Complessità e Teoria dell'Informazione

Prof.ssa Carla Piazza

Appunti di Claudio Desideri e Lara Vignotto – a.a. 2023/2024

Indice

Introduction

Programma:

- Teoria dell'Informazione
- Teoria della Complessità
- Algoritmi su Grafi ecc ...

Parte I

Teoria dell'Informazione

Capitolo 1

Introduzione

1.1 Probabilità, Entropia e Inferenza

Claude Shannon, 1948, *A Mathematical Theory of Communication*.

Un messaggio è una sequenza di lettere (simboli) da un alfabeto. Qual è l'informazione in una frase (messaggio)? Come possiamo misurare la quantità di informazione? Dipende dal contesto.

Esempio Il messaggio è "Piove!". Qual è la quantità di informazione? Per il signor Muller, che vive a Vienna, dove piove spesso, la quantità di informazione è bassa. Per Fatima, che vive nel deserto, invece, è alta.

Si ha quindi che

- Bassa probabilità di un evento

...

1.1.1 Teorema di Bayes

Sappiamo che nel caso di due **eventi indipendenti**, la probabilità congiunta è

$$p(a_i, b_j) = p(a_i) \cdot p(b_j)$$

Nel caso, invece, di due **eventi dipendenti** si ha

$$p(a_i, b_j) = p(a_i|b_j) \cdot p(b_j) = p(b_j|a_i) \cdot p(a_i)$$

e quindi

$$p(a_i|b_j) = \frac{p(b_j|a_i) \cdot p(a_i)}{p(b_j)}$$

con $p(b_j)$ fattore di normalizzazione.

TODO: vedere al capitolo 2 del libro il prior, poterior, likelihood, ecc.

1.1.2 Proprietà dell'Entropia

Libro, pag. 33. Le proprietà della funzione di entropia sono:

- $H(P) \geq 0$ con uguaglianza iff $p_i = 1$ per un qualche i. In particolare, $H(P) = 0$ iff $\exists i \; p_i = 1$, ovvero quando c'è un evento certo l'entropia è nulla.
- L'entropia è massimizzata se la distribuzione p è uniforme.

Analizziamo meglio e dimostriamo la seconda proprietà.

Proprietà 1.1.1 (Entropia massima)**.**

$$\mathcal{H}(P) \leq \log_2 |P|$$

con $|P|$ numero di eventi ($|X|$), e

$$\mathcal{H}\left(\frac{1}{k}, \frac{1}{k}, \ldots, \frac{1}{k}\right) = \log_2 k$$

dove k è il numero di eventi, ovvero $|P|$.

Dimostrazione La dimostrazione è basata su proprietà di funzioni complesse, in particolare sulla diseguaglianza di Jensen, la quale è una disuguaglianza che lega il valore di una funzione convessa al valore della medesima funzione calcolata nel valor medio del suo argomento.

Sia $\boldsymbol{f(x) = -x \log_2 x}$ (cfr. definizione di entropia). Vogliamo controllare se è concava o convessa, calcoliamo quindi la sua derivata seconda:

$$f''(x) = -\frac{1}{x}$$

Sappiamo che $0 \leq x \leq 1$ perché è una probabilità, e quindi abbiamo che $-1/x < 0$: la funzione è concava. Prendiamo ora due punti x_1 e x_2, e un punto x tra i due.

TODO: disegno

Abbiamo che la media pesata di x_1 e x_2 è

$$x = \lambda x_1 + (1 - \lambda) x_2 \qquad \text{con} \quad 0 \leq \lambda \leq 1$$

con λ il peso. In particolare, se $\lambda = 1$ allora $x = x_1$, se $\lambda = 0$ allora $x = x_2$, e se $\lambda = 1/2$ allora x si troverà esattamente a metà tra x_1 e x_2. Se prendiamo la combinazione lineare di $f(x_1)$ e $f(x_2)$, otteniamo la seguente disuguaglianza:

$$\lambda f(x_1) + (1 - \lambda) f(x_2) \leq f(\lambda x_1 + (1 - \lambda) x_2)$$

Tale disuguaglianza si può generalizzare alla combinazione lineare di un qualsiasi numero di punti. Se $f''(x) \leq 0$, $\forall x \in [x_1, x_n]$ (ovvero $f''(x)$ è concava) si ha la disuguaglianza di Jensen:

$$\sum_{i=1}^{n} \lambda_i f(x_i) \leq f\left(\sum_{i=1}^{n} \lambda_i x_i\right)$$

con $\lambda_i \geq 0$ e $\sum_{i=1}^{n} \lambda_i = 1$. La disuguaglianza cambia verso ($\geq$) quando la funzione è convessa, cioè $f''(x) \geq 0$. Ricordiamo che vogliamo arrivare alla combinazione lineare dove i punti hanno la stessa probabilità. Quindi con

$$\lambda_i = \frac{1}{k} \qquad |P| = |X| = k \qquad P = \{p_1, \ldots, p_k\}$$

scriviamo la disuguaglianza di Jensen come

$$\sum_{i=1}^{k} \underbrace{\frac{1}{k}}_{\lambda_i} \underbrace{(-p_i \log_2 p_i)}_{f(x_i)} \leq \underbrace{-\left(\sum_{i=1}^{k} \frac{1}{k} p_i\right) \log_2 \left(\sum_{i=1}^{k} \frac{1}{k} p_i\right)}_{f(\sum \lambda_i x_i)}$$

semplifichiamo perché $k > 0$

$$-\frac{1}{k} \sum_{i=1}^{k} p_i \log_2 p_i \leq -\frac{1}{k} \log_2 \frac{1}{k}$$

$$\mathcal{H}(P) \leq \log_2 k = \mathcal{H}\left(\frac{1}{k}, \ldots, \frac{1}{k}\right)$$

Ovvero l'entropia è massima per la distribuzione uniforme. □

Proprietà 1.1.2 (Entropia Congiunta). Siano P, Q due distribuzioni, e x_i, y_j coppia di eventi tali che $x_i \in P$ e $y_j \in Q$. L'entropia congiunta di P, Q è:

$$\mathcal{H}(P,Q) = -\sum_{i,j} p(x_i, y_j) \log_2(p(x_i, y_j))$$

Se x e y sono indipendenti (quindi la probabilità congiunta è il prodotto delle due probabilità) l'entropia è additiva:

$$\mathcal{H}(P,Q) = \mathcal{H}(P) + \mathcal{H}(Q)$$

La somma è possibile perché usiamo i logaritmi, e una delle loro proprietà è $\log(a \cdot b) = \log(a) + \log(b)$.

1.1.2.1 Scomponibilità dell'Entropia

Libro, pag. 33. Sia P una distribuzione (vettore) di probabilità, e X delle variabili.

$$\begin{aligned} P &= \{p_1,\ p_2,\ \ldots,\ p_n\} \\ X &= \{\underbrace{x_1}_{p_1}, \underbrace{x_2, \ldots, x_n}_{1-p_1}\} \end{aligned}$$

In questo contesto, la probabilità, ad esempio, del secondo evento x_2 è, normalizzata, pari a $p_2/1-p_1$, e quella dell'ultimo elemento x_n è $p_n/1-p_1$.

Esempio Abbiamo una moneta regolare. Al primo lancio esce H, e come risultati desiderati per il secondo e terzo lancio vogliamo T e T. Abbiamo quindi:

$$\underbrace{H}_{\substack{p_1=\frac{1}{2} \\ \frac{1}{2}}} \quad \underbrace{T \quad T}_{\substack{1-p_1=\frac{1}{2} \\ \frac{1}{4} \quad \frac{1}{4}}}$$

La quantità di informazione ricevuta da P è uguale a quella ricevuta dal processo in due passaggi.

$$\begin{aligned} \mathcal{H}(P) &= \sum p_i \log_2 \frac{1}{p_i} \\ &= \underbrace{\mathcal{H}(p_1, 1-p_1) + (1-p_1) \cdot \mathcal{H}\left(\underbrace{\frac{p_2}{1-p_1}, \frac{p_3}{1-p_1}, \ldots, \frac{p_n}{1-p_1}}_{\substack{p_i \text{ normalizzati la cui somma è 1} \\ 1-p_1 = p_2+p_3+\cdots+p_n}}\right)}_{\text{si possono dividere in diversi punti, ottenendo, ad esempio, tante entropie}} \end{aligned}$$

TODO: vdere proprietà libro pag 33 sez 2.6?

1.1.3 Inferenza

TODO: capitolo 3, esercizio 3.8 pag 57

Capitolo 2

Compressione

TODO: capitolo 4, esercizio 4.1 pag 66

2.1 Il Teorema Della Codifica Sorgente

Capitolo 4 del libro di MacKay. In questo capitolo discuteremo come misurare il contenuto informativo del risultato di un esperimento aleatorio.

