Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

Dispense del corso

di

Informatica Teorica

(A.A. 2000/2001)

Alberto Bertoni
Carlo Mereghetti

Circuiti booleani e macchine di Turing

1 Funzioni booleane

Una *funzione booleana* a n argomenti e m valori è una funzione del tipo

$$f : \{0,1\}^n \to \{0,1\}^m.$$

Chiameremo $\mathcal{B}_{n,m}$ l'insieme di tali funzioni e diremo che n è la loro *arietà* (o *arità*). Abbrevieremo con $\mathbf{x}$ il vettore di variabili $[x_1, \ldots, x_n]$. Una funzione f in $\mathcal{B}_{n,m}$ può essere rappresentata dalla tabella $(\mathbf{x}, f(\mathbf{x}))$:

x_1	x_2	$\ldots$	x_n	f_1	$\ldots$	f_m
0	0	$\ldots$	0	$f_1(0,\ldots,0)$	$\ldots$	$f_m(0,\ldots,0)$
0	0	$\ldots$	1	$f_1(0,\ldots,1)$	$\ldots$	$f_m(0,\ldots,1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ldots$	$\ldots$	$\ldots$
1	1	$\ldots$	1	$f_1(1,\ldots,1)$	$\ldots$	$f_m(1,\ldots,1)$

Essa può quindi essere vista come una m–tupla di funzioni $\langle f_1, \ldots, f_m \rangle$ ciascuna ad n argomenti e a un valore. La classe $\mathcal{B}_{n,1}$, che in seguito denoteremo con $\mathcal{B}_n$, assume dunque particolare rilievo. Osserviamo che la tabella che rappresenta una funzione f in $\mathcal{B}_n$ può essere interpretata come un vettore di 2^n componenti in $\{0,1\}$: $\mathcal{B}_n$ contiene quindi 2^{2^n} funmioni. Pnù in generale, è semplice verificare che $\mathcal{B}_{n,m}$ contiene $2^{m \cdot 2^n}$ funzioni.

Esempio 1.1 $\mathcal{B}_1$ contiene quattro funzioni: le *costanti* 0 e 1, l'*identità* e la *negazione* (NOT, $\bar{}$): $\overline{x} = 1$ sse $x = 0$.

Esempio 1.2 $\mathcal{B}_2$ contiene sedici funzioni; ricordiamo qui la *congiunzione* (AND, $\wedge$), la *disgiunzione* (OR, $\vee$) e la *disgiunzione esclusiva* (XOR, $\oplus$).

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Richiamiamo brevemente due importanti strutture algebriche basate su tali operazioni.

1. La struttura algebrica $\langle \{0,1\}, \wedge, \vee, \overline{\ } \rangle$ è un'*algebra booleana*; valgono infatti le seguenti proprietà:

Associativa	$a \vee (b \vee c) = (a \vee b) \vee c$	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$
Commutativa	$a \vee b = b \vee a$	$a \wedge b = b \wedge a$
Idempotenza	$a \vee a = a$	$a \wedge a = a$
Unità	$a \vee 0 = a$	$a \wedge 1 = a$
Complemento	$a \vee \overline{a} = 1$	$a \wedge \overline{a} = 0$
Distributiva	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
Assorbimento	$a \vee (a \wedge b) = a$	$a \wedge (a \vee b) = a$
De Morgan	$\overline{a \vee b} = \overline{a} \wedge \overline{b}$	$\overline{a \wedge b} = \overline{a} \vee \overline{b}$

2. La struttura algebrica $\langle \{0,1\}, \oplus, \wedge \rangle$ è il *campo di Galois* di ordine 2. Essa è infatti isomorfa al campo $\mathbf{Z}_2$ dei resti modulo 2, con somma e prodotto (modulo 2).

Osserviamo infine che ogni funzione booleana $f : \{0,1\}^n \to \{0,1\}$ è individuata dalla controimmagine $f^{-1}(1) = \{\mathbf{x} \in \{0,1\}^n \mid f(\mathbf{x}) = 1\}$. Poiché $\{0,1\}^n$ può essere visto come l'insieme dei vertici di un ipercubo in $\mathbf{R}^n$, talvolta può essere utile vedere f come il sottoinsieme $f^{-1}(1)$ dei vertici dell' ipercubo stesso. A tal proposito, consideriamo il seguente esempio:

Esempio 1.3 Una funzione f in $\mathcal{B}_n$ è detta *linearmente separabile* se $f^{-1}(1)$ può essere separato da $f^{-1}(0)$ mediante un iperpiano in $\mathbf{R}^n$. Se $\sum_{k=1}^{n} w_k \cdot x_k - a = 0$ è l'equazione dell'iperpiano separatore, possiamo scrivere:

$$f(x_1, \ldots, x_n) = \mathrm{HS}\left(\sum_{k=1}^{n} w_k \cdot x_k - a \right),$$

dove

$$\mathrm{HS}(r) = \begin{cases} 1 & \text{se } r > 0 \\ \\ 0 & \text{altrimenti.} \end{cases}$$

È facile osservare che AND e OR sono linearmente separabili, mentre XOR non lo è.

2 Programmi lineari, circuiti logici e formule

Se rappresentiamo la funzione booleana f in $\mathcal{B}_n$ con la tabella $(\mathbf{x}, f(\mathbf{x}))$, il calcolo di $f(\mathbf{x})$ può essere immediatamente effettuato ispezionando la tabella in posizione $\mathbf{x}$. Il problema è che tali tabelle sono vettori booleani di dimensione 2^n, e per n abbastanza grandi la loro memorizzazione risulta praticamente impossibile.

Una diversa rappresentazione di f può essere ottenuta attraverso un programma di calcolo della f stessa. Se la dimensione del programma è molto minore di 2^n e le risorse computazionali necessarie ad eseguire il programma sono ragionevoli, tale rappresentazione

risulta essere un'eccellente alternativa. Descriviamo allora un semplicissimo linguaggio di programmazione per il calcolo di funzioni booleane in $\mathcal{B}_n$. A tal riguardo, supponiamo di fissare un insieme finito Ω di funzioni booleane che chiameremo *base* (in seguito ci riferiremo generalmente alla base $\Omega = \{\wedge, \vee, \bar{}\}$).

Definizione 2.1 *Data una base di funzioni booleane Ω e un insieme $X = \{x_1, \ldots, x_n\}$ di variabili booleane, un* Ω–programma lineare *P con variabili interne $z_1, \ldots, z_b$ è una sequenza di istruzioni*

$$\begin{aligned} &Istr_1;\\ &Istr_2;\\ &\vdots\\ &Istr_b. \end{aligned}$$

in cui la generica istruzione $Istr_k$, per ogni $1 \leq k \leq b$, ha la forma

$$z_k := h(g_1, \ldots, g_a),$$

dove h è una funzione in Ω di arietà a, mentre g_j può essere: una costante (0 oppure 1), una variabile x_j in X ($1 \leq j \leq n$) o una variabile interna z_j ($1 \leq j < k$, osserviamo che la limitazione $j < k$ evita di avere dei cicli all'interno di P). La semantica *si ottiene associando ad ogni elemento q in $\{0, 1, x_1, \ldots, x_n, z_1, \ldots, z_b\}$ una funzione booleana* $\mathrm{ris}_q(\mathbf{x})$ *definita come segue:*

- *alle costanti 0 e 1 associamo le funzioni booleane (costanti)* $\mathrm{ris}_0(\mathbf{x}) = 0$ *e* $\mathrm{ris}_1(\mathbf{x}) = 1$,
- *ad ogni variabile x_j ($1 \leq j \leq n$) associamo la funzione booleana (proiezione) definita come* $\mathrm{ris}_{x_j}(\mathbf{x}) = x_j$,
- *ad ogni variabile interna z_k ($1 \leq k \leq b$) associamo la funzione booleana* $\mathrm{ris}_{z_k}(\mathbf{x})$ *definita induttivamente da*

$$\mathrm{ris}_{z_k}(\mathbf{x}) = h(\mathrm{ris}_{g_1}(\mathbf{x}), \ldots, \mathrm{ris}_{g_a}(\mathbf{x})),$$

  se $Istr_k$ è $z_k := h(g_1, \ldots, g_a)$.

La funzione booleana calcolata dal programma *P è allora* $\mathrm{ris}_{z_b}(\mathbf{x})$.

Nella precedente definizione, un programma lineare calcola la funzione in $\mathcal{B}_n$ corrispondente alla variabile interna z_b, che si configura quindi come variabile di *uscita*. La semplice estensione per il calcolo di funzioni in $\mathcal{B}_{n,m}$ può essere ottenuta elencando m variabili scelte in $\{z_1, \ldots, z_b\}$, da interpretare come variabili di uscita. I programmi lineari si differenziano dai comuni programmi per elaboratori per il fatto che non contengono istruzioni di salto.

È utile interpretare i programmi lineari come grafi orientati etichettati. Più precisamente, ad ogni Ω–programma su variabili $X = \{x_1, \ldots, x_n\}$, possiamo associare un grafo orientato i cui nodi sono etichettati come segue:

- I nodi a grado di entrata nullo sono etichettati dalle variabili in X o dalle costanto 0 o 1 e vengono detti *nodi di input*.

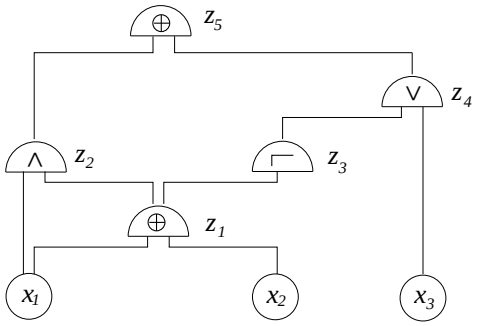
- Per ogni istruzione del tipo $z_k := h(g_1, \ldots, g_a)$, esiste un nodo etichettato $h \in \Omega$ ed un arco dal nodo per g_j $(1 \leq j \leq a)$ al nodo per z_k. I nodi di questo tipo vengono detti *porte interne*.
- I nodi a grado di uscita nullo vengono designati come *nodi di output*.

È facile verificare che i grafi associati a programmi lineari sono privi di cicli; chiameremo tali grafi *circuiti logici* o *reti combinatorie*. e diremo che un Ω–circuito è un circuito associato ad un Ω–programma lineare.

Esempio 2.1 Si consideri il seguente programma lineare:

$$\begin{aligned} z_1 &:= x_1 \oplus x_2; \\ z_2 &:= x_1 \wedge z_1; \\ z_3 &:= \overline{z_1}; \\ z_4 &:= z_3 \vee x_3; \\ z_5 &:= z_2 \oplus z_4. \end{aligned}$$

Ad esso è associato il circuito:



Il *fan–in di un nodo* z_k è il numero di archi entranti in z_k, mentre il suo *fan–out* è il numero di archi uscenti. Il *fan–in (fan–out) di un circuito* è il massimo fan–in (fan–out) delle porte interne. Circuiti con fan-out 1 sono chiamati *formule*. È semplice provare che tali circuiti ammettono la seguente definizione induttiva:

- 0, 1, x_j sono formule,
- se h è una funzione in Ω di arietà a ed $F_1, \ldots, F_a$ sono formule, allora anche $h(F_1, \ldots, F_a)$ è una formula.

Essenzialmente, i grafi associati a formule sono alberi e possono essere descritti come espressioni logiche. Dato un circuito, è facile costruire una formula equivalente (che calcola cioè la stessa funzione). Il numero di nodi della formula può però crescere esponenzialmente rispetto a quello del grafo di partenza.

Esempio 2.2 Consideriamo qui varie sottoclassi di formule sulla base $\{\wedge, \vee, \overline{\ }\}$. Un *letterale* è una variabile x_j o la sua negazione $\overline{x_j}$; una *clausola* è una disgiunzione $l_1 \vee l_2 \vee \ldots \vee l_m$ di letterali, mentre una congiunzione di letterali $l_1 \wedge l_2 \wedge \ldots \wedge l_m$ è detta *monomio*. Una *forma congiunta* è una congiunzione di clausole mentre una *forma disgiunta* è una disgiunzione di monomi.

3 Misure di complessità

Data una base, è possibile che la stessa funzione sia calcolata da molti circuiti differenti; siamo in generale interessati a determinare il circuito *più efficiente* per calcolarla. I criteri che possono essere adottati per definire tale nozione di efficienza sono molteplici, ma due caratteristiche sicuramente molto significative sono:

1. il costo di costruzione del circuito,
2. il tempo impiegato dal circuito per generare il segnale in uscita dal momento in cui ha ricevuto i vari segnali in ingresso.

Per essere più precisi, distinguiamo l'esecuzione *sequenziale* di un programma lineare da quella *parallela*:

- Nell'esecuzione sequenziale, ad ogni istante di tempo viene eseguita una singola istruzione da un unico processore. Supponendo per semplicità che ogni istruzione richieda lo stesso tempo per essere eseguita, una ragionevole misura del tempo di calcolo in computazioni sequenziali è il numero di istruzioni del programma o, equivalentemente, il numero di porte interne del circuito logico.
- Nell'esecuzione parallela, molte istruzioni possono essere eseguite allo stesso istante su processori diversi; questo è il caso, ad esempio, dell'esecuzione dei circuiti in logica cablata. Se ipotizziamo che il tempo di esecuzione sia costante per ogni istruzione e che i tempi di comunicazione siano trascurabili, una ragionevole misura del tempo di esecuzione parallelo è la lunghezza del più lungo cammino nel circuito da un nodo di ingresso a uno di uscita.

Queste osservazioni rendono naturale la seguente:

Definizione 3.1 *La* dimensione *o* complessità $C(P)$ *di un circuito P è il numero di porte interne di P. La* profondità $D(P)$ *di P è la lunghezza del più lungo cammino in P.*

Quindi, $C(P)$ è una stima del costo di costruzione del circuito o del tempo di esecuzione sequenziale, mentre $D(P)$ si riferisce al tempo di esecuzione in caso di arbitrario grado di parallelismo.

