possa accogliere le informazioni sugli altri nastri senza perdere le sue, ovvero ricodificarlo nei simboli del nuovo alfabeto; nello stato iniziale q_i, la macchina leggerà un simbolo σ, scriverà il simbolo

$$(\sigma, \underbrace{\sqcup, \cdots \sqcup}_{k-1}, \{\uparrow_1, \dots \uparrow_k\})$$

  (all'inizio tutti i cursori sono sulla prima cella), si sposterà a destra ed entrerà nello stato $\dashrightarrow$.

- nello stato $\dashrightarrow$, finché la macchina incontrerà un simbolo $\sigma \in \Sigma$, essa lo sostituirà con un il simbolo

$$(\sigma, \underbrace{\sqcup, \cdots \sqcup}_{k-1}, \emptyset) \ ;$$

  non appena troverà una cella vuota, ovvero quando avrà finito la ricodifica dell'input, si sposterà invece a sinistra ed entrerà nello stato $\dashleftarrow$. In questo stato, la macchina non farà che spostare il cursore a sinistra riscrivendo ogni volta il simbolo letto, fino a ritornare ad inizio nastro.

Arrivata all'inizio del nastro (dopo $2n$ passi dall'inizio), la macchina entrerà nello stato

$$(\dashrightarrow, q_i, \underbrace{\sqcup, \cdots \sqcup}_{k}, \downarrow) \ ,$$

ed inizierà le scansioni vere e proprie della stringa (ricordiamo che ogni scansione "avanti-indietro" servirà ad emulare un passo di M'). Negli stati appartenenti a $\{\dashrightarrow\} \times \Sigma^k \times Q \times Q_\uparrow$, in cui la macchina acquisisce informazioni, essa seguirà le seguenti regole:

- se legge un simbolo che non riporta l'indicazione di alcun cursore, ovvero appartenente a:

$$\Sigma^k \times \{\emptyset\} \ ,$$

  essa si sposterà a destra senza cambiare né stato né simbolo.

- In caso contrario (se il simbolo riporta l'indicazione di un cursore), "memorizzerà" il simbolo (o i simboli) corrispondente al cursore letto (o i cursori letti); ad esempio, se in uno stato

$$(\dashrightarrow, \underbrace{\sqcup}_{k}, q, \downarrow) ,$$

essa leggerà il simbolo:

$$(\sigma^1, \sigma^2, \ldots, \sigma^k, \{\downarrow_2, \downarrow_k\}) ,$$

che indica la presenza del secondo e del k-esimo cursore, essa dovrà memorizzare il secondo ed il k-esimo simboli letti, per cui si sposterà nello stato

$$(\dashrightarrow, \sqcup, \sigma^2, \sqcup, \ldots, \sqcup, \sigma^k, q, \downarrow) ,$$

riscriverà il simbolo letto (non abbiamo ancora sufficienti informazioni per dedurre il comportamento che avrebbe M in tale situazione) e si sposterà a destra.

- Se invece trova una cella vuota, ovvero avrà finito la scansione, essa si troverà in uno stato

$$(\dashrightarrow, \sigma^1, \sigma^2, \ldots \sigma^k, q, \downarrow) ;$$

a quel punto, essa entrerà nello stato $(\dashleftarrow, \sigma^{1\prime}, \sigma^{2\prime}, \ldots \sigma^{k\prime}, q', \mu)$, dove i nuovi simboli $\sigma^{i\prime}$, lo stato q' e l'indicatore degli spostamenti dei cursori μ sono quelli dati da δ (la funzione di transizione di M) applicata a

$$((\sigma^1, \sigma^2, \ldots \sigma^k), q)$$

(ad esempio μ sarà ancora $\downarrow$ se la configurazione è tale che M non sposterebbe alcun cursore) e si sposterà a sinistra, apprestandosi a scrivere sul nastro le informazioni memorizzate.

Negli stati appartenenti a $\{\dashleftarrow\} \times \Sigma^k \times Q \times Q_\uparrow$, la macchina M' si comporterà quindi come segue:

- se il simbolo letto non riporta l'indicazione di alcun cursore, riscriverà lo stesso simbolo, resterà nello stesso stato e si sposterà a sinistra;

- altrimenti, modificherà il simbolo letto aggiungendo l'informazione aggiornata riguardo a quei nastri "virtuali" di cui la cella "contiene" il puntatore: ad esempio, supponiamo che nello stato

$$(\dashleftarrow, \sigma^{1\prime}, \ldots \sigma^{k\prime}, q', \downarrow)$$

essa legga un simbolo della forma

$$(\tau^1, \tau^2, \tau^3, \ldots, \tau^k, \{\uparrow_3\}) \ :$$

essa lo sostituirà con

$$(\tau^1, \tau^2, \sigma^{3\prime}, \ldots, \tau^k, \{\uparrow_3\}) ,$$

"liberando" memoria, ovvero entrando nello stato

$$(\leftarrow\!\!-, \sigma^{1\prime}, \sigma^{2\prime}, \sqcup, \sigma^{4\prime}, \dots \sigma^{k\prime}, q', \downarrow)$$

e quindi continuando il suo spostamento a sinistra.
Dovranno però essere scritte anche le informazioni sullo spostamento del cursore 3; se ad esempio, invece di $\downarrow$, lo stato avesse avuto

$$(\overrightarrow{\leftarrow}, \{1,3\}, \overleftarrow{\rightarrow}\{k\}) \ ,$$

il simbolo scritto non sarebbe più stato

$$(\tau^1, \tau^2, \sigma^{3\prime}, \dots, \tau^k, \{\uparrow_3\}) \ ,$$

ma

$$(\tau^1, \tau^2, \sigma^{3\prime}, \dots, \tau^k, \emptyset) \ ;$$

invece, il cursore (quello vero) avrebbe dovuto spostarsi a destra e lì aggiungere l'informazione sulla presenza del cursore 3 (uno stato con l'informazione sul i-esimo cursore ma senza i-esimo simbolo memorizzato indica appunto ciò); quindi, avrebbe normalmente continuato lo spostamento a sinistra.

A questo punto M' avrà emulato un passo di M in tempo al più $f(n)+k$: l'unica attenzione che bisogna ancora avere riguarda la terminazione. È sufficiente aggiungere un'ultima serie di regole della forma:

$$\bar{\delta}(*, q_f, * \dots) = q_f$$

per garantire che la M' termini se e solo se termina M. □

La dimostrazione dei casi nondeterministico e alternante può essere affrontata in modo assolutamente analogo, anche se in realtà nel caso alternante si può ottenere risultati più forti, che in questo contesto non ci interessano.

Bibliografia

[1] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 1981.

[2] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 2004.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.

[4] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 1965.

[5] Neil Immerman. *Descriptive complexity*. Springer, 1999.

[6] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 1972.

[7] David Lichtenstein and Michael Sipser. Go is polynomial-space hard. *Journal of the ACM*, 1980.

[8] Christos H. Papadimitriou. *Computational Complexity*. 1994.

[9] Paul Rendell. Turing machine implemented in conway's game of life, April 2000. `http://rendell-attic.org/gol/tm.htm`.

[10] Walter Savitch. Relationship between nondeterministic and deterministic tape classes. *Journal of Computer and System Sciences*, 1970.

[11] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, Steve Sutphen David Lichtenstein, and Michael Sipser. Checkers is solved. *Science*, 2007.

[12] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. *Proceedings of the 5th ACM Symposium on the Theory of Computing*, 1973.