Studiamo $\mathcal{H}(\{p, 1-p\})$, con $0 \leq p \leq 1$

$$\begin{aligned} \mathcal{H}(\{p, 1-p\}) &= \\ &= p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p} \\ &= -p \log(p) - (1-p) \log(1-p) \\ &= \mathcal{H}(p) \end{aligned}$$

TODO: finire
LEZ 3
TODO: soluzione esercizio 4.1
TODO: muddy children puzzle

2.2 Codici Simbolo

Capitolo 5 del libro di MacKay. Nell'ultimo capitolo abbiamo visto una prova dello status fondamentale dell'entropia come misura del contenuto informativo medio. Abbiamo definito uno schema di compressione dei dati utilizzando codici a blocchi di lunghezza fissa. Abbiamo così verificato la possibilità di compressione dei dati, ma la codifica a blocchi definita nella dimostrazione non ha fornito un algoritmo pratico. In questo capitolo studieremo algoritmi pratici di compressione dei dati.

Immagina un guanto di gomma pieno d'acqua. Se comprimiamo due dita del guanto, qualche altra parte del guanto deve espandersi, perché il volume totale dell'acqua è costante (l'acqua è essenzialmente incomprimibile). Allo stesso modo, quando accorciamo le parole in codice per alcuni risultati, ci devono essere altre parole in codice che si allungano, se lo schema non è *lossy*. In questo capitolo scopriremo l'equivalente teorico dell'informazione del volume dell'acqua.

Definizione 2.2.1 (Alfabeti di input/output)**.**

$$\begin{aligned} \text{Alfabeto di input} \quad & \mathcal{A} = \{a_1, a_2, \ldots, a_k\} \\ \text{Alfabeto di output} \quad & \mathcal{B} = \{b_1, b_2, \ldots, b_D\} \end{aligned}$$

Definizione 2.2.2 (Codice). Sia $\mathcal{A}^*$ un messaggio (sequenza di caratteri) sull'alfabeto $\mathcal{A}$. Il **codice** c è

$$c : \mathcal{A}^* \to \mathcal{B}^*$$

ovvero un messaggio dall'alfabeto $\mathcal{A}$ all'alfabeto $\mathcal{B}$ (iniettiva).

Con $\mathcal{A}^* = \bigcup_{n\in\mathbb{N}} A^n$, ovvero l'insieme di tutte le possibili stringhe che si possono creare utilizzando l'alfabeto $\mathcal{A}$, compresa la stringa vuota.

Si vuole comprimere il messaggio in modo da ottenere il messaggio più corto possibile. Per farlo, utilizziamo una codifica.

Definizione 2.2.3 (Codifica). Una codifica è una funzione

$$\varphi : \mathcal{A} \to \mathcal{B}^*$$

Inoltre

$$\varphi(\underbrace{x_1, x_2, \dots, x_m}_{\in\mathcal{A}^*}) = \varphi(x_1)\varphi(x_2)\dots\varphi(x_m)$$

Esempio 1 Alfabeti: $\mathcal{A} = \{a, b, c\}$, $\mathcal{B} = \{0, 1\}$; codifica: $\varphi(a) = 0$, $\varphi(b) = 10$, $\varphi(c) = 01$. È una buona codifica? No, perché è ambigua. Ad esempio

$$\varphi(ab) = 010 = \varphi(ca)$$

È iniettiva nella codifica ma non sul messaggio. Una codifica di questo tipo viene detta **not uniquely decodable** (non univocamente decodificabile).

Definizione 2.2.4 (Univocamente decodificabile). Un codice $\varphi : \mathcal{A} \to \mathcal{B}^*$ è univocamente decodificabile (uniquely decodable) se

$$\forall m_1, m_2 \in \mathcal{A}^* \qquad \varphi(m_1) \neq \varphi(m_2)$$

In questo corso non utilizzeremo codici non univocamente decodificabili.

Esempio 2 Alfabeti: $\mathcal{A} = \{a, b, c\}$, $\mathcal{B} = \{0, 1\}$; codifica: $\varphi(a) = 0$, $\varphi(b) = 01$, $\varphi(c) = 011$. Ad esempio, il messaggio *aabcbbca* viene codificato come $\varphi(aabcbbca) = 000101101010110$. È univocamente decodificabile (UD)?

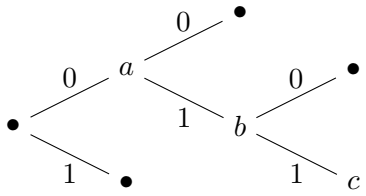
Sì, è UD con delay 1. Ad ogni 0, si controlla il carattere successivo: se è un altro 0, la lettera è a, altrimenti si prosegue fino al primo 1 per decidere se è b o c. Il delay 1 è riferito allo zero che si incontra, che significa l'inizio di un'altra lettera.

Esempio 3 Alfabeti: $\mathcal{A} = \{a, b, c\}$, $\mathcal{B} = \{0, 1\}$; codifica: $\varphi(a) = 00$, $\varphi(b) = 1$, $\varphi(c) = 10$. Ad esempio, il messaggio *bcbaca* viene codificato come $\varphi(bcbaca) = 1101001000$. È UD?

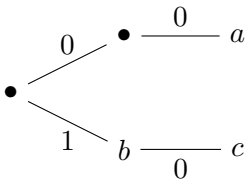
Sì, si controlla se c'è un numero pari o dispari di 0 dopo un 1: se è pari, si tratta di una b seguita da una o più a, se è dispari di una c, eventualmente seguita da una o più a. È UD con unbounded delay.

Abbiamo visto che nel caso di distribuzione uniforme, la quantità di informazione è pari all'entropia. Quanto è complesso computare una codifica/decodifica?

Poiché l'alfabeto di input è binario, per rappresentare una codifica si può utilizzare un albero binario. Quello per l'**Esempio 2** è il seguente:



Quello per l'**Esempio 3** è il seguente: TODO: fixare in modo che gli archi ad a e c siano inclinati



Quando si finisce in un nodo con un'etichetta ci si deve chiedere se la conclusione è che ci si può fermare. Il grado (fattore) di diramazione è pari alla cardinalità dell'alfabeto di output. Inoltre, se l'albero ha altezza h, tutte le codifiche hanno lunghezza h.

Ci chiediamo, qual è una codifica sicuramente UD e senza delay?

Definizione 2.2.5 (Codice prefisso)**.**

$$\varphi : \mathcal{A} \to \mathcal{B}^* \text{ è un codice prefisso}$$
$$\Updownarrow$$
$$\forall a_i, a_j \in \mathcal{A} \quad \varphi(a_i) \in \mathcal{B}^* \text{ non è un prefisso di } \varphi(a_j) \in \mathcal{B}^*$$

Al posto di codice prefisso (prefix code) utilizzeremo il termine prefisso (prefix). è corretto?

Esempio 3 (vedi sopra) φ non è un prefisso, perché la codifica di b è 1, che è un prefisso della codifica di c, ovvero 10.

Lemma 2.2.1. φ è un codice prefisso $\Rightarrow$ φ è UD senza delay

Per memorizzare l'albero si utilizza CONSTANT SPACE, che è un sottoinsieme di LINEAR TIME. Ciò equivale a dire che è possibile computarlo con un automa.

Esempio 3 (vedi sopra) $\varphi(bbb) = 111$: il messaggio di input ha lunghezza 3, e viene codificato in un messaggio di lunghezza uguale (3).

$\varphi(aca) = 001000$: il messaggio di input ha lunghezza 3, e viene codificato in un messaggio di lunghezza 6.

Il numero di possibili messaggi di lunghezza 3 in output è $2^3 = 8$.

Definizione 2.2.6 (Lunghezza media di una codifica, EL (Expected Length))**.** Siano $\varphi : \mathcal{A} \to \mathcal{B}^*$ una codifica, e P una distribuzione di probabilità sull'alfabeto di input $\mathcal{A}$

$$EL(\varphi) = \sum_{i=1}^{n} p(a_i) \cdot |\varphi(a_i)|$$

con $|\varphi(a_i)| = l_i$ lunghezza della codifica di a_i.

Esempio Supponiamo che nell'**Esempio 3** le probabilità siano $p(a) = 1/2$, $p(b) = 1/4$, $p(c) = 1/4$. La lunghezza media è

$$EL(\varphi) = \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 2 = 1 + \frac{3}{4}$$

Immaginiamo una codifica φ^* diversa, per la quale $|\varphi^*(a)| = 1$, $|\varphi^*(b)| = 2$, $|\varphi^*(c)| = 2$. La lunghezza media è

$$EL(\varphi^*) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{4} \cdot 2 = \frac{3}{2}$$

Abbiamo che $EL(\varphi) > EL(\varphi^*)$, quindi φ^* è migliore di φ.