Data una base Ω e una funzione booleana f, è ragionevole tentare di costruire, se possibile, il circuito che, fra quelli che calcolano f, minimizza la dimensione e/o la profondità. Questo porta alla seguente:

Definizione 3.2 *La* complessità di circuito $C_\Omega(f)$ *di f rispetto alla base Ω è la dimensione del più piccolo Ω–circuito che calcola f. La* profondità $D_\Omega(f)$ *di f rispetto alla base Ω è la profondità del meno profondo Ω–circuito che calcola f.*

Dunque, dalla definizione 3.2, si osserva che le due nozioni di complessità e profondità dipendono dalla base scelta. Date due basi Ω e Ω', si può tentare di trasformare un Ω–circuito in un Ω'–circuito semplicemente esprimendo ogni h in Ω con un circuito in Ω'. Questo può essere sicuramente fatto per *basi complete*, dove una base Ω è detta *completa* se ogni funzione booleana può essere calcolata da un Ω–circuito. Ne consegue che per basi complete $C_\Omega(f)$ e $C_{\Omega'}(f)$ (e analogamente $D_\Omega(f)$ e $D_{\Omega'}(f)$) differiscono al più per una costante moltiplicativa:

Fatto 3.1 *Siano Ω e Ω' due basi complete. Allora*

$$C_{\Omega'}(f) = O(C_{\Omega}(f)) \quad e \quad D_{\Omega'}(f) = O(D_{\Omega}(f)).$$

Può essere talvolta interessante restringere l'attenzione a sottoclassi di circuiti, per esempio alle formule o, nel caso di base $\{\wedge, \vee, \bar{}\}$, alle forme disgiunte (vedi esempio 2.2). Consideriamo qui la seguente:

Definizione 3.3 *La* dimensione di formula $L_{\Omega}(f)$ *di f rispetto alla base Ω è la dimensione della più piccola formula sulla base Ω che calcola f.*

In seguito denoteremo con $C(f)$, $D(f)$ e $L(f)$ rispettivamente la complessità di circuito, la profondità e la dimensione di formula di f rispetto alla base $\{\wedge, \vee, \bar{}\}$. *Si osservi che mentre per basi complete Ω, le misure $C_{\Omega}(f)$ e $D_{\Omega}(f)$ differiscono da $C(f)$ e $D(f)$ per una semplice costante moltiplicativa, questo non è necessariamente vero per $L_{\Omega}(f)$ e $L(f)$.*

4 Forme disgiunte e minimizzazione

La base $\{\wedge, \vee, \bar{}\}$ è un esempio di base completa. Consideriamo infatti, per ogni vettore $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ in $\{0,1\}^n$, il monomio

$$m_{\mathbf{c}}(\mathbf{x}) = x_1^{c_1} \wedge x_2^{c_2} \wedge \ldots \wedge x_n^{c_n},$$

dove, per convenzione, poniamo ${x_j}^0 = \overline{x_j}$ e ${x_j}^1 = x_j$. Vale che:

$$m_{\mathbf{c}}(\mathbf{x}) = \begin{cases} 1 & \text{se } \mathbf{x} = \mathbf{c} \\ \\ 0 & \text{altrimenti.} \end{cases}$$

Questo prova il seguente

Fatto 4.1 *Per ogni funzione booleana $f : \{0,1\}^n \to \{0,1\}$ vale:*

$$f(\mathbf{x}) = \bigvee_{\{\mathbf{c} \in \{0,1\}^n | f(\mathbf{c}) = 1\}} m_{\mathbf{c}}(\mathbf{x}).$$

Il precedente risultato dimostra che ogni funzione booleana può essere rappresentata da una formula in forma disgiunta e quindi, a fortiori, che *la base $\{\wedge, \vee, \bar{}\}$ è completa.*

Affrontiamo ora il problema di determinare la *forma disgiunta minima* per f, detta *polinomio minimo.* Dato un monomio m, diremo che esso è un *implicante* di f se

$$\forall \mathbf{x} \in \{0,1\}^n \quad (m(\mathbf{x}) = 1 \Rightarrow f(\mathbf{x}) = 1);$$

diremo inoltre che m è un *implicante primo* di f se m è un implicante di f e nessun sottomonomio proprio di m è un implicante di f.

Consideriamo ora la forma disgiunta per f di dimensione minima, $\bigvee_{m\in A} m(\mathbf{x})$, dove A è l'insieme di monomi la cui disgiunzione calcola f: ogni m in A dev'essere un implicante primo. Se infatti m non fosse un implicante, esisterebbe $\mathbf{c}$ per cui $m(\mathbf{c}) = 1$ e contemporaneamente $f(\mathbf{c}) = 0$; allora

$$1 = m(\mathbf{c}) = \bigvee_{m\in A} m(\mathbf{c}) = f(\mathbf{c}) = 0,$$

il che è assurdo. Supponiamo. inoltre, che A contenga un implicante m non primo, tale cioè che $m = m' \wedge m''$, con m' implicante a sua volta. Allora, si avrebbe $m \vee m' = m'$ e la sostituzione di m in A con m' porterebbe ad una forma disgiunta per f di dimensione minore. Per quanto visto, *i monomi costituenti il polinomio minimo per f sono implicanti primi di f*. Una ragionevole strategia per costruire il polinomio minimo per f consiste allora di due passi:

1. Calcolo degli implicanti primi di f.
2. Eliminazione degli implicanti primi ridondanti.

Per il passo (1) esistono efficienti algoritmi: presentiamo qui l'algoritmo di Quine–Mc Cluskey. Data f in $\mathcal{B}_n$, l'algoritmo permette di calcolare gli insiemi $I_k(f)$ e $IP_k(f)$, contenenti rispettivamente gli implicanti e gli implicanti primi con k letterali ($1 \le k \le n$). L'algoritmo di Quine–Mc Cluskey si basa sulle seguenti considerazioni:

(a) Gli implicanti contenenti n letterali sono i monomi $m_{\mathbf{c}}(\mathbf{x})$ per cui $f(\mathbf{c}) = 1$.

(b) Se il monomio m non contiene né x_j né $\overline{x_j}$, allora m è un implicante di f se e solo se $m \wedge x_j$ e $m \wedge \overline{x_j}$ lo sono.

Algoritmo (Quine–McCluskey)
INGRESSO: la tavola $(\mathbf{x}, f(\mathbf{x}))$ di una funzione booleana f in $\mathcal{B}_n$.

$I_n := \{m_{\mathbf{c}}(\mathbf{x}) \mid \mathbf{c} \in \{0,1\}^n \wedge f(\mathbf{c}) = 1\}$;
$k := n$;
while $I_k \neq \emptyset$ **do**
begin
$k := k - 1$;
$I_k := \{m \mid$ esiste j tale che m non contiene né x_j né $\overline{x_j}$
ma $m \wedge x_j$ e $m \wedge \overline{x_j}$ sono in $I_{k+1}\}$;
$IP_{k+1} := \{m$ in $I_{k+1} \mid$ per ogni m' in I_k, m' non è sottomonomio di $m\}$;
end

USCITA: l'insieme $IP(f) = \bigcup_k IP_k$ degli implicanti primi di f.

Procediamo ora ad una rapida stima del tempo di calcolo dell'algoritmo. Osserviamo per prima cosa che la dimensione dell'ingresso è $N = 2^n$. Esistono inoltre 3^n monomi su n variabili; nella formazione di un monomio esistono infatti tre possibilità per ognuna delle n variabili x_j: assenza dal monomio, presenza come x_j, presenza come $\overline{x_j}$.

Supponiamo inoltre di memorizzare gli elementi in I_k e IP_k in alberi bilanciati (esempio: 2–3 alberi) in modo che l'inserimento e la ricerca avvengano in tempi proporzionali al logaritmo degli elementi inseriti. Per costruire I_k conoscendo I_{k+1} è sufficiente, per ogni variabile x_j ed ogni monomio $m \wedge x_j$ in I_{k+1} contenente la variabile x_j, controllare se $m \wedge \overline{x_j}$ è in I_{k+1} e, in caso affermativo, inserire m in I_k ($n > k \geq 1$). Poiché ci sono n variabili, al più 3^n monomi in I_{k+1} e la ricerca costa quindi al più [1] $O(\log 3^n)$ passi, il tempo complessivo per passare da I_{k+1} a I_k è $O(n^2 \cdot 3^n)$. Analogo discorso vale nella costruzione di IP_k a partire da I_k. Il tempo complessivo è allora limitato da $O(n^3 \cdot 3^n)$; poiché $n = \log N$, rispetto alla dimensione N dell'ingresso il tempo è

$$O(N^{\log 3} \cdot (\log N)^3).$$

5 Complessità di circuito per "quasi tutte" le funzioni booleane

Una funzione booleana f può essere espressa come disgiunzione di al più 2^n monomi, ognuno dei quali contiene al più n letterali e $n-1$ operazioni $\wedge$. Ne segue che f può essere espressa da una formula (con operazioni $\wedge$, $\vee$ a due argomenti) di profondità $O(n)$ e contenente $O(n \cdot 2^n)$ operazioni. È possibile che passando dalla forma normale disgiunta ad un circuito per f si possano risparmiare operazioni: dimostriamo qui che esistono sempre circuiti sulla base $\{\wedge, \vee, \bar{}\}$ per f di dimensione $O\left(\frac{2^n}{n}\right)$.

Teorema 5.1 *Per $f \in \mathcal{B}_n$ vale che:*

$$C(f) \leq 13 \cdot \frac{2^n}{n} \qquad (n \to +\infty).$$

Dimostrazione. Consideriamo l'identità

$$f(x_1, \ldots, x_n) = [x_n \wedge f(x_1, \ldots, x_{n-1}, 1)] \vee [\overline{x_n} \wedge f(x_1, \ldots, x_{n-1}, 0)].$$

Essa permette di esprimere la generica $f \in \mathcal{B}_n$ con una formula contenente dei letterali, due funzioni in $\mathcal{B}_{n-1}$ e tre operazioni $\wedge$, $\vee$. Più in generale, riapplicando la proprietà a $f(x_1, \ldots, x_{n-1}, 1)$ e a $f(x_1, \ldots, x_{n-1}, 0)$, e così via, possiamo esprimere la generica $f \in \mathcal{B}_n$ con una formula F_k contenente dei letterali, al più 2^k funzioni del tipo $f(x_1, \ldots, x_{n-k}, c_{n-k+1}, \ldots, c_n)$ con $(c_{n-k+1}, \ldots, c_n) \in \{0,1\}^k$ e $3 \cdot (2^{n-k} - 1)$ operazioni $\wedge$, $\vee$. Un circuito ϕ_k per f può essere realizzato come segue:

1. Costruendo la formula G_α in forma disgiunta per ogni funzione f_α in $\mathcal{B}_k$.
2. Collegando le porte d'uscita di ogni formula G_α alle porte in F_k contenenti funzioni f_α tali che $f_\alpha = f(x_1, \ldots, x_k, c_{k+1}, \ldots, c_n)$.

Poiché ognuna delle 2^{2^k} funzioni in $\mathcal{B}_k$ è ottenibile con una formula di al più $k \cdot 2^k$ operazioni $\wedge$, $\vee$, il circuito ϕ_k ha al più $3 \cdot (2^{n-k} - 1) + 2^{2^k} \cdot 2^k \cdot k$ operazioni $\wedge$, $\vee$ ed al più $n-1$ operazioni $\bar{}$. Posto $k = \lceil \log n \rceil - 2$, otteniamo un circuito ϕ che calcola f e che ha al più $3 \cdot \frac{2^{n+2}}{n} \cdot (1 + o(1)) \leq 13 \cdot \frac{2^n}{n}$ operazioni $\wedge, \vee, \bar{}$ (per n sufficientemente grande). ∎

[1] D'ora innanzi, con log intenderemo $\log_2$.

Abbiamo, così mostrato che, almeno per n grande, ogni funzione $f \in \mathcal{B}_n$ può essere calcolata da circuiti di dimensione $13 \cdot \frac{2^n}{n}$. Mostriamo ora che "quasi tutte" le funzioni booleane richiedono almeno $\frac{1}{2} \cdot \frac{2^n}{n}$ componenti $\{\wedge, \vee, \bar{}\}$ per essere calcolate; qui il termine "quasi tutte" ha il seguente significato:

Definizione 5.1 *Data una proprietà P sulle funzioni booleane, si dice che* "quasi tutte" *le funzioni verificano P se vale:*

$$\lim_{n \to +\infty} \frac{|\{f | f \in \mathcal{B}_n \text{ e } f \text{ verifica } P\}|}{|\mathcal{B}_n|} = 1.$$

Il significato di "quasi tutte" è chiaramente asintotico: se n è abbastanza grande, la probabilità che una funzione, scelta a caso in $\mathcal{B}_n$, verifichi P è vicina a 1.

Teorema 5.2 *Per quasi tutte le funzioni booleane,*

$$C(f) \geq \frac{1}{2} \cdot \frac{2^n}{n},$$

essendo n l'arietà di f.

Dimostrazione. Utilizziamo una semplice tecnica introdotta da Shannon. Denotiamo con Π_g^n il numero di circuiti booleani per funzioni in n variabili di dimensione al più g. Ponendo $g = g(n)$, risulta chiaro che $|\{f \mid f \in \mathcal{B}_n \wedge f \text{ ha circuiti di dimensione } g(n)\}| \leq \Pi_{g(n)}^n$. Osservando, inoltre, che $|\mathcal{B}_n| = 2^{2^n}$, abbiamo che se vale

$$\lim_{n \to +\infty} \frac{\Pi_{g(n)}^n}{2^{2^n}} = 0,$$

allora per quasi tutte le funzioni risulta $C(f) \geq g(n)$, essendo n l'arietà di f. Procediamo ad una grossolana stima di Π_g^n: in un circuito l'istruzione di posto k è del tipo $z_k := op(g_i, g_j)$, con $op \in \{\wedge, \vee, \bar{}\}$ e $g_i, g_j \in \{0, 1, x_1, \ldots, x_n, z_1, \ldots, z_{k-1}\}$. Esistono pertanto al più $3 \cdot (k+n+1)^2$ istruzioni in posizione k; se $g > n$, tale numero è maggiorato da $3 \cdot (2 \cdot g)^2$ ed esistono al più $g \cdot 3^g \cdot (2g)^{2g}$ circuiti di dimensione g. Posto $g(n) = \frac{1}{2} \cdot \frac{2^n}{n}$, vale che:

$$\log \frac{\Pi_{g(n)}^n}{2^{2^n}} \leq \log \left(\frac{1}{2} \cdot \frac{2^n}{n} \right) + \frac{1}{2} \cdot \frac{2^n}{n} \cdot \log 3 + \frac{2^n}{n} \cdot \log \frac{2^n}{n} - 2^n = -\frac{2^n}{n} \cdot (\log n + O(1)).$$

Pertanto:

$$\lim_{n \to +\infty} \frac{\Pi_{g(n)}^n}{2^{2^n}} \leq \lim_{n \to +\infty} 2^{-\frac{2^n}{n} \cdot (\log n + O(1))} = 0.$$

Quindi, per "quasi tutte" le funzioni f, $C(f) \geq \frac{1}{2} \cdot \frac{2^n}{n}$. ∎

6 Complessità di alcune funzioni aritmetiche notevoli

I precedenti risultati provano che gran parte delle funzioni booleane richiedono "grandi circuiti" per essere calcolate. Una funzione booleana può essere infatti identificata da un vettore a 2^n componenti binarie; funzioni booleane calcolabili con "piccoli" circuiti ammettono una rappresentazione succinta, e non possono essere quindi "troppo" numerose.

D'altro lato, le funzioni che traggono giovamento dalla rappresentazione circuitale sono proprio quelle che ammettono circuiti "piccoli". Esse sono quindi dotate di "struttura" intrinseca. Qual è la dimensione minima e la profondità minima possibile per una funzione $f \in \mathcal{B}_n$? Consideriamo qui funzioni f che dipendono essenzialmente da tutte le variabili, dove si dice che $f(x_1, \ldots, x_n)$ *dipende essenzialmente da* x_k se

$$f(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_n) \neq f(x_1, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_n).$$

Teorema 6.1 *Se $f \in \mathcal{B}_n$ dipende essenzialmente da tutte le variabili, allora*

$$C(f) \geq n - 1 \quad e \quad D(f) \geq \log n.$$

Dimostrazione. In ogni circuito ottimo per dimensione per f, tutte le variabili hanno fan–out maggiore o uguale a 1 (data la dipendenza essenziale) e l'unica porta con fan–out 0 è quella d'uscita; esistono allora almeno $n + C(f) - 1$ archi. D'altro lato, poiché il fan–in di ogni porta è 2, il numero di archi è $2 \cdot C(f)$. Quindi $n + C(f) - 1 \leq 2 \cdot C(f)$, che prova che $C(f) \geq n - 1$.

Dato un circuito ottimo dal punto di vista della profondità, è possibile costruire una formula per f di ugual profondità; essa è rappresentabile da un albero binario di altezza $D(f)$, quindi $\lceil \log L(f) \rceil \leq D(f)$. La tesi segue dal fatto che $L(f) \geq C(f) \geq n - 1$. ∎

Funzioni booleane $f \in \mathcal{B}_n$ ammettono *circuiti efficienti* (in dimensione e profondità, a meno di costanti moltiplicative) se possono essere calcolte da circuiti che contemporaneamente hanno dimensione $O(n)$ e profondità $O(\log n)$. Vogliamo qui studiare il problema della realizzabilità con circuiti efficienti di funzioni di interesse pratico quali la *somma*, il *prodotto* e l'*accesso a memoria*. Ricordiamo che, in binario, gli interi sono rappresentabili con parole binarie, e che la parola $x_{n-1}x_{n-2}\ldots x_1x_0 \in \{0,1\}^n$ rappresenta l'intero

$$|x_{n-1}x_{n-2}\ldots x_1x_0| = \sum_{k=0}^{n-1} x_k \cdot 2^k.$$

6.1 Addizione

La funzione *addizione* $add_n \in \mathcal{B}_{2n,n+1}$ è definita da:

$$\begin{aligned} &add_n(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}) \\ &\quad = z_0, \ldots, z_n \text{ sse } |x_{n-1}\ldots x_0| + |y_{n-1}\ldots y_0| = |z_n \ldots z_0|. \end{aligned}$$

Porremo in particolare

$$z_n = bps_n(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1})$$

il bit più significativo dell'addizione: un circuito per add_n può essere facilmente ottenuto dai circuiti per $bps_1, bps_2, \ldots, bps_n$. Un semplice algoritmo per il calcolo di bps_n (e quindi di add_n) è quello imparato alle scuole elementari: si basa sul calcolo successivo dei resti $r_0, r_1, \ldots, r_n$, dove r_n è proprio il bit più significativo.

1. $r_{-1} = 0$,
2. **for** $k = 0, \ldots, n$ **do** $r_k = \begin{cases} 1 & \text{se } [r_{k-1}, x_k, y_k] \text{ contiene almeno due 1} \\ 0 & \text{se } [r_{k-1}, x_k, y_k] \text{ contiene 1 al più una volta,} \end{cases}$
3. $z_n = r_n$.

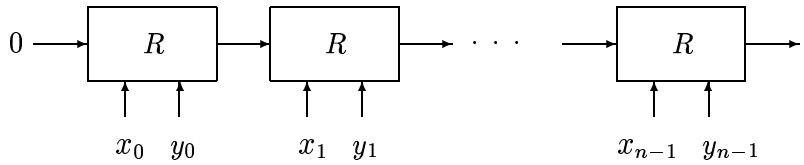
Per costruire il corrispondente circuito, basta considerare la funzione booleana in $\mathcal{B}_3$:

$$r(x,y,z) = \begin{cases} 1 & \text{se } x+y+z \geq 2 \\ 0 & \text{altrimenti.} \end{cases}$$

Essa può essere ottenuta dal circuito R, dove:

$$R(x,y,z) = (x \wedge y) \vee ((x \oplus y) \wedge z).$$

Vale che $C(R) = 4$, $D(R) = 3$; mediante tale circuito è immediato costruire il seguente circuito per bps_n:



Teorema 6.2 *Il metodo imparato alle scuole elementari porta ad un circuito per bps_n di dimensione $5 \cdot n$ e profondità $3 \cdot n - 1$.*

Questo circuito è ottimale dal punto di vista della dimensione, ma la profondità appare elevata. È possibile costruire un circuito contemporaneamente di dimensione $O(n)$ e profondità $O(\log n)$? Si può ottenere una semplice soluzione basata sulle seguenti osservazioni:

1. Fissate le variabili x, y, la funzione $r(x,y,z)$ diventa una funzione $r_{xy}(z)$ in una variabile z. In particolare
$$r_{00}(z) = 0, \quad r_{01}(z) = r_{10}(z) = z, \quad r_{11}(z) = 1.$$
2. Il bit più significativo può essere ottenuto applicando a 0 successivamente le funzioni $r_{x_k y_k}$ $(0 \leq k \leq n-1)$:
$$z_n = r_{x_{n-1}y_{n-1}}(\cdots r_{x_1 y_1}(r_{x_0 y_0}(0))\cdots).$$

3. Ricordando la nozione di composizione di funzioni $(f \circ g)(z) = f(g(z))$, si ottiene

$$z_n = (r_{x_{n-1}y_{n-1}} \circ r_{x_{n-1}y_{n-2}} \circ \ldots \circ r_{x_1y_1} \circ r_{x_0y_0})(0).$$

Il calcolo può essere effettuato allora calcolando *prima* la funzione f ottenuta per composizione delle $r_{x_ky_k}$ $(0 \leq k \leq n-1)$, valutando *poi* f su 0.

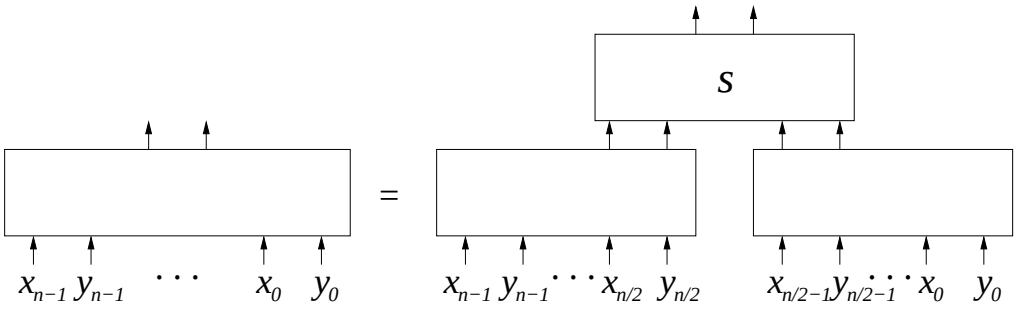
La composizione di funzioni è un'operazione associativa; nel nostro caso può essere descritta dalla seguente tabella (si ricordi che $r_{10} = r_{01}$):

$\circ$	r_{00}	r_{01}	r_{11}
r_{00}	r_{00}	r_{00}	r_{00}
r_{01}	r_{00}	r_{01}	r_{11}
r_{11}	r_{11}	r_{11}	r_{11}

Ponendo $r_{uv} = r_{x_2y_2} \circ r_{x_1y_1}$, la composizione è una funzione booleana in $\mathcal{B}_{4,2}$:

$$u = [(x_1 \oplus y_1) \wedge x_2] \vee (x_1 \wedge y_1)$$
$$v = [(x_1 \oplus y_1) \wedge y_2] \vee (x_1 \wedge y_1).$$

Vale che $C(S) = 6$ e $D(S) = 3$. Possiamo allora ottenere il circuito per la composizione di n funzioni da due circuiti che calcolano la composizione di $\frac{n}{2}$ funzioni come segue:



Detta D_n la profondità del circuito per calcolare la composizione di n funzioni e detta C_n la sua dimensione, vale:

$$C_n = 2 \cdot C_{\frac{n}{2}} + 6 \quad \text{e} \quad D_n = 3 + D_{\frac{n}{2}}.$$

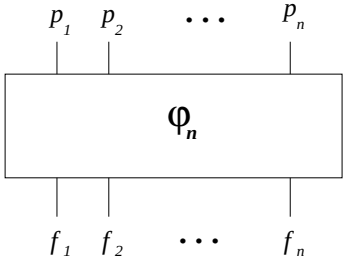
La soluzione delle due precdenti equazioni dà $C_n = O(n)$ e $D_n = O(\log n)$.

Teorema 6.3 *Esistono circuiti ϕ_n per bps_n tali che:*

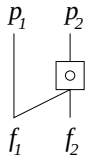
$$C(\phi_n) = O(n) \quad e \quad D(\phi_n) = O(\log n).$$

Un circuito per add_n è facilmente ottenibile dai i circuiti $\phi_1, \phi_2, \ldots, \phi_n$: la sua profondità è $O(\log n)$ ma la dimensione è $O(n^2)$. Per avere un circuito per add_n contemporaneamente di profondità $O(\log n)$ e dimensione $O(n)$ dobbiamo faticare un po' di più.

A tal riguardo, basta costruire un circuito φ_n che avendo in ingresso $f_1, \ldots, f_n$, dia in uscita $p_1, \ldots, p_n$ con $p_k = f_1 \circ f_2 \circ \ldots \circ f_k$ $(1 \leq k \leq n)$, essendo $\circ$ un'operazione associativa (nel nostro caso, le f sono le funzioni riporto r, mentre le p sono le fuzioni bps):

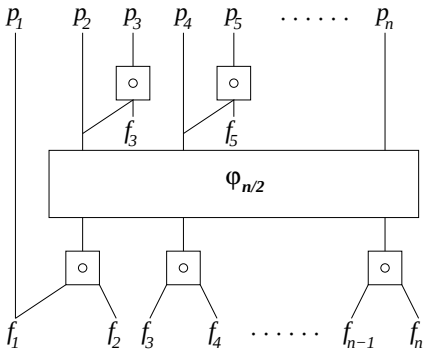


Supponiamo per semplicità che n sia pari. Se $n = 2$ il circuito φ_2 è dato da:



Se $n > 2$ il circuito può essere ottenuto ricorsivamente come segue:

1. componi in parallelo $f_1 \circ f_2, f_3 \circ f_4, \ldots, f_{n-1} \circ f_n$,
2. fornisci i risultati in ingresso al circuito $\varphi_{\frac{n}{2}}$ ottenendo in uscita $p_2, p_4, \ldots, p_n$,
3. calcola in parallelo $p_1 = f_1, p_3 = p_2 \circ f_3, \ldots, p_{n-1} = p_{n-2} \circ f_{n-1}$.



Vale evidentemente

$$C(\varphi_n) = n \cdot C(0) + C(\varphi_{\frac{n}{2}}) \quad \text{e} \quad D(\varphi_n) = 2 \cdot D(0) + D(\varphi_{\frac{n}{2}}).$$

Risolvendo tali equazioni, poiché $C(0)$ e $D(0)$ sono costanti:

$$C(\varphi_n) = O(n) \quad \text{e} \quad D(\varphi_n) = O(\log n).$$

Possiamo pertanto concludere:

Teorema 6.4 *Esistono circuiti φ_n per add_n con*

$$C(\varphi_n) = O(n) \quad e \quad D(\varphi_n) = O(\log n).$$

Il miglior addizionatore (per alti valori di n) è stato proposto da Krapchenko: la sua dimensione è asintotica a $9 \cdot n$ e la sua profondità a $\log n$. È noto che ogni circuito addizionatore deve avere almeno $5 \cdot n - 3$ porte e profondità $\log n$.

6.2 Moltiplicazione

La *moltiplicazione* $mult_n \in \mathcal{B}_{2n,2n}$ è definita da:

$$\begin{aligned} &mult_n(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}) \\ &\quad = z_0, \ldots, z_{2n-1} \text{ sse } |x_{n-1} \ldots x_0| \cdot |y_{n-1} \ldots y_0| = |z_{2n-1} \ldots z_0|. \end{aligned}$$

Il metodo elementare di moltiplicazione riduce il calcolo del prodotto ad una somma iterata come segue:

$$x \cdot y = \left(\sum_{k=0}^{n-1} x_k \cdot 2^k \right) \cdot y = \sum_{k=0}^{n-1} x_k \cdot 2^k \cdot y.$$

Posto $y^{(k)} = 2^k \cdot y$, $y^{(k)}$ è ottenuto facilmente in rappresentazione binaria giustapponendo k zeri alla rappresentazione binaria di y. Quindi:

$$x \cdot y = \sum_{\{0 \le k \le n-1 | x_k = 1\}} y^{(k)}.$$

La costruzione di un circuito per la somma iterata può essere agevolata utilizzando porte che, avendo in ingresso tre numeri a, b, c di n bit, danno in uscita due numeri u, v di $n+1$ bit tali che $|u| + |v| = |a| + |b| + |c|$. Queste porte sono chiamate CSA (Carry Save Adder) e possono essere implementate con efficienti circuiti grazie ad un ingegnoso trucco proposto da Ofman (1963).

Teorema 6.5 CSA *può essere realizzata da un circuito di dimensione* $5 \cdot n$ *e profondità* 3.

Dimostrazione. La somma di tre bit dà al massimo 3, ed i numeri minori o uguali a tre sono rappresentati da due bit r_i e s_i:

$$a_i + b_i + c_i = s_i + 2 \cdot r_i.$$

Questa funzione booleana viene implementata da un Full Adder (FA):

$$s_i = a_i \oplus b_i \oplus c_i, \quad r_i = (a_i \wedge b_i) \vee ((a_i \oplus b_i) \wedge c_i).$$

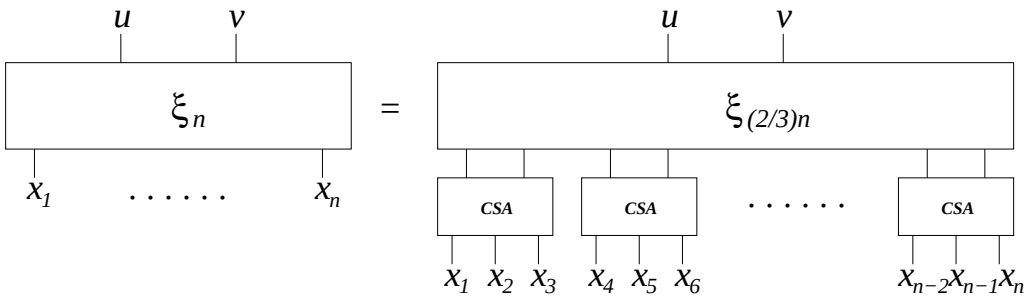
Si verifica facilmente che $D(\text{FA}) = 3$ e $C(\text{FA}) = 5$. Dati allora $a = a_{n-1} \ldots a_0$, $b = b_{n-1} \ldots b_0$, $c = c_{n-1} \ldots c_0$, si ha:

$$\begin{aligned} |a| + |b| + |c| &= \sum_{k=0}^{n-1} (a_k + b_k + c_k) \cdot 2^k \\ &= \sum_{k=0}^{n-1} (s_k + 2 \cdot r_k) \cdot 2^k = \sum_{k=0}^{n-1} s_k \cdot 2^k + \sum_{k=0}^{n-1} r_k \cdot 2^{k+1} = |u| + |v|, \end{aligned}$$

dove $u = s_0 s_1, \ldots, s_{n-1} 0$ mentre $v = 0 r_0 r_1, \ldots, r_{n-1}$. Otteniamo dunque un circuito per CSA con $5 \cdot n$ porte e profondità 3. ∎

Un circuito ξ_n che, avendo in ingresso n numeri $x_1, \ldots, x_n$ di n bit l'uno, dà in uscita due numeri u e v di $2 \cdot n$ bit l'uno tali che $|x_1| + \ldots + |x_n| = |u| + |v|$ può essere allora ottenuto ricorsivamente come segue:

1. Calcola $u_1, u_2, \ldots, u_{\frac{2}{3}\cdot n}$ tali che
   $x_1 + x_2 + x_3 = u_1 + u_2$,
   $x_4 + x_5 + x_6 = u_3 + u_4$,
   $\vdots$
2. Fornisci $u_1, u_2, \ldots, u_{\frac{2}{3}\cdot n}$ in ingresso a $\xi_{\frac{2}{3}\cdot n}$.