Si vuole trovare la codifica con la minor EL sotto l'assunzione che la sorgente del messaggio non abbia memoria. Dati $\mathcal{A}$, $\mathcal{B}^*$, P, si vuole trovare la miglior codifica φ. Non è sufficiente considerare solo codici prefix-free.

Possiamo raggiungere il minimo di EL considerando i codici prefisso. EL è codificata da $\mathcal{H}(P)$. I codici possono essere asintotticamente ottimali, o ottimali.

2.2.1 Limite imposto dalla Decodificabilità Univoca

È possibile definire un codice $\varphi : \mathcal{A} \to \mathcal{B}^*$ che sia UD, date le lunghezze delle codifiche $l_1, \ldots, l_k$? Definiamo della terminologia:

$$k = |\mathcal{A}| \qquad D = |\mathcal{B}| (= 2 \text{ nel libro})$$

Teorema 2.2.2 (Disuguaglianza di Kraft-McMillan, o Teorema Inverso)**.**

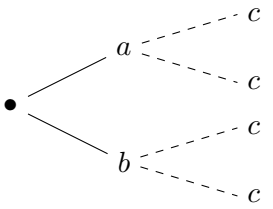
$$\varphi \text{ è UD} \quad \Rightarrow \quad \sum_{i=1}^{k} \frac{1}{D^{l_i}} \leq 1 \; = \; \sum_{i=1}^{k} D^{-l_i} \leq 1$$

Se > 1 non esiste un codice UD con tale lunghezza.

Esempio $\mathcal{A} = \{a, b, c\}$, $\mathcal{B} = \{0, 1\}$, $l_1 = 1$, $l_2 = 1$, $l_3 = 2$ (lunghezze delle codifiche di a, b, c). Applicando il teorema si ottiene

$$\frac{1}{2^1} + \frac{1}{2^1} + \frac{1}{2^2} > 1$$

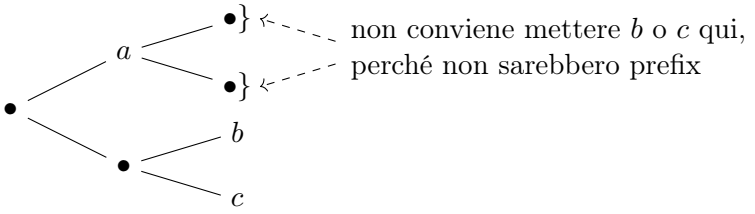
Infatti, non importa dove si sceglie di codificare la c, la codifica non è UD.



Esempio $\mathcal{A} = \{a, b, c\}$, $\mathcal{B} = \{0, 1\}$, $l_1 = 1$, $l_2 = 2$, $l_3 = 2$. Applicando il teorema si ottiene

$$\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^2} = 1$$

è UD (cfr. anche Teorema Diretto).



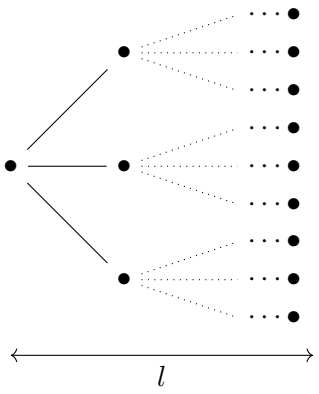
Teorema 2.2.3 (Teorema Diretto)**.**

$$\sum_{i=1}^{k} D^{-l_i} \leq 1 \quad \Rightarrow \quad \exists \varphi \text{ prefisso con lunghezze } l_1, \ldots, l_k$$

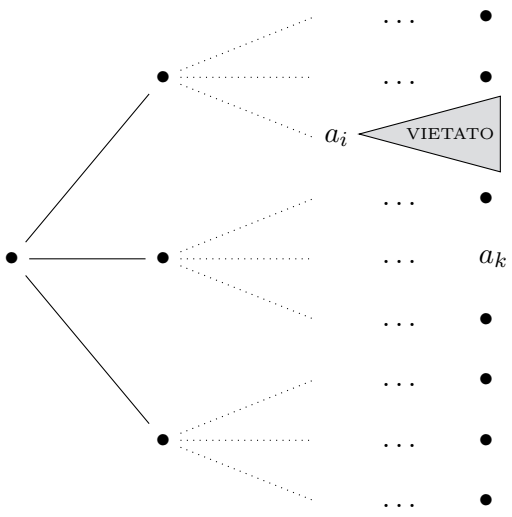
Prefisso è più forte di UD.

I due risultati (teoremi) affermano che, anche se ci limitiamo all'utilizzo di codici prefisso, non perdiamo alcuna potenza nella compressione. È sufficiente l'utilizzo dei codici prefisso, comprimono a sufficienza.

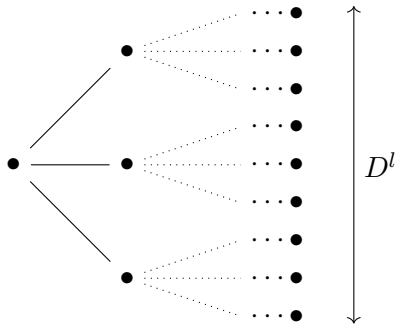
Dimostrazione Teorema Inverso, caso prefisso + Teorema Diretto Sia φ un codice prefisso e $l_1 \leq l_2 \leq \cdots \leq l_k = l$. Consideriamo l'albero D-ario (ogni nodo ha D figli) che rappresenta φ, di altezza l.



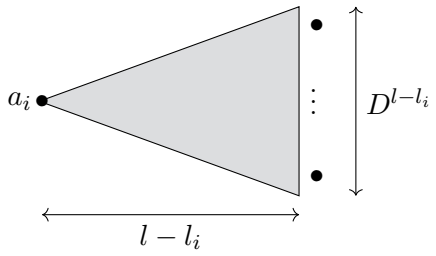
La codifica più lunga a_k è in una delle foglie. Supponiamo che la codifica a_i sia in uno dei nodi interni. Nessuno dei nodi interni (e in particolare nessuna delle foglie) del sottoalbero con a_i come radice può essere utilizzato per un'altra codifica.



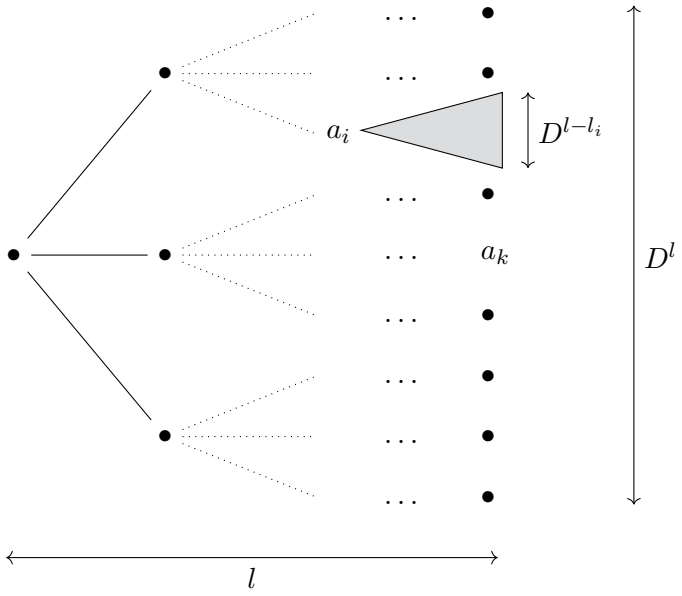
In un tale albero, il numero di foglie è pari a D^l.



Il numero di foglie di un sottoalbero che parte da un nodo interno a_i è D^{l-l_i}.



Quindi, riassumendo:



Scriviamo che la differenza tra il numero totale di foglie D^l e il numero di foglie dei sottoalberi (ovvero il numero di foglie vietate a causa dei sottoalberi creati da ogni a_i) è maggiore o uguale a 1:

$$D^l - \sum_{i=1}^{k-1} D^{l-l_i} \geq 1$$

$$D^l \left(1 - \sum_{i=1}^{k-1} D^{-l_i}\right) \geq 1$$

$$1 - \sum_{i=1}^{k-1} D^{-l_i} \geq D^{-l_k}$$

$$1 \geq \sum_{i=1}^{k} D^{-l_i}$$

□

Per il **Teorema Diretto**, si può leggere la dimostrazione "al contrario". In altre parole, vengono date le istruzioni per costruire l'albero, ovvero si disegna l'albero completo e si inizia ad etichettare. Più precisamente, si prende a_1, lo si mette a lunghezza l_1, e fino alle foglie si segna il resto dell'albero (il sottoalbero con radice a_1) come vietato. Si continua così per tutti gli a_i. □

Dimostrazione Teorema Inverso, caso φ UD Siano $l_1 \leq \cdots \leq l_k = l$ le lunghezze delle codifiche di $a_1, \ldots, a_k$. Consideriamo $\mathcal{N}(n,h)$ numero di stringhe su $\mathcal{A}^n$ (ovvero di lunghezza n) che hanno una codifica φ UD di lunghezza h. Sia $|\mathcal{B}^h| = D^h$ (numero di stringhe di lunghezza h su $\mathcal{B}$). Poiché φ è UD, $\mathcal{N}(n,h) \leq D^h$.