Vale che:

$$C(\xi_n) \leq C(\xi_{\frac{2}{3}\cdot n}) + \frac{5}{3} \cdot n \cdot (2 \cdot n) \quad \text{e} \quad D(\xi_n) = D(\xi_{\frac{2}{3}\cdot n}) + 3.$$

Risolvendo tali relazioni di ricorrenza, si ha:

$$C(\xi_n) = O(n^2) \quad \text{e} \quad D(\xi_n) = O(\log n).$$

Sommando infine le uscite u e v con un circuito di profondità $O(\log n)$, e dimensione $O(n)$ otteniamo:

Teorema 6.6 *$mult_n$ è calcolata da circuiti di dimensione $O(n^2)$ e profondità $O(\log n)$.*

Questo circuito è asintoticamente ottimo (a meno di una costante moltiplicativa) per quanto riguarda la profondità. Esistono circuiti di dimensione $o(n^2)$? Una risposta è stata data da Shönage e Strassen, utilizzando una tecnica completamente diversa basata sul calcolo della trasformata di Fourier discreta:

Teorema 6.7 *$mult_n$ è calcolata da circuiti di dimensione $O(n \cdot \log n \cdot \log\log n)$ e profondità $O(\log n)$.*

L'ottimalità del precedente risultato, che diamo senza dimostrazione, non è stata dimostrata. È ancora aperto il problema se $mult_n$ possa essere calcolata da circuiti di dimensione $O(n)$ e profondità $O(\log n)$.

6.3 Accesso diretto a memoria

La funzione di *accesso diretto a memoria* $\mathrm{AD}_n \in \mathcal{B}_{n+k}$, dove $n = 2^k$, è definita come:

$$\mathrm{AD}_n(a_{k-1}, \ldots, a_0, x_0, \ldots, x_{n-1}) = x_{|a_{k-1} \ldots a_0|}.$$

Essa, cioè, permette di ottenere il bit memorizzato all'indirizzo $a_{k-1}, \ldots, a_0$. Osserviamo che se $a_{k-1} = 0$, allora la ricerca deve avvenire nella prima metà della memoria $(x_0, \ldots, x_{\frac{n}{2}-1})$; se $a_{k-1} = 1$, allora la ricerca avverrà nella seconda metà $(x_{\frac{n}{2}}, \ldots, x_{n-1})$. Ciò consente di dare una descrizione ricorsiva di un circuito $\widehat{\mathrm{AD}}_n$ per AD_n:

$$\begin{aligned}&\widehat{\mathrm{AD}}_n(a_{k-1}, \ldots, a_0, x_0, \ldots, x_{n-1}) \\ &\quad = (a_{k-1} \wedge \widehat{\mathrm{AD}}_{\frac{n}{2}}(a_{k-2}, \ldots, a_0, x_{\frac{n}{2}}, \ldots, x_{n-1})) \vee (\overline{a_{k-1}} \wedge \widehat{\mathrm{AD}}_{\frac{n}{2}}(a_{k-2}, \ldots, a_0, x_0, \ldots, x_{\frac{n}{2}-1})).\end{aligned}$$

Vale allora:

$$C(\widehat{\mathrm{AD}}_n) = 2 \cdot C(\widehat{\mathrm{AD}}_{\frac{n}{2}}) + 3 \quad \text{e} \quad D(\widehat{\mathrm{AD}}_n) = D(\widehat{\mathrm{AD}}_{\frac{n}{2}}) + 2.$$

Posto $\widehat{\mathrm{AD}}_1(x_0) = x_0$, segue che $C(\widehat{\mathrm{AD}}_1) = D(\widehat{\mathrm{AD}}_1) = 0$. Risolvendo le precedenti equazioni, otteniamo

$$C(\widehat{\mathrm{AD}}_n) = 3 \cdot n - 3 \quad \text{e} \quad D(\widehat{\mathrm{AD}}_n) = 2 \cdot \log n.$$

Pertanto:

Teorema 6.8 AD_n *è calcolata da un circuito di dimensione* $3 \cdot n - 3$ *e profondità* $2 \cdot \log n$.

Con una diversa tecnica, esprimendo AD_n in forma disgiunta ed applicando risultati sviluppati per le funzioni simmetriche, si ottiene un risultato asintoticamente ottimo:

Teorema 6.9 AD_n *può essere calcolata da un circuito di dimensione* $2 \cdot n + O(n^{\frac{1}{2}})$ *e profondità* $\log n + O(\log \log n)$.

Infatti si dimostra che ogni circuito per AD_n deve avere dimensione almeno $2 \cdot n - 2$.

7 Profondità di circuito e dimensione di formula

Una formula è un circuito con fan–out 1. Anche se la realizzazione VLSI di circuiti pone dei vincoli al fan–out di circuito, il vincolo non è mai così severo da essere 1. L'interesse dei circuiti con fan–out 1 va cercato altrove: le motivazioni principali che spingono allo studio di circuiti con fan–out 1 sono due:

1. Circuiti a fan–out 1 sono formule nell'accezione usuale della logica matematica, come abbiamo già osservato.
2. Esiste una stretta relazione tra la profondità di circuito e la dimensione di formula, come andiamo a dimostrare.

Vale infatti il seguente risultato:

Teorema 7.1 (Spira)

$$\log(L(f)+1) \leq D(f) \leq 5{,}13 \cdot \log(L(f)+1).$$

Dimostrazione. Se esiste un circuito per f di profondità $D(f)$, allora si può costruire una formula F per f di ugual profondità. F può essere vista come un albero binario di altezza $D(f)$; quindi, il numero $L(f)$ di nodi interni è al più $2^{D(f)} - 1$, per cui

$$\log(L(f)+1) \leq D(f).$$

Consideriamo ora il seguente algoritmo ricorsivo che trasforma una formula F in un circuito $\mathcal{C}(f)$ tale che:

1. F e $\mathcal{C}(f)$ calcolano la stessa funzione,
2. $D(f) \leq D(\mathcal{C}(f)) \leq 5{,}13 \cdot \log(L(f)+1)$.

Ricordiamo che la funzione *selettore* $sel(x,y,z) = (\overline{x} \wedge z) \vee (x \wedge y)$ è tale che

$$sel(x,y,z) = \begin{cases} y & \text{se } x = 1 \\ \\ z & \text{altrimenti.} \end{cases}$$

procedure $\mathcal{C}(F)$
if $L(F) = 1$ **then**
  return(F)
else
  begin
    determina op, F_1, F_2 tale che $F = op(F_1, F_2)$ e $L(F_1) \leq L(F_2)$;
    scegli un sottoalbero F_3 di F_2 tale che $\frac{1}{3} \cdot L(F_2) \leq L(F_3) \leq \frac{2}{3} \cdot L(F_2)$;
    $F_{2,0}$:= formula ottenuta da F sostituendo con 0 il sottoalbero F_3;
    $F_{2,1}$:= formula ottenuta da F sostituendo con 1 il sottoalbero F_3;
    return$(op(\mathcal{C}(F_1), sel(\mathcal{C}(F_3), \mathcal{C}(F_{2,0}), \mathcal{C}(F_{2,1}))))$
  end

La correttezza dell'algoritmo è dimostrata facilmente per induzione sulla dimensione della formula F; si osservi che $sel(F_3, F_{2,0}, F_{2,1})$ calcola la stessa funzione calcolata da F_2. Poniamo ora $D(n) = \max\{D(\mathcal{C}(F)) \mid L(f) = n\}$, la massima profondità di circuiti ottenuti da formule di dimensione n. Osserviamo che se la dimensione di F è n, allora la dimensione di F_3 è al più $\frac{2}{3} \cdot n$ e quella di $F_{2,0}$ e $F_{2,1}$ è al più $n - \frac{1}{3} \cdot n$, cioè $\frac{2}{3} \cdot n$. Inoltre, la dimensione di F_1 è al più $\frac{n}{2}$. Si ottiene allora:

$$D(1) = 1, \quad D(n) \leq 1 + \max\left\{ D\left(\frac{n}{2}\right), D(sel) + D\left(\frac{2}{3} \cdot n\right) \right\}.$$

Poiché $D\left(\frac{n}{2}\right) \leq D\left(\frac{2}{3} \cdot n\right)$, segue che:

$$D(1) = 1, \quad D(n) \leq 1 + D(sel) + D\left(\frac{2}{3} \cdot n\right).$$

La precedente relazione di ricorrenza ha soluzione $D(n) \leq (1 + D(sel)) \cdot k + 1$, con $n \cdot \left(\frac{2}{3}\right)^k = 1$. Ricordando che $D(sel) = 2$, segue infine:

$$D(n) \leq \frac{3}{\log \frac{3}{2}} \cdot \log n \approx 5{,}13 \cdot \log n.$$

∎

8 Limite inferiore alla dimensione di formula: teorema di Krapchenko

Per stimare un limite superiore alla dimensione $C(f)$ di una funzione booleana f basta esibire un circuito ϕ che calcola f e stimarne il numero di porte. Provare che un numero α è invece un limite inferiore per $C(f)$, cioè che $C(f) \geq \alpha$, è un compito in linea di principio molto più difficile: bisogna dimostrare che un *qualsiasi* circuito ξ per f è tale che $C(\xi) \geq \alpha$. Non stupisce quindi che risultati generali su limiti inferiori siano molto pochi. Vogliamo qui esibire una tecnica che può fornire limiti non lineari alla dimensione di formula.

Data $f : \{0,1\}^n \to \{0,1\}$, sia con $L(f)$ il minimo numero di $\wedge, \vee, \bar{}$ necessari a realizzare una formula F che calcola f, e con $L^*(f)$ il minimo numero di porte $\wedge, \vee$ necessarie a realizzare una formula F per f. Mostriamo che $L(f)$ e $L^*(f)$ sono abbastanza vicini:

Fatto 8.1

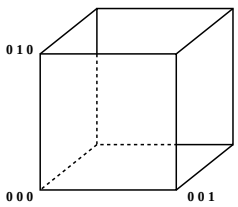
$$L^*(f) \leq L(f) \leq 2 \cdot L^*(f) + 1.$$

Dimostrazione. Evidentemente $L^*(f) \leq L(f)$. Sia F la formula minima per $L^*(f)$, cioè, $L^*(f) = L^*(F)$. Applicando ripetutamente le leggi di De Morgan, possiamo costruire una nuova formula $\tilde{F}$ che contiene lo stesso numero di $\wedge, \vee$ di F e le eventuali negazioni $\bar{}$ nelle variabili. Poiché in un albero binario le foglie superano di uno i nodi interni, il numero di negazioni in $\tilde{F}$ è al più $L^*(\tilde{F}) + 1$. Allora

$$L^*(f) \leq L(f) \leq L^*(\tilde{F}) + L^*(\tilde{F}) + 1 = 2 \cdot L^*(f) + 1.$$

∎

Vogliamo ottenere una stima inferiore di $L^*(f)$. A tal riguardo, occorre considerare un'interpretazione geometrica di $f : \{0,1\}^n \to \{0,1\}$. Il dominio $\{0,1\}^n$ di f è pensabile come un grafo non orientato con vertici $\{0,1\}^n$ e lati che connettono i vertici $\mathbf{x}, \mathbf{y}$ qualora $\mathbf{x}, \mathbf{y}$ siano vettori che differiscono in una sola componente. Tali grafi sono detti *n–cubi*. La figura rappresenta un 3–cubo:



Dati ora $A, B \subseteq \{0,1\}^n$ due sottoinsiemi *disgiunti* di $\{0,1\}^n$, sia $H(A,B)$ il numero di lati del n–cubo che hanno un estremo in A e l'altro in B, mentre al solito $|A|, |B|$ denotano il numero di elementi di A e di B. Ad ogni funzione booleana $f : \{0,1\}^n \to \{0,1\}$, associamo:

Definizione 8.1

$$K(f) = \max\left\{ \frac{H^2(A,B)}{|A| \cdot |B|} \mid A \subseteq f^{-1}(1) \wedge B \subseteq f^{-1}(0) \right\}.$$

Vale, intanto, che:

Fatto 8.2

$$K(\overline{f}) = K(f), \quad K(f \vee g) \leq K(f) + K(g), \quad K(f \wedge g) \leq K(f) + K(g).$$

Dimostrazione.

- $K(\overline{f}) = K(f)$ segue immediatamente dal fatto che $\overline{f}^{-1}(1) = f^{-1}(0)$, $\overline{f}^{-1}(0) = f^{-1}(1)$ ed inoltre $H(A,B) = H(B,A)$.

- Siano $A, B \subseteq \{0,1\}^n$ tali che $K(f \vee g) = \frac{H^2(A,B)}{|A| \cdot |B|}$. Poiché $(f \vee g)^{-1}(0) \subseteq f^{-1}(0)$ e $(f \vee g)^{-1}(0) \subseteq g^{-1}(0)$, essendo $B \subseteq (f \vee g)^{-1}(0)$ segue che $B \subseteq f^{-1}(0)$ e $B \subseteq g^{-1}(0)$. Scegliamo una partizione di A in due sottoinsiemi A_1, A_2 (tali, cioè, che $A_1 \cap A_2 = \emptyset$ e $A_1 \cup A_2 = A$), con $A_1 \subseteq f^{-1}(1)$ e $A_2 \subseteq g^{-1}(1)$. Vale che:

$$\begin{aligned} K(f \vee g) &= \frac{H^2(A_1 \cup A_2, B)}{|A_1 \cup A_2| \cdot |B|} \\ &= \frac{(H(A_1,B) + H(A_2,B))^2}{|A_1| \cdot |B| + |A_2| \cdot |B|} \leq \frac{H^2(A_1,B)}{|A_1| \cdot |B|} + \frac{H^2(A_2,B)}{|A_2| \cdot |B|} \leq K(f) + K(g). \end{aligned}$$

  Il terzo punto di tale catena di equazioni e disequazioni dipende dal fatto che, per $a, b, c, d \geq 0$, vale la seguente disuguaglianza:

$$\frac{(a+b)^2}{c+d} \leq \frac{a^2}{c} + \frac{b^2}{d}.$$

  Questa disuguaglianza può essere verificata direttamente con facili conti:

$$\frac{(a+b)^2}{c+d} \leq \frac{a^2}{c} + \frac{b^2}{d} \;\Leftrightarrow\; (a+b)^2 \cdot cd \leq a^2 d \cdot (c+d) + b^2 c \cdot (c+d) \;\Leftrightarrow\; 0 \leq (ad - cb)^2.$$

- Osserviamo che

$$K(f \wedge g) = K(\overline{\overline{f} \vee \overline{g}}) = K(\overline{f} \vee \overline{g}) \leq K(\overline{f}) + K(\overline{g}) = K(f) + K(g).$$

∎

Possiamo ora enunciare un limite inferiore di $L^*(f)$ in termini di $K(f)$:

Teorema 8.1 (Krapchenko)

$$L^*(f) \geq K(f) - 1.$$

Dimostrazione. Per induzione. Per le variabili x_k per cui $L^*(x_k) = 0$, vale che $K(x_k) = 1$ come si verifica direttamente (si ponga $A = \{(0 \ldots 1 \ldots 0)\}$, $B = \{(0 \ldots 0)\}$). Quindi $L^*(x_k) = K(x_k) - 1$.

Sia $L^*(f) = n + 1$; diciamo F la formula ottima che calcola f e sia $F = F_1 \vee F_2$ (il caso in cui $F = F_1 \wedge F_2$ è simmetrico). Osserviamo che F_1 e F_2 sono formule ottime che calcolano le funzioni booleane f_1 e f_2. Poiché inoltre $L^*(F) = L^*(F_1) + L^*(F_2) + 1$, risulta che $L^*(f_1) = L^*(F_1) \leq n$ e $L^*(f_2) = L^*(F_2) \leq n$, e quindi per ipotesi di induzione $L^*(f_1) \geq K(f_1) - 1$ e $L^*(f_2) \geq K(f_2) - 1$. Allora:

$$\begin{aligned} K(f) - 1 = K(f_1 \vee f_2) - 1 &\leq K(f_1) + K(f_2) - 1 \\ &\leq L^*(f_1) + 1 + L^*(f_2) + 1 - 1 = L^*(F_1) + L^*(F_2) + 1 = L^*(F) = L^*(f). \end{aligned}$$

∎

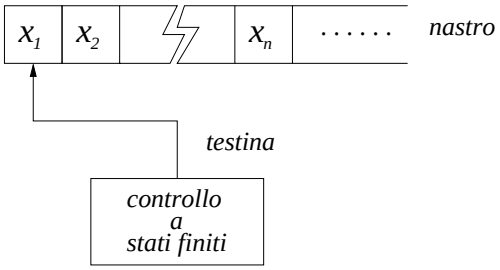
9 Macchine di Turing deterministiche

Abbiamo visto le famiglie di circuiti booleani come dispositivi di calcolo parallelo. Introduciamo ora un modello matematico per la nozione di computazione sequenziale: le macchine di Turing.

Una *macchina di Turing* (nel seguito abbreviata con MdT) *deterministica* ad un nastro è una 6–tupla $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \{q_s, q_n\}\rangle$ in cui:

- Q è un insieme *finito* di *stati*,
- Σ è l'*alfabeto di input*,
- Γ è l'*alfabeto di lavoro*. Vale che $\Sigma \subset \Gamma$; esiste, in particolare, un simbolo speciale *blank* $B \in \Gamma - \Sigma$.
- $\delta : (Q - \{q_s, q_n\}) \times \Gamma \to Q \times (\Gamma - \{blank\}) \times \{-1, 1\}$ è la *funzione di transizione*,
- $q_0 \in Q$ è lo *stato iniziale*,
- $\{q_s, q_n\} \subseteq Q$ è l'insieme degli *stati finali*. In particolare, q_s è detto *accettante*, mentre q_n è detto *non accettante*

La nostra MdT M è schematizzata come segue:



Il *nastro* consta di una sequenza infinita a destra di *celle*, ognuna delle quali è in grado di memorizzare un simbolo di Γ. Una *testina* di lettura/scrittura può leggere e scrivere singoli caratteri da/sul nastro. Lo stato del *controllo* è rappresentato, istante per istante, da uno degli elementi dell'insieme Q. Informalmente, M viene inizializzata memorizzando la stringa di input $x = x_1 \dots x_n \in \Sigma^*$ carattere per carattere nelle prime n celle del nastro, mentre il resto delle celle contiene il simbolo blank B; il controllo a stati finiti si trova nello stato iniziale q_0; la testina viene posizionata sul primo carattere di x. Se M:

1. si trova nello stato $q \notin \{q_s, q_n\}$,
2. la testina sta scandendo il simbolo σ,
3. $\delta(q, \sigma) = (q', \sigma', d)$, con $d \in \{-1, 1\}$,

allora M, in una *mossa (o passo)*:

1. assume lo stato q',
2. sovrascrive σ' a σ,
3. sposta la testina di una posizione a destra ($d = 1$) o a sinistra ($d = -1$). (Si assume che δ venga definita in modo da evitare spostamenti a sinistra della testina qualora la stessa si trovi alla prima cella del nastro.)

Una sequenza così fatta di mosse è detta *computazione di M su input x*. Tale computazione può essere finita o infinita, secondo che si giunga o meno ad uno dei due stati finali q_s, q_n. La computazione di M su x è *accettante* se termina nello stato q_s; in tal caso, diciamo che M *accetta* x. Se invece la computazione termina nello stato q_n oppure non termina, diciamo che M *non accetta* x. Il *linguaggio L_M accettato da M* è definito come l'*insieme delle stringhe in Σ^* accettate da M*.

Volendo formalizzare tali concetti, è utile introdurre la nozione di configurazione. Una *configurazione (istantanea)* di M è una descrizione, ad un dato istante: (1) dello stato assunto dal controllo, (2) dalla posizione della testina sul nastro, (3) dal contenuto del nastro. Più precisamente, diciamo che M si trova nella configurazione $\langle q, k, w \rangle$ se: il controllo è nello stato q, la testina è posizionata sulla k–esima cella del nastroe la porzione non blank del nastro è rappresentata dalla stringa $w \in \Gamma^*$. Una configurazione può anche essere denotata da uqv, con ciò intendendo che M si trova nello stato q, il contenuto non blank del nastro di lavoro è la stringa $uv \in \Gamma^*$, e la testina sta scandendo il primo

carattere di v. La *configurazione iniziale* di M su input x è data da $c_0 = \langle q_0, 1, x\rangle$ o, equivalentemente, $c_0 = q_0 x$.

Data una configurazione c, denotiamo con $q(c)$ lo stato contenuto in c; c è detta *accettante (non accettante)* se $q(c) = q_s$ ($q(c) = q_n$). Date due configurazioni c' e c'', con $q(c') \notin \{q_s, q_n\}$, diciamo che c'' è la *configurazione successiva* a c' (e scriviamo $c' \vdash c''$) se M passa da c' a c'' in una mossa. Le configurazioni accettanti e non accettanti non hanno successori e sono, quindi, da ritenersi *configurazioni d'arresto*. L'aggettivo "deterministico" esprime il fatto che per ogni configurazione di M esiste *al più* una configurazione successiva univocamente determinata dalla funzione di transizione di M.

La computazione di M su x può dunque essere vista come una sequenza (eventualmente infinita) $c_0, c_1, \ldots, c_m, \ldots$ di configurazioni tale che:

1. $c_0 = \langle q_0, 1, x\rangle$,
2. $\forall i \geq 0$ $(c_i \vdash c_{i+1})$.

M *accetta* x se la computazione di M su x termina in una configurazione *accettante*. Il *linguaggio accettato (o riconosciuto)* da M è l'insieme

$$L_M = \{x \in \Sigma^* \mid M \text{ accetta } x\}.$$

Notiamo esplicitamente che, per le stringhe non appartenenti a L_M, la relativa computazione può terminare nello stato non accettante oppure non terminare affatto.

10 Macchine di Turing e problemi di decisione

Una prima importante prerogativa delle macchine di Turing è, dunque, quella di riconoscere insiemi (o equivalentemente linguaggi, assumendo opportune codifiche). Ricordiamo, a tal proposito, il seguente notevole

Teorema 10.1 *La classe degli insiemi ricorsivamente numerabili coincide con la classe degli insiemi riconosciuti da macchine di Turing deterministiche.*

In altre parole, una MdT deterministica può essere vista come una *procedura* che accetta le stringhe appartenenti ad un insieme ricorsivamente numerabile. Come osservato in precedenza, sulle stringhe non appartenenti all'insieme, tale procedura può anche non terminare. Nella soluzione pratica di problemi, siamo invece interessati a procedure che terminano su ogni ingresso, cioè, ad *algoritmi*.

Vediamo come il riconoscimento di linguaggi da parte di MdT deterministiche possa formalizzare il concetto di *algoritmo deterministico per la soluzione di un problema di decisione*.

Un problema di decisione A viene definito fornendo:

- un'ISTANZA *generica*: designa un'insieme di *istanze (particolari)* di A,
- una QUESTIONE: chiede la validità o meno di una certa proprietà definita sulle istanze particolari di A.

Un esempio di problema di decisione è il seguente:

PARITÀ

ISTANZA: Un intero positivo n.

QUESTIONE: n è pari?

Indichiamo con A_s l'insieme delle istanze del problema A a risposta positiva. (Nel nostro esempio, $\mathbf{PARITÀ}_s = \{n \in \mathbf{N} \mid n = 2 \cdot k \text{ per qualche } k \in \mathbf{N}\}$.) Un *algoritmo deterministico per la soluzione di A* è una procedura di calcolo che, avendo in ingresso un'istanza I di A, risponde SÌ se $I \in A_s$, NO altrimenti.

Vediamo ora come sia possibile associare un linguaggio ad A. Supponiamo che le istanze di A vengano codificate con parole su un certo alfabeto Σ. Il *linguaggio* $L[A] \subseteq \Sigma^*$ *associato ad* A è l'insieme delle stringhe che codificano gli elementi in A_s. Diciamo, allora, che la MdT deterministica M risolve il problema di decisione A se *la computazione di* M *si arresta su ogni stringa in input*, e

$$L_M = L[A].$$

M è l'*algoritmo deterministico per la soluzione di* A. Sottolineamo che nella definizione data precedentemente di linguaggio accettato, la terminazione delle computazioni di M era richiesta unicamente sulle stringhe in L_M. Affinché M modelli un algoritmo deterministico per la soluzione di un problema di decisione, richiediamo che le computazioni di M abbiano termine su *tutte* le stringhe in ingresso, fornendo risposte appropriate.

Tornando al problema **PARITÀ** ed assumendo una codifica binaria sugli interi, vale chiaramente che

$$L[\mathbf{PARITÀ}] = \{x \in \{0,1\}^* \mid \text{ il bit meno significativo di } x \text{ è } 0\}.$$

Una MdT deterministica che termina su ogni input ed accetta $L[\mathbf{PARITÀ}]$ — quindi, che risolve **PARITÀ** — non fa altro che spostare la testina sul bit meno significativo della stringa in input, accettando se quest'ultimo è 0 e rifiutando se è 1.

Concludiamo ricordando che:

Teorema 10.2 *La classe degli insiemi ricorsivi coincide con la classe degli insiemi riconosciuti da algoritmi deterministici.*

11 Macchine di Turing per il calcolo di funzioni

Oltre che come riconoscitori di linguaggi, le macchine di Turing deterministiche possono essere viste come dispositivi per il calcolo di funzioni. Diciamo che la MdT deterministica M calcola la funzione parziale $f : \Sigma^* \to \Gamma^*$ se:

1. Su ogni stringa $x \in \Sigma^*$ su cui f è definita, la computazione di M su x si arresta in una configurazione avente $f(x) \in \Gamma^*$ sul nastro di lavoro.
2. Su ogni stringa $x \in \Sigma^*$ su cui f non è definita, la computazione di M su x non si arresta.

È ben noto che le macchine di Turing deterministiche costituiscono un *sistema di programmazione accettabile*. Infatti:

1. La classe delle funzioni calcolate da MdT deterministiche coincide con quella delle funzioni *ricorsive parziali*.
2. Esiste una MdT deterministica *universale*, in grado, cioè, di simulare ogni altra MdT deterministica.
3. Vale il *teorema* S_n^m.

In tale contesto, possiamo enunciare la seguente versione della

TESI DI CHURCH

La classe delle funzioni intuitivamente calcolabili coincide con la classe delle funzioni calcolate da macchine di Turing deterministiche.

12 Tempo di calcolo

La principale risorsa computazionale di cui tener conto nella valutazione delle prestazioni di algoritmi sequenziali è senz'altro il *tempo di calcolo*. Mediante il modello delle MdT, possiamo dare una formalizzazione ben precisa di tale risorsa.

Definizione 12.1 *Data la MdT deterministica* $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \{q_s, q_n\}\rangle$ *ed una stringa* $x \in \Sigma^*$, *sia* $T(x)$ il numero di mosse della computazione di M su x. *Poniamo* $T(x) = +\infty$ *se tale computazione non termina. La* complessità in tempo (caso peggiore) di M *è la funzione* $t : \mathbf{N} \to \mathbf{N}$ *definita come:*

$$t(n) = \max\{T(x) \mid x \in \Sigma^* \wedge |x| = n\}.$$

Diciamo che il linguaggio $L \subseteq \Sigma^*$ *è* riconosciuto in tempo (deterministico) $f(n)$ *se esiste una MdT deterministica* M *tale che:*

- $L = L_M$,
- M *ha complessità in tempo* $t(n) \leq f(n)$.

La complessità in tempo (deterministica) di L *è la minima complessità in tempo possibile per una MdT deterministica che accetta* L.

Possiamo, a questo punto, introdurre la definizione di classe di complessità in tempo intesa come classe di linguaggi accettati da MdT entro il medesimo vincolo di tempo.

Definizione 12.2 *Per ogni funzione* $t : \mathbf{N} \to \mathbf{N}$, DTIME$(t(n))$ *è la classe dei linguaggi accettati da MdT deterministiche in tempo* $t(n)$.

Sottolineamo che l'"ampiezza" di DTIME$(t(n))$ dipende essenzialmente dal tasso di crescita di $t(n)$ e non tanto da eventuali costanti moltiplicative. Ciò in virtù della seguente

Proposizione 12.1 *Per ogni funzione* $t : \mathbf{N} \to \mathbf{N}$ *tale che* $\lim_{n\to+\infty} \frac{t(n)}{n} = +\infty$ *e per ogni costante* $c > 0$*, vale:*

$$\mathrm{DTIME}(c \cdot t(n)) = \mathrm{DTIME}(t(n)).$$

In altre parole, per complessità in tempo più che lineari, possiamo intendere la classe $\mathrm{DTIME}(t(n))$ come la classe dei linguaggi riconosciuti in tempo deterministico $O(t(n))$.