$$\sum_{i=1}^{k} D^{-l_i} = D^{-l_1} + D^{-l_2} + \cdots + D^{-l_k}$$

Studiamo la crescita di tale oggetto alla potenza di n, quando n va ad infinito. Questo perché, se la somma è > 1, allora la potenza va ad infinito; se la somma è < 1, allora la potenza va a 0 (è limitata); se la somma è $= 1$, allora la potenza va a 1.

$$\forall n \qquad \left(D^{-l_1} + D^{-l_2} + \cdots + D^{-l_k}\right)^n \qquad \text{(chiamiamola } \alpha^n\text{)}$$

Se si svolge l'elevamento a potenza di tale polinomio, si otterrà una serie di addendi del seguente tipo:

$$D^{-l_1 \cdot n} + \cdots + D^{(-l_1)+(-l_2)+(-l_1)+\ldots} + \cdots + D^{-l_k \cdot n}$$

dove $-l_1 \cdot n$ è la lunghezza della codifica della stringa $a_1 a_1 \ldots a_1$, con n ripetizioni di a_1, ovvero $|\varphi(a_1 \ldots a_1)| = l_1 \cdot n$. Allo stesso modo, $-l_k \cdot n$ è la lunghezza della codifica della stringa $a_k a_k \ldots a_k$, con n ripetizioni di a_k. Un generico esponente all'interno, ad esempio $-(l_1 + l_2 + l_1 + \ldots)$ è una somma di lunghezze di codifiche, e ammonta alla lunghezza di una generica stringa. Ad esempio, $-(10)$, se chiamo $10 = h$.

Si ha che D^{-h} si verifica nella somma esattamente $\mathcal{N}(n,h)$ volte (alcune delle quali saranno 0). Quindi, la somma $D^{-l_1 \cdot n} + \cdots + D^{-l_k \cdot n}$ si può riscrivere come

$$\begin{aligned}
\mathcal{N}(n,0)D^{-0} + \mathcal{N}(n,1)D^{-1} + \cdots + \underbrace{\mathcal{N}(n, n\cdot l)}_{\geq 1} D^{-n\cdot l} &\leq D^0 D^{-0} + D^1 D^{-1} + \cdots + D^{n\cdot l} D^{-n\cdot l} \\
1 + 1 + \cdots + 1 &\leq n \cdot l + 1 \\
\forall n \qquad \alpha^n &\leq \underbrace{l}_{costante} \cdot n
\end{aligned}$$

Da cui si ricava

$$\alpha \leq 1$$

Quindi

$$\sum_{i=1}^{k} D^{-l_i} \leq 1$$

□

2.2.2 Compressione Massima

La sorgente che genera il messaggio del codice è stazionaria e senza memoria (Fig. 2.1 nella pagina seguente). Poiché il messaggio codificato deve passare attraverso un canale, il quale ha una sua capacità e una sua velocità, lo si vuole comprimere il più possibile.

Ricordiamo che

$$P = \{p_1, \ldots, p_k\} \qquad \mathcal{A} = \{a_1, \ldots, a_k\} \qquad l_i = |\varphi(a_i)| \qquad EL(\varphi) = \sum_{i=1}^{k} p_i l_i$$

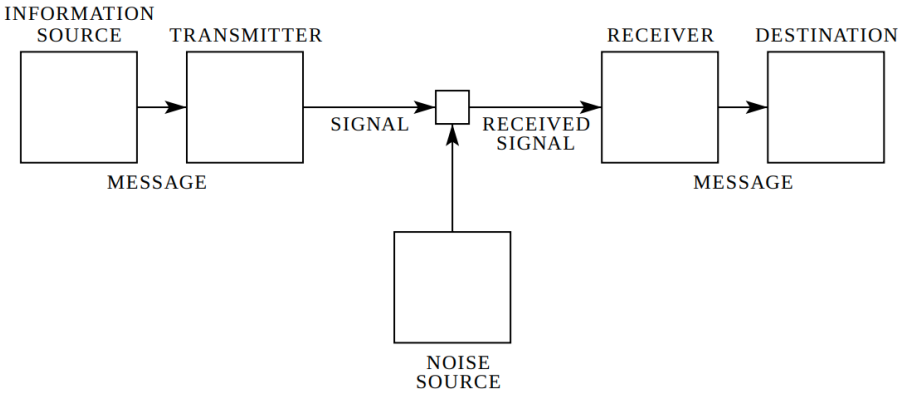


Figura 2.1: Diagramma schematico di un sistema di comunicazione, da C. E. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, 1948.

Teorema 2.2.4 (1° Shannon)**.**

$$\varphi \text{ è UD} \quad \Rightarrow \quad EL(\varphi) \geq \mathcal{H}_D(P)$$

con

$$\mathcal{H}_D(P) = \sum_{i=1}^{k} p_i \cdot \log_D \frac{1}{p_i}$$

Dimostrazione

$$\begin{aligned} EL(\varphi) - \mathcal{H}_D(P) &= \sum_{i=1}^{k} p_i \underbrace{l_i}_{\log_D D^{l_i}} + \sum_{i=1}^{k} p_i \cdot \log_D p_i \\ &= \sum_{i=1}^{k} p_i \cdot \log_D (D^{l_i} \cdot p_i) \end{aligned}$$

Prima di proseguire, ricordiamo la seguente proprietà dei logaritmi su $\mathbb{N}$:

$$\log_e x \leq x - 1; \qquad -\log_e x \geq -(x-1)$$

e anche la proprietà dei logaritmi:

$$\log_b x = \frac{\log_c x}{\log_c b}$$

Continuiamo la dimostrazione:

$$\begin{aligned} EL(\varphi) - \mathcal{H}_D(P) &= \sum_{i=1}^{k} p_i \cdot \log_D (D^{l_i} \cdot p_i) \\ &= \frac{1}{log_e D} \sum_{i=1}^{k} p_i \cdot \log_e (D^{l_i} \cdot p_i) \\ &= -\frac{1}{\log_e D} \sum_{i=1}^{k} p_i \cdot \log_e \left(\frac{1}{D^{l_i} \cdot p_i} \right) \\ &\geq -\frac{1}{\log_e D} \sum_{i=1}^{k} p_i \cdot \left(\frac{1}{D^{l_i} \cdot p_i} - 1 \right) \end{aligned}$$

$$= -\frac{1}{\log_e D} \underbrace{\left(\sum_{i=1}^{k} \frac{1}{D^{l_i}} - \underbrace{1}_{\sum p_i} \right)}_{\leq 0}$$

$$\geq 0$$

□

2.2.3 Shannon Code

Dal Teorema 1° Shannon abbiamo:

$$EL(\varphi) = \sum_{i=1}^{k} p_i l_i \;\geq\; \mathcal{H}_D(P) = \sum_{i=1}^{k} p_i \log_D \frac{1}{p_i}$$

La differenza sta in l_i e $\log_D \frac{1}{p_i}$, quindi vogliamo provare ad eguagliarli:

$$l_i = \log_D \frac{1}{p_i}$$

Ma $\log_D \frac{1}{p_i}$ non è necessariamente intero. Decidiamo quindi di considerare il suo primo intero più grande:

$$l_i = \left\lceil \log_D \frac{1}{p_i} \right\rceil = \lceil -\log_D p_i \rceil$$

È sempre possibile definire un codice UD con tali lunghezze? Possiamo utilizzare Kraft-McMillan. Vogliamo controllare se

$$\sum_{i=1}^{k} D^{-\lceil -\log_D p_i \rceil} \overset{?}{\leq} 1$$

Sappiamo che

$$\lceil -\log_D p_i \rceil = -log_D p_i + \beta_i \qquad\qquad 0 \leq \beta_i < 1$$

Quindi possiamo scrivere

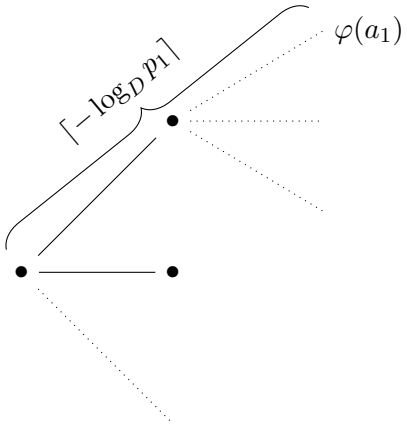
$$\begin{aligned} \sum_{i=1}^{k} D^{-(-log_D p_i + \beta_i)} &= \sum_{i=1}^{k} D^{log_D p_i} \cdot D^{\beta_i} \\ &= \sum_{i=1}^{k} p_i \cdot D^{-\beta_i} \\ &= \sum_{i=1}^{k} p_i \cdot \frac{1}{D^{\beta_i}} \end{aligned}$$

Ricordiamo che $0 \leq \beta_i < 1$ e $D > 1$, di conseguenza $\frac{1}{D^{\beta_i}} \leq 1$. Quindi

$$\sum_{i=1}^{k} p_i \cdot \frac{1}{D^{\beta_i}} \leq 1$$

$$\sum_{i=1}^{k} D^{-\lceil -\log_D p_i \rceil} \leq 1$$

Esiste quindi un prefix code con lunghezze $l_i = \lceil -\log_D pi \rceil$ definibile utilizzando una strategia greedy sull'albero D-ario.