Particolare rilevanza nell'ambito della teoria della complessità strutturale assume la classe P dei linguaggi riconosciuti in tempo deterministico *polinomiale*, cioè:

$$\mathrm{P} = \bigcup_{k\geq 0} \mathrm{DTIME}(n^k) = \mathrm{DTIME}(n^{O(1)}).$$

Ricordando le nozioni al paragrafo 10, possiamo alternativamente considerare P come classe di problemi di decisione. L'importanza di tale classe è data dal fatto che essa è unanimemente considerata la classe dei problemi "praticamente risolubili" su dispositivi sequenziali — con tempi di calcolo, cioè, non proibitivi. È facile convincersi di ciò andando a confrontare tra loro complessità in tempo polinomiali ed esponenziali. Quest'ultime presentano valori spaventosamente alti anche per input di dimensioni alquanto contenute. Tali considerazioni danno corpo alla

> Tesi di Church estesa
>
> Un problema di decisione è effettivamente risolubile se ammette un algoritmo deterministico polinomiale in tempo.

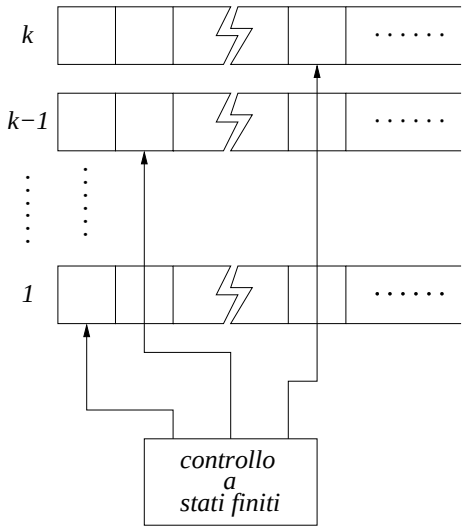
Osserviamo che la portata di questa tesi si estende in modo naturale dai problemi di decisione alle funzioni in genere. È sufficiente, infatti, applicare le nozioni e le considerazioni qui introdotte alle MdT che calcolano funzioni (vedi paragrafo 11).

Un'altra caratteristica di rilievo della classe P — che pure dà consistenza alla tesi di Church estesa — è la sua *robustezza*, ovvero, l'invarianza rispetto alla definizione mediante altri formalismi. È ben noto, ad esempio, che:

- P può essere definita come la classe dei linguaggi accettati da macchine RAM che lavorano in tempo polinomiale (assumendo costo logaritmico).
- Come mostrato al corollario 15.1, P coincide con la classe dei linguaggi che ammettono famiglie di circuiti aventi un numero polinomiale di porte logiche, uniformemente descrivibili in spazio logaritmico.

Vogliamo qui approfondire l'invarianza di P rispetto alla definizione mediante MdT a più nastri.

Una *MdT deterministica* M *a* $k > 1$ *nastri* è schematizzata nella seguente figura:



La definizione formale di M rimane sostanzialmente invariata. L'unico cambiamento saliente si ha nella funzione di transizione la quale assume la forma

$$\delta : (Q - \{q_s, q_n\}) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 1\}^k.$$

Ciò è dovuto al fatto che ora M legge e scrive k–tuple di simboli e deve comandare il moto di k testine. Possiamo assumere che le stringhe in ingresso vengano memorizzate sul primo nastro. Le definizioni date in precedenza di configurazione, computazione, etc., vengono modificate in maniera ovvia; la definizione di complessità in tempo rimane inalterata. Ciò che vogliamo stimare è il costo in tempo della simulazione di una MdT deterministica a $k > 1$ nastri mediante una MdT deterministica ad un singolo nastro:

Teorema 12.1 *Se un linguaggio* L *è riconoscibile in tempo* $t(n)$ *da una MdT deterministica a* $k > 1$ *nastri, allora* L *è riconoscibile da una MdT deterministica a un nastro in tempo* $O(t^2(n))$.

Dimostrazione. Sia M una MdT deterministica a k nastri che riconosce L in tempo $t(n)$ e sia Γ l'alfabeto di lavoro di M. Possiamo definire un nuovo alfabeto di lavoro $\Gamma' = (\Gamma \times \{0, 1\})^k$. Così, in maniera del tutto ovvia, una stringa $x \in \Gamma'^*$ può descrivere il contenuto delle porzioni non blank dei k nastri di M e la posizione delle corrispondenti testine. Infatti, se $x_j = ((a_1, l_1), \ldots, (a_k, l_k))$ è la j–esima lettera di x, allora ogni a_i è il contenuto della j–esima cella nel nastro i–esimo $(1 \leq i \leq k)$, mentre $l_i = 1$ se e solo se su tale cella è posizionata la relativa testina.

Definiamo ora una nuova MdT deterministica M' ad un solo nastro, avente Γ' come alfabeto di lavoro, che simula M. La computazione di M' su un generico input è suddivisa in fasi, una per ogni mossa della macchina M. In ogni fase, M' scorre la porzione non blank del suo nastro dalla prima all'ultima posizione e riporta quindi la testina sulla prima cella. Durante questa doppia passata, la macchina aggiorna opportunamente il contenuto

del nastro in modo da simulare un passo di M: nella prima scansione si aggiornano le componenti corrispondenti agli spostamenti delle testine di M verso destra, mentre nella seconda si aggiornano quelli relativi agli spostamenti verso sinistra. Inoltre, al termine di ogni fase, M' ha memorizzato nello stato il simbolo letto da ciascuna testina di M in modo da determinare la mossa da simulare nella fase successiva (cioè, i nuovi simboli da stampare e gli spostamenti delle testine).

Poiché M in $t(n)$ passi può visitare al più $t(n)+1$ celle su ogni nastro, ogni fase di M' su input di dimensione n può essere eseguita in al più $2 \cdot t(n) + 1$ passi. Ne segue che il numero totale di mosse compiute da M' è al più uguale a $2 \cdot t^2(n) + t(n)$. ∎

È immediato dunque concludere che la classe P non cambia qualora la si definisca mediante MdT deterministiche con $k > 1$ nastri.

13 Spazio di lavoro

Veniamo ora alla formalizzazione dell'altra importante risorsa di calcolo, ovvero lo *spazio* inteso come quantità di memoria occupata durante la computazione.

Definizione 13.1 *Data la MdT deterministica* $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \{q_s, q_n\}\rangle$ *ed una stringa* $x \in \Sigma^*$, *sia* $S(x)$ il numero di celle diverse del nastro visitate durante la computazione di M su x. *Poniamo* $S(x) = +\infty$ *se M visita infinite celle. La* complessità in spazio (caso peggiore) di M *è la funzione* $s : \mathbf{N} \to \mathbf{N}$ *definita come*

$$s(n) = \max\{S(x) \mid x \in \Sigma^* \wedge |x| = n\}.$$

Diciamo che il linguaggio $L \subseteq \Sigma^*$ *è* riconosciuto in spazio (deterministico) $f(n)$ *se esiste una MdT deterministica M tale che:*

- $L = L_M$,
- *M ha complessità in spazio* $s(n) \leq f(n)$.

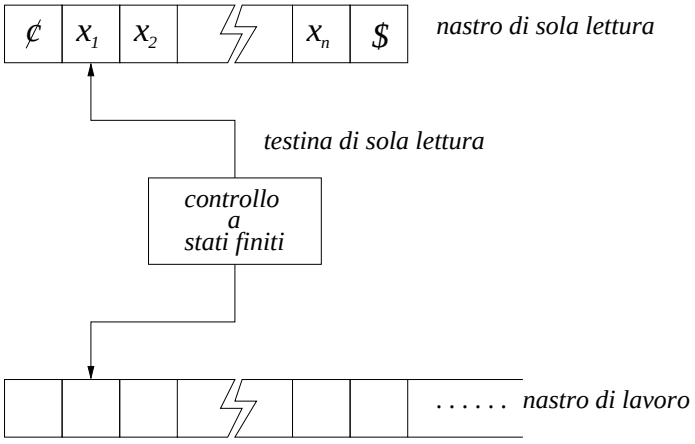
La complessità in spazio (deterministica) di L *è la minima complessità in spazio per una MdT deterministica che accetta L.*

Dalla definizione data appare chiaro che, per ogni MdT, vale $s(n) \geq n$. Dobbiamo infatti considerare quantomeno lo spazio necessario per memorizzare le stringhe in ingresso, spazio che verrà necessariamente scandito dalla testina.

Per poter apprezzare complessità in spazio sublineari, dobbiamo ricorrere ad una variante del modello tradizionale di MdT. Forniamo, cioè, alla MdT M un *nastro di sola lettura (o di input)* su cui memorizzare le stringhe in ingresso tra due caratteri speciali di fine stringa $¢, \$ \notin \Sigma$. Anche in tal caso, la definizione formale di M rimane pressoché invariata. Solamente, la funzione di transizione assume la forma

$$\delta : (Q - \{q_s, q_n\}) \times (\Sigma \cup \{¢, \$\}) \times \Gamma \to Q \times (\Gamma - \{blank\}) \times \{-1, 1\}^2.$$

Ciò esprime il fatto che M ora legge un simbolo sia dal nastro di input che dal nastro di lavoro; quindi, M deve modificare il solo nastro di lavoro e comandare il moto delle due testine. La nuova variante di MdT è schematizzata nella seguente figura:



Anche per questa variante, le nozioni di configurazione, computazione, etc., vanno modificate in maniera ovvia. In particolare, una configurazione per M è una 4–tupla $c = \langle q, k, k', w\rangle$ in cui q è lo stato del controllo, k (k') è la posizione della testina sul nastro di input (lavoro) e w è il contenuto non blank del nastro di lavoro. Equivalentemente, c può essere espressa come $\langle k, uqv\rangle$, con $uv = w$ e $|u| + 1 = k'$.

Veniamo alla definizione di spazio per tale modello, tenendo conto della definizione 13.1. Per ogni stringa $x \in \Sigma^*$, $S(x)$ è ora dato dal numero di celle diverse *del solo nastro di lavoro* visitate da M durante la computazione su x. Dunque, la *complessità in spazio deterministica* $s(n)$ *di* M è da intendersi come il massimo numero di celle diverse del *nastro di lavoro* visitate durante la computazione di M su stringhe di lunghezza n.

Definizione 13.2 *Per ogni funzione* $s : \mathbf{N} \to \mathbf{N}$, $\mathrm{DSPACE}(s(n))$ *è la classe dei linguaggi accettati da MdT deterministiche con nastro di sola lettura in spazio* $s(n)$.

Anche per lo spazio, abbiamo che:

Proposizione 13.1 *Per ogni costante* $c > 0$, *vale:*

$$\mathrm{DSPACE}(c \cdot s(n)) = \mathrm{DSPACE}(s(n)).$$

Possiamo, quindi, considerare $\mathrm{DSPACE}(s(n))$ come la classe dei linguaggi accettati in spazio deterministico $O(s(n))$. Particolarmente interessanti sono le classi

$$\mathrm{L} = \mathrm{DSPACE}(\log \mathrm{n}), \quad \mathrm{PSPACE} = \bigcup_{k \geq 0} \mathrm{DSPACE}(n^k) = \mathrm{DSPACE}(n^{O(1)}).$$

Al paragrafo 16, porremo in relazione la classe L con la classe dei linguaggi efficientemente riconoscibili su dispositivi paralleli.

Analizziamo, molto sinteticamente, alcune relazioni tra classi di complessità in tempo ed in spazio. A tal proposito, assumiamo il modello di MdT con nastro di sola lettura anche per la valutazione del tempo. Tale richiesta risulta essere ragionevole — ad esempio, non modifica la classe P, com'è facilmente dimostrabile dal Teorema 12.1 — e consente un confronto più agevole tra tempo e spazio. Un primo semplice risultato è il seguente:

Fatto 13.1 *Per ogni funzione* $f : \mathbf{N} \to \mathbf{N}$*, vale:*

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)).$$

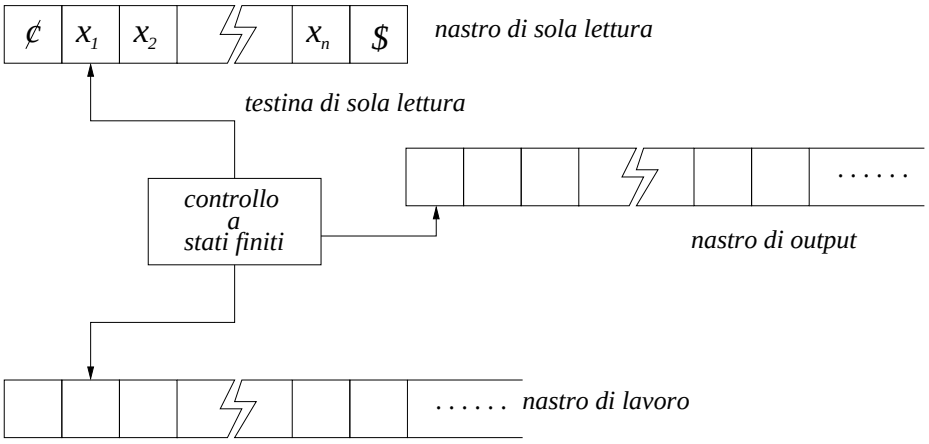
Dimostrazione. È sufficiente osservare che una MdT deterministica che lavora in tempo $f(n)$ può visitare al più $f(n)$ celle diverse sul nastro di lavoro, una nuova cella per ogni mossa. ∎

È quindi immediato ottenere $\mathrm{P} \subseteq \mathrm{PSPACE}$. Come semplice conseguenza dei corollari 15.1 e 15.2, è possibile ottenere $\mathrm{L} \subseteq \mathrm{P}$. In conclusione, dunque, valgono le seguenti relazioni:

$$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE}.$$

Non è ancora noto se tali inclusioni siano proprie o meno.

Concludiamo il paragrafo con alcune considerazioni circa la complessità in spazio per il calcolo di funzioni. Nei paragrafi successivi, saremo interessati a funzioni calcolate da MdT deterministiche in spazio logaritmico. Chiaramente tale complessità non è consentita dal modello ad un solo nastro proposto al paragrafo 11. Per superare tale limitazione, introduciamo il seguente dispositivo M:



In sostanza, abbiamo aggiunto ad una MdT deterministica con nastro di sola lettura un *nastro di output di sola scrittura*. Su tale nastro, la relativa testina può muovere solamente verso destra, scrivendo simboli da un alfabeto Δ. La computazione di M è del tutto analoga a quella di una MdT deterministica usuale; solamente, ad ogni mossa, M scrive un simbolo sul nastro di output. Diciamo che M calcola la funzione $f : \Sigma^* \to \Delta^*$ se la computazione di M su ogni x su cui f è definita termina con $f(x)$ scritta sul nastro di output. Sulle stringhe su cui f non è definita, la computazione di M non ha termine.

La complessità in spazio (ed in tempo) di M è definita esattamente come per il modello con nastro di sola lettura.