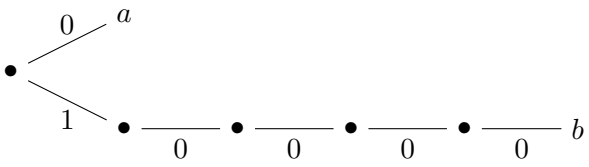


Esempio Siano $\mathcal{A} = \{a, b\}$, $\mathcal{B} = \{0, 1\}$, $P = \{1 - \frac{1}{32}, \frac{1}{32}\}$. Abbiamo che

$$l_2 = -\log_2 32 = 5 \qquad l_1 = \left\lceil -\log_2 \left(1 - \frac{1}{32}\right) \right\rceil = 1$$

Ciò significa che lo shannon code codifica l'alfabeto come

$$\varphi(a) = 0 \qquad \varphi(b) = 10000$$



TODO: finire lezione 5
TODO: osservazione su φ_{SF} in appunti lez 6

Definizione 2.2.7 (Efficienza (Efficiency of code))**.**

$$Eff(\varphi) = \frac{\mathcal{H}_D(P)}{EL(\varphi)}$$

Eff è sempre ≤ 1, e ci dice quanto siamo vicini all'entropia, che non è sempre raggiungibile.

Ci chiediamo perché Huffmann (merge delle foglie) è ottimale, mentre Shannon-Fano (splitting dalla radice) non lo è?

2.3 Codici Stream

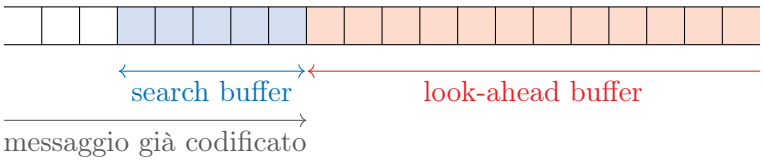
Capitolo 6 del libro di MacKay. In questo capitolo discuteremo uno schema di compressione dei dati. La codifica Lempel-Ziv è un metodo "universale", progettato secondo la filosofia per cui vorremmo un unico algoritmo di compressione che faccia un lavoro ragionevole per qualsiasi fonte. In effetti, per molte fonti della vita reale, le proprietà universali di questo algoritmo valgono solo nel limite di quantità di dati eccessivamente grandi, ma, comunque, la compressione Lempel-Ziv è ampiamente utilizzata e spesso efficace.

2.3.1 Un po' di Storia

TODO: scrivere la storia

2.3.2 Lempel-Ziv

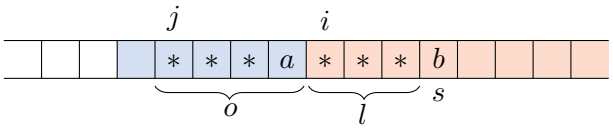
Immaginiamo un messaggio come uno stream (vettore) di caratteri su $\mathcal{A}^*$.



Cerchiamo il prefisso più lungo del look-ahead buffer che sia presente nel search buffer. Se lo troviamo, lo sostituiamo con un puntatore alla sua posizione nel search buffer e la sua lunghezza. Se non lo troviamo, scriviamo il carattere e passiamo al carattere successivo.

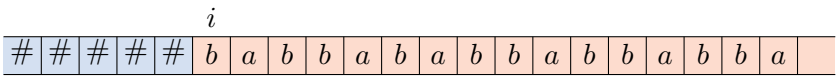
Ad ogni iterazione, l'algoritmo produce triple della forma (o, l, s), dove

- o è l'offset, ovvero la distanza tra i (primo carattere del look-ahead buffer) e j (primo carattere del prefisso nel search buffer)
- l è la lunghezza del prefisso
- s è il primo carattere mismatching

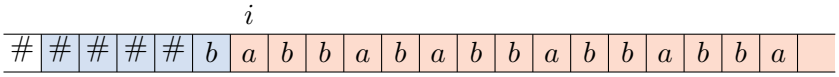


2.3.2.1 LZ77

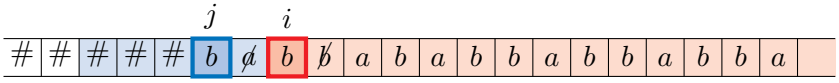
Abbiamo un search buffer di lunghezza 5, e la stringa *babbababbabbabba*. All'inizio, il search buffer è vuoto, e possiamo quindi immaginarlo con caratteri al di fuori dell'alfabeto.



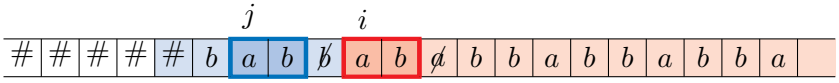
La prima tripla (o, l, s) della codifica è quindi $(0, 0, b)$. Proseguiamo spostando il search buffer di un carattere a destra:



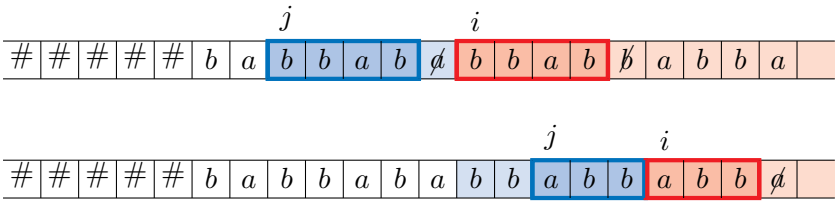
Il prefisso del look-ahead buffer inizia per a, e quindi non è presente nel search buffer. La seconda tripla è $(0, 0, a)$, e il search buffer viene spostato di un carattere a destra:



Confrontiamo nuovamente il prefisso del look-ahead buffer con il search buffer. Il prefisso inizia per b, che è presente anche nel search buffer. Continua con b, ma nel search buffer la b è seguita da una a. Il prefisso è quindi b, di lunghezza 1, la distanza tra j e i è 2, e il primo carattere mismatching è b. La terza tripla è quindi $(2, 1, b)$, e il search buffer viene spostato di due caratteri a destra:



Anche in questo caso cominciamo a guardare i caratteri del look-ahead buffer, e cerchiamo il più lungo prefisso presente anche nel search buffer. Il prefisso è ab (lunghezza 2), la distanza tra j e i è 3, e il primo carattere mismatching è una a. La tripla è quindi $(3, 2, a)$, e proseguiamo spostando il search buffer di tre caratteri a destra. Gli ultimi due passaggi sono i seguenti:



Corrispondenti alle triple $(5, 4, b)$ e $(3, 3, a)$ rispettivamente.

La codifica del messaggio *babbababbabbabba* con l'algoritmo LZ77 è quindi rappresentata dalla sequenza di triple $(0, 0, b)(0, 0, a)(2, 1, b)(3, 2, a)(5, 4, b)(3, 3, a)$.

L'esempio appena mostrato utilizza un messaggio piuttosto corto, e quindi non è possibile apprezzare la compressione. Per messaggi più lunghi, la dimensione delle triple è molto più piccola della dimensione del messaggio originale.

In quale caso LZ77 comprime molto? Immaginiamo di avere il messaggio *aaaaaaaaaaaaaab*. Il primo mismatch si trova solo all'ultimo carattere, tutta la parte di ripetizione di a è il prefisso più lungo. Nel caso migliore, si può ottenere una compressione esponenziale.

Per questo tipo di codifica, il decodificatore deve essere a conoscenza dell'alfabeto e della lunghezza del search buffer. Più precisamente, deve avere un buffer di lunghezza almeno pari alla lunghezza del buffer del codificatore.

Note La sorgente che genera le lettere del messaggio è stazionaria e senza memoria. Le probabilità non cambiano nel tempo.

LZ esegue i cambiamenti sulle probabilità durante la codifica, mentre S e SF no. Questi ultimi due dovrebbero costruire un altro albero se le probabilità cambiano.

Capitolo 3

Codifica di Canale Rumoroso

3.1 Variabili Aleatorie Dipendenti

Capitolo 8 del libro di MacKay. Negli precedenti capitoli sulla compressione dei dati ci siamo concentrati sui vettori aleatori x provenienti da una distribuzione di probabilità estremamente semplice, ovvero la distribuzione separabile in cui ciascuna componente x_n è indipendente dalle altre.

In questo capitolo considereremo insiemi congiunti in cui le variabili aleatorie sono dipendenti. Questo materiale ha due motivazioni. Innanzitutto, i dati del mondo reale hanno correlazioni interessanti, quindi per eseguire una buona compressione dei dati, dobbiamo sapere come lavorare con modelli che includono dipendenze. In secondo luogo, un canale rumoroso con input x e output y definisce un insieme congiunto in cui x e y sono dipendenti (se fossero indipendenti, sarebbe impossibile comunicare sul canale) quindi la comunicazione sui canali rumorosi è descritto in termini di entropia degli insiemi congiunti.

3.1.1 Divergenza e Disuguaglianza di Gibbs

La divergenza di Kullback-Leibler è una misura della differenza tra due distribuzioni di probabilità. Siano P e Q due distribuzioni di probabilità su X. La divergenza di Kullback-Leibler tra P e Q è definita come

$$\begin{aligned} D(P||Q) &= \sum_{x\in X} P(x)\log\left(\frac{P(x)}{Q(x)}\right) \\ &= \sum_{x\in X} P(x)\left(\log\left(\frac{1}{Q(x)}\right) - \log\left(\frac{1}{P(x)}\right)\right) \end{aligned}$$

Notiamo che $\sum_{x\in X} P(x)\log\left(\frac{P(x)}{Q(x)}\right)$ è il vaore atteso di $\log\left(\frac{P(x)}{Q(x)}\right)$ rispetto a P. Inoltre, $\log\left(\frac{P(x)}{Q(x)}\right)$ può essere espresso come $\log\left(\frac{1}{Q(x)}\right) - \log\left(\frac{1}{P(x)}\right)$, con $-P(x)\log\left(\frac{1}{P(x)}\right)$ entropia della distribuzione P.

Il significato della divergenza, in questo contesto, è "quanti bit in più devo usare per codificare una sorgente utilizzando Q al posto di P", con P distribuzione reale dei dati.

Esempio Se si hanno dei caratteri con probabilità 0, si utilizzerà per essi una codifica molto lunga, così da conservare codifiche brevi per altri caratteri con probabilità maggiori.

$$\begin{aligned} \mathcal{A} &= \{a, b, c, d, e\} \\ P &= \{\ldots, \ldots, \ldots, 0, 0\} \\ Q &= \{0.2, 0.2, 0.2, 0.2, 0.2\} \end{aligned}$$

Non conosciamo la distribuzione reale dei dati P, ma abbiamo dedotto Q.

Proprietà La divergenza non è simmetrica.

$$D(P||Q) \neq D(Q||P)$$

La divergenza è sempre non negativa, e vale 0 se e solo se $P = Q$.

$$D(P||Q) \geq 0 \qquad D(P||Q) = 0 \Leftrightarrow P = Q$$

Quest'ultima proprietà è una conseguenza della **disuguaglianza di Gibbs**.

3.1.2 Entropia e Mutual Information

Siano X, Y variabili aleatorie. Come abbiamo già visto, l'entropia congiunta è

$$\mathcal{H}(X,Y) = \sum_{\substack{x\in X \\ y\in Y}} p(x,y) \log\left(\frac{1}{p(x,y)}\right)$$

L'entropia è additiva sse X e Y sono indipendenti

$$\mathcal{H}(X,Y) = \mathcal{H}(X) + \mathcal{H}(Y) \quad \Leftrightarrow \quad p(x,y) = p(x)p(y)$$

L'entropia condizionale di X dato $Y = y$, ovvero se si conosce il valore di una delle due variabili, è l'entropia della distribuzione di probabilità $P(x|y)$

$$\mathcal{H}(X|Y=y) = \sum_{x\in X} p(x|y) \log\left(\frac{1}{p(x|y)}\right)$$

L'entropia condizionale di X dato Y è la media, su tutti i valori di Y, dell'entropia condizionale di X dato $Y = y$. L'incertezza del valore di X se si conosce il valore di Y è quindi

$$\begin{aligned}\mathcal{H}(X|Y) &= \sum_{y\in Y} p(y)\mathcal{H}(X|Y=y) \\ &= \sum_{\substack{x\in X \\ y\in Y}} p(x,y) \log\left(\frac{1}{p(x|y)}\right)\end{aligned}$$

***X* e *Y* indipendenti** Se X e Y sono indipendenti, allora

$$\begin{gathered}p(x,y) = p(x)p(y) \\ p(x|y) = p(x) \quad \Rightarrow \quad \mathcal{H}(X,Y) = \mathcal{H}(X)\end{gathered}$$

(Dimostrare che se X e Y sono indipendenti, allora $\mathcal{H}(X,Y) = \mathcal{H}(X)$)

***X* e *Y* dipendenti** Conosciuta anche come *chain rule* per l'entropia. L'entropia congiunta, $\mathcal{H}(X,Y)$, l'entropia condizionale, $\mathcal{H}(X|Y)$ o $\mathcal{H}(Y|X)$, e l'entropia marginale, $\mathcal{H}(X)$ o $\mathcal{H}(Y)$, sono legate dalla seguente relazione

$$\begin{aligned}\mathcal{H}(X,Y) &= \mathcal{H}(X) + \mathcal{H}(Y|X) \\ &= \mathcal{H}(Y) + \mathcal{H}(X|Y)\end{aligned}$$

Abbiamo un analogo al caso delle probabilità, solo con l'addizione al posto della moltiplicazione, in quanto passiamo alla funzione logaritmo.

3.1.2.1 Mutual Information

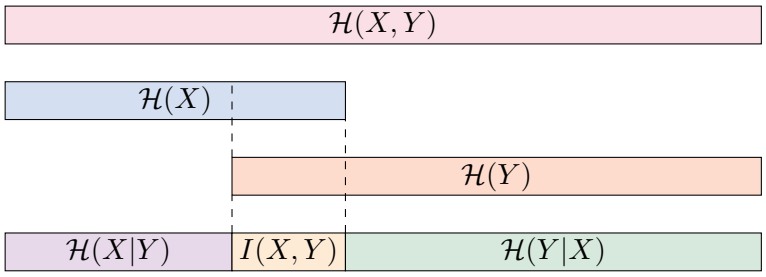
La **mutual information** è una misura della dipendenza tra due variabili aleatorie. È definita come

$$\begin{aligned} I(X,Y) &= \mathcal{H}(X) - \mathcal{H}(X|Y) \\ &= \mathcal{H}(Y) - \mathcal{H}(Y|X) \end{aligned}$$

Misura la riduzione media dell'incertezza su X se si conosce il valore di Y, o viceversa. Vale che $I(X,Y) \geq 0$, e in particolare $I(X,Y) = 0$ se X e Y sono indipendenti. Inoltre, $I(X,Y)$ è massimo quando l'incertezza su X se si conosce Y (o viceversa) è nulla:

$$X = Y \quad \Rightarrow \quad \mathcal{H}(X|Y) = 0 \quad \Rightarrow \quad I(X,Y) = \mathcal{H}(X)$$

Rappresentazione visuale $I(X,Y) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y)$



Teorema 3.1.1. La mutual information è pari alla divergenza tra la distribuzione congiunta e il prodotto delle distribuzioni marginali

$$I(X,Y) = D(XY||X \otimes Y)$$

con XY distribuzione congiunta $p(X,Y)$, e $X \otimes Y$ prodotto delle distribuzioni marginali $p(X) \cdot p(Y)$.

Se X e Y sono indipendenti, allora $p(X,Y) = p(X) \cdot p(Y)$, e quindi $I(X,Y) = 0$.

Capitolo 4

Kolmogorov Complexity

Prerequisiti al capitolo 2 del libro di Papadimitriou. Argomenti al capitolo 8 del libro di T. Cover, nel libro di Li e Vitanyi, al capitolo 3 del libro di Papadimitriou.

4.1 Nozioni Preliminari

La complessità di Kolmogorov è una nozione di complessità algoritmica. Questo argomento è un "ponte" tra le due parti di questo corso.

Mentre negli anni '40 Shannon (Princeton, Bell Labs, MIT), Fano (PoliTO, MIT), e Huffmann (MIT, UCAL Santa Cruz) lavoravano nella parte occidentale del mondo, dall'altra parte della cortina di ferro, nel 1965 Kolmogorov (Moscow University) analizzava la nozione di casualità (randomness) delle stringhe.

Ad esempio, se si riceve la stringa 0100110100111010, o la stringa 1111111111111111, quale delle due è più casuale? Poiché la sorgente che le genera è stazionaria e senza memoria, entrambe le stringhe sono equiprobabili. Kolmogorov non era soddisfatto di questa nozione di casualità.

Intuitivamente, la sua definizione di complessità di un oggetto è la lunghezza del programma più corto in grado di generarlo.