14 Linguaggi e circuiti

Una macchina di Turing riconosce un linguaggio anche infinito. Un circuito, invece, calcola funzioni booleane su insiemi finiti — precisamente su $\{0,1\}^n$, n essendo il numero di variabili di input del circuito stesso. Come definire in maniera appropriata il riconoscimento di linguaggi mediante circuiti booleani?

Dato $L \subseteq \{0,1\}^*$, sia L_n il numero di stringhe in L di lunghezza n, cioè, $L_n = \{x \mid x \in L \wedge |x| = n\}$. L è univocamente determinato dalla sequenza $\{L_n\}$ e, per ogni n, L_n è univocamente individuato dalla sua funzione caratteristica $\chi_{L_n} : \{0,1\}^n \to \{0,1\}$, che è una funzione booleana. In ultima analisi, L è univocamente determinato dalla sequenza $\{\chi_{L_n}\}$ di funzioni booleane. L può essere allora riconosciuto da una famiglia $\{\phi_n\}$ di circuiti se, per ogni n, ϕ_n calcola χ_{L_n}:

Definizione 14.1 *La famiglia $\{\phi_n\}$ di circuiti booleani* riconosce il linguaggio L *se, per ogni n, ϕ_n calcola χ_{L_n}. Si dirà anche che L* ammette la famiglia $\{\phi_n\}$ di circuiti.

Il collegamento tra MdT e circuiti booleani può essere ottenuto attraverso la nozione di descrizione uniforme. Una famiglia $\{\phi_n\}$ di circuiti può essere descritta da un programma che avendo in ingresso n stampa il circuito ϕ_n. Possiamo pensare a tale programma come ad una MdT deterministica che calcola la funzione $f : \{1\}^* \to \{0,1\}^*$, dove $f(n)$ rappresenta un'opportuna descrizione di ϕ_n (si veda il paragrafo precedente per il concetto di funzione calcolata entro una certa complessità in tempo o spazio da una MdT deterministica).

Definizione 14.2 *Una* descrizione uniforme *di una famiglia $\{\phi_n\}$ di circuiti è un programma ξ che avendo in input 1^n stampa in uscita il circuito ϕ_n.*

Se il programma ξ richiede spazio $s(n)$ (tempo $t(n)$), diremo che la famiglia $\{\phi_n\}$ è uniformemente descrivibile in spazio $s(n)$ (tempo $t(n)$).

Avendo una descrizione uniforme ξ di una famiglia $\{\phi_n\}$ di circuiti per $L \subseteq \{0,1\}^*$, è possibile decidere in modo automatico se una parola appartiene a L. È sufficiente considerare il seguente schema:

1. INGRESSO: $w \in \{0,1\}^*$,
2. calcola $n = |w|$,
3. calcola $\xi(n) = \phi_n$,
4. calcola $\phi_n(w)$.

15 Circuiti e macchine

Se un linguaggio $L \subseteq \{0,1\}^*$ è riconosciuto da una MdT deterministica entro certi vincoli di tempo o spazio, che tipo di famiglia di circuiti ammette?

Teorema 15.1 *Se L è riconosciuto in tempo $t(n)$ da una MdT a un nastro, allora ammette una famiglia di circuiti $\{\phi_n\}$ descrivibili uniformemente in spazio $O(\log t(n))$ e dimensione $C(\phi_n) = O(t^2(n))$.*

Dimostrazione. Data la macchina di Turing $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \{q_s, q_n\}\rangle$ che riconosce L in tempo $t(n)$, ogni configurazione $\langle q, k, x_1 \dots x_N\rangle$ della macchina può essere univocamente descritta dalla parola $x_1 \dots x_{k-1} q x_k \dots x_N$. Supponiamo inoltre che i simboli in $Q \cup \Gamma$ siano codificati da parole binarie di lunghezza $\lceil \log(|Q| + |\Gamma|)\rceil = A$. Se $\alpha x_s x_{s+1} x_{s+2} x_{s+3} x_{s+4} \beta$ è una configurazione e $\alpha' y_s y_{s+1} y_{s+2} y_{s+3} y_{s+4} \beta'$ è la configurazione successiva, il simbolo y_{s+2} è univocamente determinato dai simboli $x_s, x_{s+1}, x_{s+2}, x_{s+3}, x_{s+4}$. Esiste allora una funzione $f : \{0,1\}^{5\cdot A} \to \{0,1\}^A$ dove:

$$f(x_1x_2x_3x_4x_5) = \begin{cases} y & \text{se } \alpha_1\alpha_2 y \alpha_4\alpha_5 \text{ è la configurazione successiva di } x_1 \dots x_5 \\ & \text{e } \alpha_1 \dots \alpha_5 \text{ contiene al più un simbolo in } Q \\ x_3 & \text{altrimenti.} \end{cases}$$

Data ora la parola w con $|w| = n$, si consideri la sequenza di configurazioni, ognuna di lunghezza $2 \cdot t(n) + n + 3$, data da $\langle c_1, c_2, \dots, c_{t(n)}\rangle$ dove:

1. $c_1 = B^{t(n)+2} q_0 w B^{t(n)+2} = x_1 x_2 \dots x_N$,
2. c_{k+1} è la configurazione successiva a c_k $(1 \le k < t(n))$.

Osserva che $w \in L$ se e solo se $c_{t(n)}$ contiene lo stato q_s. Possiamo allora costruire il seguente circuito ϕ_n con variabili $z_{i,j}$ $(1 \le i \le t(n),\ 1 \le j \le 2 \cdot t(n) + n + 3)$ e u (variabile d'uscita):

$$\begin{aligned} & z_{1,1} := x_1; \\ & \vdots \\ & z_{1,N} := x_N; \\ & \vdots \\ & z_{i+1,1} := B; \\ & z_{i+2,2} := B; \\ & \vdots \\ & z_{i+1,j} := f(z_{i,j-2}, \dots, z_{i,j+2}, \dots); \\ & \vdots \\ & u = \bigvee\nolimits_j (z_{t(n),j} = q_s). \end{aligned}$$

Vale che:

1. $\phi_n(w) = 1$ se e solo se $w \in L$.
2. Le variabili $z_{i,j}$ sono descrivibili con $O(\log t(n))$ bit; esiste inoltre un programma che avendo in ingresso 1^n stampa ϕ_n usando spazio $O(\log t(n))$.
3. La dimensione $C(\phi_n)$ è $O(t^2(n))$.

∎

Se $t(n)$ è un polinomio, allora $t^2(n)$ è un polinomio e $\log t(n) = O(\log n)$; il precedente risultato implica allora la seguente caratterizzazione di P in termini di circuiti:

Corollario 15.1 P *è la classe dei linguaggi che ammettono circuiti di dimensione polinomiale, uniformemente descrivibili in spazio* $O(\log n)$.

Analizziamo ora il problema di descrivere circuiti ammessi da linguaggi riconoscibili con vincoli sullo spazio. Supponiamo che M sia una macchina di Turing con un nastro di sola lettura ed un nastro di lavoro, che opera in spazio $s(n)$ e tempo $t(n)$. Una configurazione di tale macchina, fissata una parola w di lunghezza n sul nastro di ingresso, è data da $\langle q, k, k', \alpha \rangle$, dove q è lo stato del controllo, k e k' sono le posizioni delle testine sul nastro d'ingresso e sul nastro di lavoro, α è la parola contenuta nel nastro di lavoro. Il numero totale N di configurazioni è al più $n \cdot s(n) \cdot |\Gamma|^{s(n)} \cdot Q = 2^{O(\log n + s(n))}$, e quindi il tempo di calcolo è al più $N = 2^{O(\log n + s(n))}$, perché altrimenti si avrebbe una sequenza di computazione $i_0, \ldots, i_{t(n)}$ contenente la stessa configurazione ripetuta in due posizioni diverse, il che è impossibile (la sequenza di calcolo sarebbe infinita!). Data la parola w, sia la matrice $(B_{i,i'})$ dove gli indici i, i' variano sulle configurazioni ed inoltre:

$$B_{i,i'} = \begin{cases} 1 & \text{se } i' \text{ è la configurazione successiva di } i \\ 0 & \text{altrimenti.} \end{cases}$$

Consideriamo la potenza B^t di B rispetto al prodotto

$$(X \cdot Y)_{i,i'} = \bigvee_{i''} (X_{i,i''} \wedge Y_{i'',i'}).$$

Vale ovviamente che $B^t_{i,i'} = 1$ *se e solo se la macchina M inizializzata con la configurazione i, dopo t passi raggiunge la configurazione i'.* Se indichiamo con i_0 la configurazione iniziale e con $q(i)$ lo stato contenuto nella configurazione i, allora:

$$w \in L \quad \text{se e solo se} \quad \bigvee_i (B^{t(n)}_{i_0,i} \wedge (q(i) = q_s)) = 1.$$

Supponiamo, senza perdita di generalità, che $t(n)$ sia una potenza di 2, cioè, $t(n) = 2^H$. Il seguente circuito ξ_n, con variabili $A^{2^k}_{i,i'}$ (i, i' configurazioni, $0 \leq k \leq H = \log t(n)$) e u (variabile d'uscita), è un circuito per $L_n = L \cap \{0,1\}^n$:

$$\begin{aligned} & A_{i,i'} := B_{i,i'}(w); \\ & \vdots \\ & A^{2^k}_{i,i'} := \textstyle\bigvee_{i''} (A^{2^{k-1}}_{i,i''} \wedge A^{2^{k-1}}_{i'',i'}); \\ & \vdots \\ & u = \textstyle\bigvee_i (A^{2^H}_{i_0,i} \wedge (q(i) = q_s)). \end{aligned}$$

La famiglia $\{\xi_n\}$ è descrivibile uniformemente in spazio $O(\log N) = O(\log n + s(n))$. La funzione booleana $B_{i,i'}(w)$ dipende da un solo bit di w (da w_k, se $w = w_1 \ldots w_k \ldots w_n$, $i = \langle q, k, k', \alpha \rangle$) e quindi è realizzabile in profondità $O(1)$. Il calcolo $A^{2^k}_{i,i'} := \bigvee_{i''} (A^{2^{k-1}}_{i,i''} \wedge A^{2^{k-1}}_{i'',i'})$ richiede profondità $O(\log N)$ e dimensione $O(N)$. Ricordando che i possibili valori di k sono $O(\log N)$ e le coppie i, i' sono $O(N^2)$, concludiamo che

$$D(\xi_n) = O(\log^2 N) = O((\log n + s(n))^2) \quad \text{e} \quad C(\xi_n) = O(N^3 \cdot \log N).$$

Dunque, abbiamo provato:

Teorema 15.2 *Se L è riconosciuto da una MdT con nastro di sola lettura e nastro di lavoro in spazio $s(n)$, allora ammette una famiglia di circuiti $\{\xi_n\}$ descrivibile uniformemente in spazio $O(\log N)$ di dimensione $C(\xi_n) = O(N^3 \cdot \log N)$ e profondità $D(\xi_n) = O(\log^2 N)$, dove $\log N = O(\log n + s(n))$.*

Un caso particolare interessante è $s(n) = \log n$. In tal caso, $\log N = O(\log n)$ e $N^3 \cdot \log N = n^{O(1)}$:

Corollario 15.2 *Ogni linguaggio riconoscibile in spazio logaritmico ammette una famiglia di circuiti descrivibili uniformemente in spazio logaritmico, di dimensione polinomiale e profondità $O(\log^2 n)$.*

16 La classe NC dei problemi efficientemente parallelizzabili

Abbiamo precedentemente caratterizzato P come l'insieme di linguaggi che ammettono circuiti di dimensione polinomiale, descrivibili in spazio logaritmico. In generale, circuiti di dimensione polinomiale sono di profondità al più polinomiale: è particolarmente interessante isolare il caso in cui i circuiti hanno profondità drasticamente minore della dimensione, per esempio richiedendo che la profondità sia polilogaritmica, cioè, $O(\log^k n)$.

In tal caso, il tempo di esecuzione parallelo (la profondità del circuito) è decisamente inferiore al tempo di esecuzione sequenziale (la dimensione del circuito): il tempo di riconoscimento per linguaggi che ammettono tali tipi di circuiti è quindi in grado di trarre grandi vantaggi dalla possibilità di eseguire l'algoritmo di riconoscimento su più processori.

Definizione 16.1 NC *è la classe di linguaggi che ammettono famiglie $\{\xi_n\}$ di circuiti uniformemente descrivibili in spazio logaritmico $O(\log n)$, di dimensione polinomiale $n^{O(1)}$ e profondità polilogaritmica $\log^{O(1)} n$.*

Dalla definizione, NC $\subseteq$ P. Abbiamo inoltre già trovato due interessanti sottoclassi di NC: i linguaggi riconoscibili in spazio logaritmico e quelli che ammettono formule di dimensione polinomiale descrivibili uniformemente in spazio logaritmico. Nel primo caso, i linguaggi ammettono circuiti di profondità $O(\log^2 n)$, nel secondo $O(\log^k n)$. Vale L $\subseteq$ NC.

17 La classe NP

È ben noto che trovare la dimostrazione di un teorema è generalmente più difficile che verificare la correttezza di una preassegnata dimostrazione. Analogamente, risolvere un'equazione è di solito più difficile che controllare se una candidata soluzione verifica l'equazione stessa. Vogliamo qui definire in termini precisi la classe di problemi in cui, se è suggerita una soluzione candidata, allora è possibile verificare facilmente che essa è proprio una soluzione. A scopo esemplificativo, consideriamo il seguente problema:

SODD

ISTANZA: Una formula $\varphi(x_1, \ldots, x_n)$.

QUESTIONE: φ è soddisfacibile? In altre parole: esiste un assegnamento $\mathbf{a} \in \{0,1\}^n$ per cui $\varphi(\mathbf{a}) = 1$?

Osserviamo che vale:

1. Se φ è soddisfacibile, allora esiste un assegnamento $\mathbf{a}$ che "testimonia" o "dimostra" la soddisfacibilità, cioè tale che $\varphi(\mathbf{a}) = 1$. La dimensione $|\mathbf{a}|$ del "testimone" $\mathbf{a}$ è al più la dimensione $|\varphi|$ della formula; inoltre, verificare se $\varphi(\mathbf{a}) = 1$ dato $\mathbf{a}$ è fattibile in tempo polinomiale $|\varphi|^{O(1)}$.

2. Se φ non è soddisfacibile, allora nessun assegnamento soddisfa φ.

Chiameremo NP la classe dei linguaggi che hanno le caratteristiche sopra delineate:

Definizione 17.1 *Un linguaggio $L \subseteq \Sigma^*$ è nella classe* NP *se esiste una relazione $V(x, y)$ calcolabile in tempo polinomiale ed un polinomio $p(n)$ per cui*

$$L = \{x \mid \exists y \ (|y| = p(|x|) \wedge V(x, y) = 1)\}.$$

Vale cioè:

1. *Se $x \in L$ allora esiste una "breve" dimostrazione di appartenenza y, cioè $|y| = p(|x|)$ e $V(x, y) = 1$; inoltre il calcolo $V(x, y)$ può essere fatto in tempo polinomiale.*

2. *Se $x \notin L$ allora non esiste alcuna dimostrazione di lunghezza $p(|x|)$, cioè se $|y| = p(|x|)$ allora $V(x, y) = 0$.*

Dalle definizione di NP e di P (paragrafo 12), appare evidente che P $\subseteq$ NP.