4.1.1 Macchine di Turing

Una macchina di Turing è un modello di calcolo inventato da Alan Turing nel 1936. È possibile immaginarla come un nastro, o registro, con un numero infinito di celle. In ogni cella si possono scrivere dei simboli dall'alfabeto Σ, o dei simboli speciali. In un dato momento, con un puntatore è possibile leggere il contenuto di una cella, eventualmente cambiarlo, cambiare lo stato del puntatore, e spostarsi a destra o a sinistra di una cella (o rimanere fermi).

Definizione 4.1.1 (Macchina di Turing)**.** Una macchina di Turing è una tupla $\mathcal{M}(K, \Sigma, \delta, s)$, dove

- K è un insieme finito di stati, di cui $s \in K$ è quello iniziale
- Σ è un alfabeto finito, e $\triangleright, \sqcup \in \Sigma$. Inoltre, $\Sigma \cap K = \emptyset$
- δ è la funzione di transizione, definita come $\delta : K \times \Sigma \to K \cup \{\text{yes, no, halt}\} \times \Sigma \times \{\rightarrow, \leftarrow, -\}$

$\forall q \in K$ con $\delta(q, \triangleright) = (q', \triangleright, \rightarrow)$.

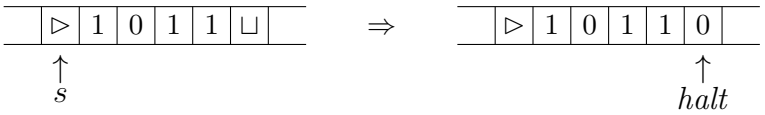
Il simbolo $\triangleright$ si legge *start*, $\sqcup$ si legge *blank*, e $\rightarrow, \leftarrow, -$ indicano rispettivamente il movimento a destra (*right*), a sinistra (*left*), o l'immobilità del puntatore (*stay*).

Esempio: raddoppia un numero Siano $\Sigma = \{0, 1, \triangleright, \sqcup\}$, $K = \{s\}$. Le funzioni di transizione per la macchina di Turing che raddoppia un numero sono

$$\delta(s, \triangleright) = (s, \triangleright, \rightarrow)$$

$$\delta(s, 0) = (s, 0, \rightarrow)$$
$$\delta(s, 1) = (s, 1, \rightarrow)$$
$$\delta(s, \sqcup) = (halt, 0, -)$$

Partendo dalla stringa 1011 e dalla macchina di Turing appena definita, si ottiene la stringa 10110.



4.2 Complessità di Kolmogorov

Definizione 4.2.1 (Macchine di Turing Universali)**.** Una macchina di Turing universale è una macchina di Turing $\mathcal{U}$ che prende in input un'altra macchina di Turing $\mathcal{M}$ e un input x, e simula l'esecuzione di $\mathcal{M}$ su x.

$$\mathcal{U}(\mathcal{M}; x) = \mathcal{M}(x)$$

La tesi di Church-Turing afferma che ogni funzione calcolabile può essere calcolata da una macchina di Turing. Di conseguenza, le macchine di Turing universali devono esistere. Intuitivamente, una machina di Turing universale può essere vista come un compilatore.

Definizione 4.2.2 (Kolmogorov Complexity)**.** La complessità di Kolmogorov di una stringa $x \in \Sigma^*$ è la lunghezza della macchina di Turing $\mathcal{M}$ più corta tale che $\mathcal{U}(\mathcal{M}) = x$.

$$K_{\mathcal{U}}(x) = \min_{\mathcal{M}:\mathcal{U}(\mathcal{M})=x} |\mathcal{M}|$$

con $\mathcal{U}$ una macchina di Turing universale fissata.

Il nastro della macchina universale $\mathcal{U}$ è composto da due parti: la prima parte contiene la codifica binaria della macchina $\mathcal{M}$, e la seconda parte contiene l'input x.

Esempio Macchina di Turing $\mathcal{M}$ che non prende nulla in input e produce in output la stringa 0110.

Per descrivere $\mathcal{M}$, è necessario specificare K, Σ, δ, s, con $\delta(s, \triangleright) = (q_1, \triangleright, \rightarrow)$, $\delta(q_1, \sqcup) = (q_2, 0, \rightarrow)$, $\delta(q_2, \sqcup) = (q_3, 1, \rightarrow)$, $\delta(q_3, \sqcup) = (q_4, 1, \rightarrow)$, $\delta(q_4, \sqcup) = (halt, 0, -)$. Per codificare $\mathcal{M}$ in binario, bisogna specificare in binario gli argomenti descritti precedentemente.

Se si considerano le macchine di Turing che producono 0110 in output e non prendono nulla in input, la lunghezza della più corta è la complessità di Kolmogorov della stringa 0110.

Definizione 4.2.3 (Complessità di Kolmogorov Condizionale)**.**

$$K_{\mathcal{U}}(x|y) = \min_{\mathcal{M}:\mathcal{U}(\mathcal{M};y)=x} |\mathcal{M}|$$

La macchina $\mathcal{M}$ conosce qualcosa della stringa x che deve produrre in output: $\mathcal{U}(\mathcal{M}; y) = \mathcal{M}(y) = x$. Quindi

$$K_{\mathcal{U}}(x) \geq K_{\mathcal{U}}(x|y)$$

La complessità di Kolmogorov condizionale è la lunghezza della macchina di Turing più corta che produce in output x se si conosce y. La conoscenza di y dà informazioni su x, e quindi aiuta a costruire una macchina $\mathcal{M}$ più corta.

Esempio Sia $000\ldots0$ una sequenza di m 0, che vogliamo produrre in output. Se non conosciamo m, si ha qualcosa del tipo

$$m \begin{cases} \texttt{print } 0 \\ \texttt{print } 0 \\ \ldots \\ \texttt{print } 0 \end{cases}$$

Ma, dato m, si ha

```
for (i = 1 to m){
  print 0
}
```

In questo esempio abbiamo quindi $K_{\mathcal{U}}(x||x|)$.

Cosa succederebbe se cambiassimo da $\mathcal{U}$ ad un'altra macchina di Turing universale $\mathcal{A}$?

Teorema 4.2.1. Siano $\mathcal{U}$ e $\mathcal{A}$ due macchine di Turing universali. Consideriamo $K_{\mathcal{U}}(x|y)$ e $K_{\mathcal{A}}(x|y)$. Allora

$$K_{\mathcal{U}}(x|y) \leq K_{\mathcal{A}}(x|y) + c_{\mathcal{A}\mathcal{U}}$$

con $c_{\mathcal{A}\mathcal{U}}$ costante che dipende solo da $\mathcal{A}$ e $\mathcal{U}$.

Dimostrazione Sia $P_{\mathcal{A}}$ il programma più corto per $\mathcal{A}$ che produce x dato y:

$$K_{\mathcal{A}}(x|y) = |P_{\mathcal{A}}|$$

Questo significa

$$\mathcal{A}(P_{\mathcal{A}}; y) = P_{\mathcal{A}}(y) = x$$

Sia $\mathcal{U}$ una macchina di Turing universale. Allora $\mathcal{U}$ con in input $\mathcal{C}_{\mathcal{A}}$ (codifica di $\mathcal{A}$) seguito da $P_{\mathcal{A}}$ e y produce in output x:

$$\mathcal{U}(\underbrace{\mathcal{C}_{\mathcal{A}}; P_{\mathcal{A}};}_{\substack{\text{programma per } \mathcal{U} \text{ che,} \\ \text{dato } y, \\ \text{produce } x}} y) = \mathcal{A}(P_{\mathcal{A}}; y) = P_{\mathcal{A}}(y) = x$$

Quindi

$$\begin{aligned} K_{\mathcal{U}}(x|y) &\leq |\mathcal{C}_{\mathcal{A}}; P_{\mathcal{A}}| \\ &= |\mathcal{C}_{\mathcal{A}}| + |P_{\mathcal{A}}| + 1 \\ &= K_{\mathcal{A}}(x|y) + \underbrace{|\mathcal{C}_{\mathcal{A}}| + 1}_{c_{\mathcal{A}\mathcal{U}}} \end{aligned}$$

□

Corollario 4.2.1.1.

$$|K_{\mathcal{U}}(x|y) - K_{\mathcal{A}}(x|y)| \leq c_{\mathcal{A}\mathcal{U}} \qquad \forall x, y$$

Quando $y = \varepsilon$

$$|K_{\mathcal{U}}(x) - K_{\mathcal{A}}(x)| \leq c_{\mathcal{A}\mathcal{U}} \qquad \forall x$$

D'ora in poi, quindi, sapendo che cambia solo una costrante, si ometterà la macchina di Turing universale, e si scriverà semplicemente $K(x)$, o $K(x|y)$.