18 Classificazione dei problemi: concetto di riduzione

Uno dei maggiori obbiettivi della complessità strutturale è quello di determinare a quale classe di complessità appartenga un dato problema. In questo contesto, ci limiteremo a considerare le importanti classi NC, P, NP.

NC individua la classe dei problemi "efficientemente risolubili con algoritmi paralleli", P quella dei problemi "efficientemente risolubili", NP contiene molti problemi di carattere pratico. Sappiamo che NC $\subseteq$ P $\subseteq$ NP e, anche se il problema della separazione di queste classi è ancora aperto, sembra ragionevole assumere che NC $\neq$ P $\neq$ NP. Vogliamo qui sviluppare delle tecniche per localizzare un dato problema rispetto a queste tre classi. Per fissare le idee, consideriamo i seguenti problemi:

VALORE DI FORMULA (FV)

ISTANZA: Una formula $F(x_1, \ldots, x_n)$; un assegnamento $(a_1, \ldots, a_n) \in \{0, 1\}^n$.

QUESTIONE: È $F(a_1, \ldots, a_n) = 1$?

VALORE DI CIRCUITO (CV)

ISTANZA: Un circuito $\phi(x_1, \ldots, x_n)$; un assegnamento $(a_1, \ldots, a_n) \in \{0, 1\}^n$.

QUESTIONE: È $\phi(a_1, \ldots, a_n) = 1$?

SODDISFACIBILITÀ DI CLAUSOLE DI HORN (H–SAT)

ISTANZA: Una congiunzione $F = C_1 \wedge \ldots \wedge C_m$ di clausole di Horn, dove una clausola di Horn è una clausola contenente tutte le variabili negate, ad esclusione al più di una.

QUESTIONE: Esiste un assegnamento che rende vera F?

SODDISDACIBILITÀ (SAT)

ISTANZA: Una congiunzione $F = C_1 \wedge \ldots \wedge C_m$ di clausole.

QUESTIONE: Esiste un assegnamento che rende vera F?

Per i primi tre problemi sono noti algoritmi risolutivi che lavorano in tempo polinomiale; il problema **FV**, inoltre, può essere risolto in tempo polilogaritmico. Quindi, $\mathbf{FV}, \mathbf{CV}, \mathbf{H\text{–}SAT} \in \mathrm{P}$, $\mathbf{FV} \in \mathrm{NC}$, $\mathbf{SAT} \in \mathrm{NP}$. Restano aperte le seguenti questioni: $\mathbf{CV}, \mathbf{H\text{–}SAT} \in \mathrm{NC}$? $\mathbf{SAT} \in \mathrm{P}$? Un possibile approccio alla soluzione di quest'ultima problematica consiste nell'introdurre una nozione di "riducibilità" tra problemi, cioè, una relazione binaria '$<$' tra problemi per cui succede che se $A < B$ e B è "computazionalmente facile", allora anche A è "computazionalmente facile". Evidentemente gli elementi massimali della relazione $<$ in una data classe potranno essere interpretati come i problemi "computazionalmente più difficili".

Considereremo nel nostro contesto la relazione $<_l$ di riducibilità in spazio logaritmico e la relazione $<_p$ di riducibilità in tempo polinomiale. Come al solito, le istanze di un problema sono codificate dalle parole in Σ^*, ed il problema è estensivamente denotato dal linguaggio $L \subseteq \Sigma^*$ formato dalle istanze per cui la questione ha risposta positiva.

Definizione 18.1 *Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$, diremo che L_1 è* polinomialmente riducibile "molti–uno" *a L_2 se esiste una funzione $f : \Sigma^* \to \Sigma^*$ tale che:*

1. *f è calcolabile in tempo polinomiale,*
2. *$x \in L_1$ se e solo se $f(x) \in L_2$.*

Scriveremo in tal caso $L_1 <_p L_2$. Se ulteriormente f è calcolabile in spazio logaritmico, diremo che L_1 è riducibile "molti–uno" in spazio logaritmico *a L_2 scrivendo $L_1 <_l L_2$.*

Le precedenti riduzioni sono giustificate dal seguente

Fatto 18.1 *Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$, allora:*

- *se $L_1 <_p L_2$ e $L_2 \in \mathrm{P}$, allora $L_1 \in \mathrm{P}$,*
- *se $L_1 <_l L_2$ e $L_2 \in \mathrm{NC}$, allora $L_1 \in \mathrm{NC}$.*

Dimostrazione. Sia $f : \Sigma^* \to \Sigma^*$ una riduzione in tempo polinomiale $p(n)$ tra L_1 e L_2, e sia L_2 calcolabile in tempo polinomiale $q(n)$. Il seguente algoritmo decide L_1:

INGRESSO: $x \in \Sigma^*$
  $y := f(x)$;
  if $y \in L_2$ **then** ACCETTA **else** RIFIUTA

La correttezza deriva immediatamente dal fatto che, essendo f una riduzione, $x \in L_1$ se e solo se $y \in L_2$. Sia $t(n)$ il tempo di esecuzione dell'algoritmo. Vale:

$$t(|x|) = p(|x|) + q(|y|).$$

Poiché la macchina che calcola f lavora in $p(|x|)$ passi e ad ogni passo stampa al più un carattere, la lunghezza della parola y stampata in uscita è al più $p(|x|)$, cioè, $|y| \leq p(|x|)$. Concludiamo che $t(|x|) \leq p(|x|) + q(p(|x|))$ è maggiorato da un polinomio, essendo i polinomi chiusi rispetto a composizione e somma. Questo implica che $L_1 \in$ P.

Sia ora f ulteriormente calcolabile in spazio logaritmico. Come visto al corollario 15.2, esiste una sequenza $\{\phi_n\}$ uniforme in spazio logaritmico di circuiti tali che:

1. ϕ_n ha profondità $O(\log^2 n)$,
2. $\phi_n(x) = f(x)$, per ogni $|x| = n$.

Se inoltre L_2 è descrivibile da una sequenza $\{\Psi_n\}$ di circuiti uniforme in spazio logaritmico e di profondità $O(\log^k n)$, allora $\{\Psi_n(\phi_n)\}$ è una famiglia uniforme per L_1 di profondità polilogaritmica $O(\log^2 n + \log^k n)$. ∎

19 Problemi P–completi e NP–completi

I problemi P–completi (rispettivamente, NP–completi) sono i problemi più difficili della classe P (rispettivamente, NP) rispetto alla riduzione $<_l$ (rispettivamente, $<_p$).

Definizione 19.1 *Un problema A è detto* P–completo *se:*

1. $A \in$ P,
2. $\forall X \in$ P $(X <_l A)$.

Un problema B è detto NP–completo *se:*

1. $B \in$ NP,
2. $\forall X \in$ NP $(X <_p B)$.

I problemi completi sono i più difficili della classe. Non è allora stupefacente il seguente

Fatto 19.1 *Dati due problemi A, B, allora:*

- *se A è* P*–completo e $A \in$* NC*, allora* NC = P,
- *se B è* NP*–completo e $B \in$* P*, allora* P = NP.

Dimostrazione. Sia X un qualsiasi problema in NC. Poiché A è P–completo, allora $X <_l A$ e, poiché $A \in$ NC, allora per il fatto 18.1 segue che $X \in NC$.

Analogamente per P = NP. ∎

Se ora accettiamo la congettura P $\neq$ NC, i problemi P–completi non possono appartenere a NC: essi identificano allora problemi che ammettono algoritmi sequenziali efficienti, ma che non possono essere parallelizzati (almeno nel senso forte da noi attribuito). Analogamente, i problemi NP–completi non possono essere risolti mediante algoritmi che lavorano in tempo polinomiale.

Esibiamo ora alcuni problemi P–completi e NP–completi.

Fatto 19.2 CV *è* P*–completo.*

Dimostrazione. Dato un linguaggio $X \in$ P, supponiamo che esso possa essere descritto da una famiglia di circuiti $\{\phi_n\}$ uniforme in spazio logaritmico. Allora la legge che associa ad ogni parola $x \in \Sigma^*$ l'istanza $(\phi_{|x|}, x)$ di **CV** è una riduzione in spazio logaritmico, e quindi $X <_l$ **CV**. ∎

Questo risultato mostra che, nonostante l'apparente somiglianza, **FV** e **CV** esibiscono proprietà molto diverse rispetto alla parallelizzazione: **FV** è efficientemente risolubile con algoritmi paralleli, **CV** non lo è (a meno che P=NC).

Fatto 19.3 H–SAT *è* P*–completo.*

Dimostrazione. Si verifica facilmente che la relazione $<_l$ è transitiva. Per dimostrare che **H–SAT** è P–completo basta allora dimostrare che il problema P–completo **CV** è riducibile a **H–SAT**.

Sia $(\xi, \mathbf{a})$ un assegnamento per **CV**, dove ξ è un circuito con variabili $x_1, \ldots, x_n$, variabili interne $z_1, \ldots, z_g$ e $\mathbf{a}$ è un assegnamento $(a_1, \ldots, a_n) \in \{0,1\}^n$. Si tratta di costruire in spazio logaritmico una congiunzione di clausole di Horn F che sia soddisfacibile se e solo se ξ nell'assegnamento $\mathbf{a}$ dà risposta 1. Consideriamo a tal riguardo le nuove variabili

$$X_{k,\alpha}, Z_{j,\beta} \quad (1 \leq k \leq n,\ 1 \leq j \leq g,\ \alpha, \beta \in \{0,1\})$$

col seguente significato:

$$X_{k,\alpha} = 1 \equiv X_k = \alpha,$$
$$Z_{j,\beta} = 1 \equiv Z_j = \beta.$$

All'assegnamento $(a_1, \ldots, a_n)$ associamo la formula A:

$$A = \bigwedge_k X_{k,a_k}.$$

Questa formula esprime il fatto che $x_k = a_k$ per ogni $1 \leq k \leq n$. Per ogni istruzione del circuito, ad esempio $z_k := op(z_j, z_s)$, associamo la seguente formula I_k:

$$I_k = \bigwedge_{a,b}((Z_{j,a} \wedge Z_{s,b}) \Rightarrow Z_{k,op(a,b)}).$$

Essa esprime il fatto che se $z_j = a$ e $z_s = b$, allora $z_k = op(a,b)$, cioè, proprio che $z_k := op(z_j, z_s)$. Supponiamo ora che, nel circuito ξ, quando $x_1 = a_1, \ldots, x_n = a_n$, allora le variabili interne valgano $z_1 = b_1, \ldots, z_g = b_g$. Per quanto detto, ogni assegnamento che rende vera la formula $A \wedge \bigwedge_{k=1}^{g} I_k$ deve essere tale che

$$1 = X_{1,a_1} = \ldots = X_{n,a_n} = Z_{1,b_1} = \ldots = Z_{g,b_g}.$$

Consideriamo ora la formula

$$F = A \wedge \bigwedge_{k=1}^{g} I_k \wedge Z_{g,1} \wedge (Z_{g,0} \Rightarrow \overline{Z_{g,1}}).$$

Vale:

1. Se l'uscita Z_g del circuito ξ su ingresso $a_1, \ldots, a_n$ vale 0, allora ogni assegnamento che verifica F, dovendo verificare $A \wedge \bigwedge_{k=1}^{g} I_k$, deve essere tale che $Z_{g,0} = 1$. Ma questo implica che $Z_{g,1} = 0$, e quindi F vale 0. Dunque, F non è soddisfacibile.

2. Se l'uscita Z_g di ξ su ingresso $a_1, \ldots, a_n$ vale 1, allora l'assegnamento $1 = X_{1,a_1} = \ldots = Z_{1,b_1} = \ldots = Z_{g,1}$, $0 = Z_{g,0}$ soddisfa F.

Poiché $a \Rightarrow b$ è equivalente a $\overline{a} \vee b$ e $\overline{a \wedge b}$ è equivalente a $\overline{a} \vee \overline{b}$, è facile trasformare F in una congiunzione di clausole. ∎

Il risultato precedente prova che **H–SAT** rimane P–completo anche quando le clausole contengono al più tre letterali.

Fatto 19.4 SAT *è* NP*–completo.*

Dimostrazione. Sia $L \in$ NP; esiste quindi una relazione $V(x, y)$ calcolabile in tempo polinomiale ed un polinomio $p(n)$ tale che:

$$L = \{x \mid \exists y \ (|y| = p(|x|) \wedge V(x, y) = 1)\}.$$

L ammette una famiglia di circuiti $\{\phi_n\}$ uniforme in spazio logaritmico; il circuito ϕ_n ha come variabili d'ingresso $x_1, \ldots, x_n, y_1, \ldots, y_{p(n)}$ e variabili interne $z_1, \ldots, z_{g(n)}$ per un opportuno polinomio $g(n)$. Come nella precedente dimostrazione, consideriamo le nuove variabili $X_{i,\alpha}$, $Y_{s,\beta}$, $Z_{k,j}$ e, per $1 \leq k \leq g(n)$, associamo alla k–esima istruzione la formula I_k e definiamo la formula F come:

$$F = \bigwedge_{k} I_k \wedge Z_{g(n),1} \wedge (Z_{g(n),0} \Rightarrow \overline{Z_{g(n),1}}).$$

Dalla precedente dimostrazione sappiamo che

(*) $X_{1,a_1} \wedge \ldots \wedge X_{n,a_n} \wedge Y_{1,b_1} \wedge \ldots \wedge Y_{p(n),b_{p(n)}} \wedge F$ è soddisfacibile se e solo se $V(a_1 \ldots a_n, b_1 \ldots b_{p(n)}) = 1$.

Consideriamo ora la formula B, dove:

$$B = (Y_{1,0} \vee Y_{1,1}) \wedge (Y_{2,0} \vee Y_{2,1}) \wedge \ldots \wedge (Y_{p(n),0} \vee Y_{p(n),1}).$$

Vale che $B = 1$ se e solo se esistono $b_1, \ldots, b_{p(n)}$ per cui $Y_{1,b_1} \wedge \ldots \wedge Y_{p(n),b_{p(n)}} = 1$, e quindi da **(*)** concludiamo che $X_{1,a_1} \wedge \ldots \wedge X_{n,a_n} \wedge B \wedge F$ è soddisfacibile se e solo se esiste $b_1, \ldots, b_{p(n)}$ per cui $V(a_1 \ldots a_n, b_1 \ldots b_{p(n)}) = 1$ se e solo se $a_1 \ldots a_n \in L$. ∎

Il precedente risultato prova che **SAT** rimane NP–completo (con riducibilità in spazio logaritmico, e quindi anche con riducibilità polinomiale) anche per formule che sono congiunzioni di clausole con al più tre letterali. Tale risultato non può essere migliorato (se P $\neq$ NP), poiché la soddisfacibilità di congiunzioni di clausole di due letterali è risolubile in tempo polinomiale.