Esempio Vogliamo produrre in output la stringa $x = 010$, e abbiamo il programma P

```
print 0
print 1
print 0
```

Codificandolo in binario, si ha, ad esempio, $\text{bin}(P) = 300$. Si può concludere che la complessità di Kolmogorov di x è al massimo 300:

$$K(010) \leq 300$$

Per poter scrivere $=$ invece che $\leq$, bisognerebbe dimostrare che tutti i programmi di lunghezza 299 non producono in output la stringa 010. Ma almeno uno di essi potrebbe non terminare, e è impossibile dimostrarlo.

Dall'esempio precedente abbiamo visto come non sia possibile calcolare la complessità di Kolmogorov di una stringa. Tuttavia, possiamo calcolare la complessità di Kolmogorov di una stringa con una certa precisione dando dei *bounds*.

Teorema 4.2.2 (Teorema sui Limiti alla Complessità di Kolmogorov sulle Stringhe)**.**

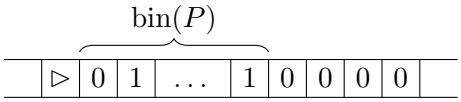
$$K(x) \leq |x| + c$$

Nel libro di T. Cover viene utilizzato un alfabeto con solo due simboli, $\Sigma = \{0, 1\}$. Questo dà un risultato diverso (più debole) rispetto a quello che si raggiunge utilizzando l'alfabeto $\Sigma = \{0, 1, \sqcup\}$.

Dimostrazione (intuizione) Il programma che produce in output $x = x_1 x_2 \dots x_n$ è della forma

$$\text{bin}(P) \begin{cases} \texttt{print } x_1 \\ \texttt{print } x_2 \\ \dots \\ \texttt{print } x_n \end{cases}$$

con $\text{bin}(P)$ della forma $010011\dots11$. Dando $\text{bin}(P)$ in input ad una macchina di Turing universale $\mathcal{U}$ utilizzando l'alfabeto $\Sigma = \{0, 1\}$, si avrà



C'è quindi un problema: utilizzando 0 invece di $\sqcup$, come si può capire dove finisce $\text{bin}(P)$? Una possibile soluzione è quella di raddoppiare ogni bit, e codificare lo stop con una coppia "non valida", ad esempio 10. Ad esempio, se $\text{bin}(P) = 010011$, si avrà

$$0\,0\,1\,1\,0\,0\,0\,0\,1\,1\,1\,1\underbrace{1\,0}_{\text{stop}}$$

Utilizzando questo metodo, quando si vuole x in output si può settare il primo bit (dopo $\triangleright$) a 0 o a 1, dando due significati diversi alla stringa che li segue. In particolare, se la prima cifra è

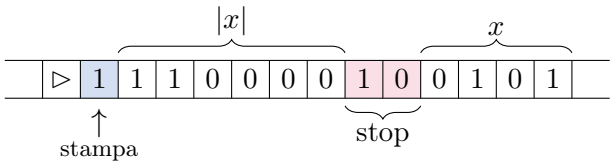
- 0, segue un generico programma P (ovvero 0 = esegui)
- 1, segue la codifica di x con il raddoppio dei bit (ovvero 1 = stampa)

Così facendo, con l'alfabeto $\Sigma = \{0, 1\}$, si ha

$$K(x) \leq 2|x| + 3$$

con 3 pari alla somma del primo bit (0 o 1) e della coppia "non valida" (10) che indica lo stop.

Analizziamo un'altra soluzione con un esempio. Sia $x = 0101$, e $|x| = 4$. Codifichiamo la lunghezza di x in binario come $|x| = 4 = 100$, e raddoppiamo i bit, ottenendo 110000. Quindi si avrà



In questo caso

$$K(x) \leq 2\log|x| + 2 + 1 + |x|$$

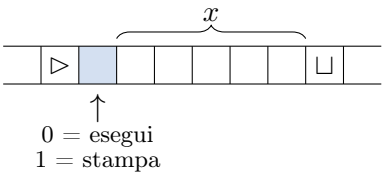
Quando x è molto lunga, $2\log|x|$ è molto più piccolo di $|x|$, e quindi si può trascurare. Quindi questa proposta è più corta (migliore) della precedente.

$$K(x) \leq 2\log|x| + 2 + 1 + |x| \leq 2|x| + 3$$

Un'ulteriore soluzione è quella di utilizzare il trucco precedente sulla lunghezza della lunghezza di x, e così via. In generale, si ha

$$K(x) \leq \cdots \leq \cdots \leq 2\log^*|x| + |x| + c$$

Ma ciò che il teorema afferma è più forte. Utilizzando l'alfabeto $\Sigma = \{0, 1, \sqcup\}$, si ha



Quindi

$$K(x) \leq |x| + 1$$

con $1 = c$ nel caso generico. □

Quante stringhe possono avere complessità di Kolmogorov minore di un dato valore k? Per stringhe di lunghezza 0 (stringa vuota) esiste 1 macchina di Turing (di lunghezza 0) che la produce in output. Per stringhe di lunghezza 1, esistono 2 macchine di Turing: quella la cui codifica è 0, e quella la cui codifica è 1, e così via.

lunghezza stringa	numero di macchine
0	0
1	2
⋮	⋮
h	2^h
⋮	⋮
$k-1$	2^{k-1}

In generale

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

ovvero al massimo $2^k - 1$ stringhe possono avere complessità di Kolmogorov minore di k.

Teorema 4.2.3.

$$\exists x \qquad K(x) \geq |x|$$

Dimostrazione Sia $|x| = k$. Allora ci sono 2^k stringhe di lunghezza k. Ma esistono al massimo $2^k - 1$ stringhe di lunghezza minore di k. Quindi esiste almeno una stringa di lunghezza k che ha complessità di Kolmogorov almeno pari a k. □

Abbiamo trovato un limite superiore e inferiore alla complessità di Kolmogorov di una stringa.

$$|x| \leq K(x) \leq |x| + c$$

4.2.1 Complessità di Kolmogorov vs Entropia di Shannon

Da Complessità a Entropia Consideriamo

$$\varphi_K(x) = \text{bin}(\mathcal{M})$$

con $\varphi_K(x)$ codifica di Kolmogorov di x, e $\mathcal{M}$ la più corta macchina di Turing che produce x in output (con input vuoto). $\varphi_K(x)$ è UD. $\mathcal{U}$ decodifica $\varphi_K(x)$, ovvero invio un programma, io e il ricevente abbiamo la stessa macchina di Turing universale (il "compilatore"), e il ricevente riceve in programma.

$$|\varphi_K(x)| = K(x)$$

Consideriamo tutti i possibili messaggi di lunghezza n

$$EL_n(\varphi_K(x)) = \sum_{x \in \Sigma^n} p(x)K(x) = E_n(K(x))$$

con $E_n(K(x))$ complessità di Kolmogorov media su tutti i messaggi di lunghezza n. Per il Teorema di Shannon

$$EL_n(\varphi_K(x)) \geq \mathcal{H}(P^n) = n\mathcal{H}(P)$$

con P distribuzione di probabilità sull'alfabeto Σ. Quindi

$$E_n(K(x)) \geq n\mathcal{H}(P)$$

La complessità di Kolmogorov media di un singolo carattere è

$$\frac{E_n(K(x))}{n} \geq \mathcal{H}(P)$$

Da Entropia a Complessità Ogni codice $\varphi(x)$ può essere interpretato come un particolare programma che produce in output la stringa (codificata) x. Quindi

$$K(x) \leq |\varphi(x)| + \text{Decoder } \varphi$$

Per lo Shannon code abbiamo visto che

$$EL_n(\varphi) \leq n\mathcal{H}(P) + 1$$

Ciò significa che

$$E_n(K(x)) \leq n\mathcal{H}(P) + 1 + K(P)$$

con $K(P) = |\text{Decoder } \varphi|$ complessità di Kolmogorov del Decoder (posso inviare il codice più corto possibile che produce in output il Decoder). Quindi

$$\frac{E_n(K(x))}{n} \leq \mathcal{H}(P) + \underbrace{\frac{1 + K(P)}{n}}_{\substack{\text{perché} \\ \text{consideriamo} \\ \text{lo Shannon} \\ \text{code}}}$$

Parte II

Complessità

Capitolo 5

Introduzione

Libro di Papadimitriou main reference.

In questa parte utilizzeremo come modello di computazione le **macchine di Turing** (MdT). Esistono diversi modelli di MdT: macchine di Turing multinastro, macchine di Turing input/output, macchine di Turing con oracolo, macchine di Turing nondeterministiche. Le MdT verranno utilizzate per confrontare i diversi risultati di complessità che possiamo ottenere.

Ci concentreremo sia su **complessità temporale** (time complexity) che **spaziale** (space complexity). Il focus non sarà sulla complessità di un dato algoritmo, ma sulla complessità di un problema. I **problemi** possono essere classificati come di decisione (decision problems), di funzione (function problems), o di ottimizzazione (optimization problems).

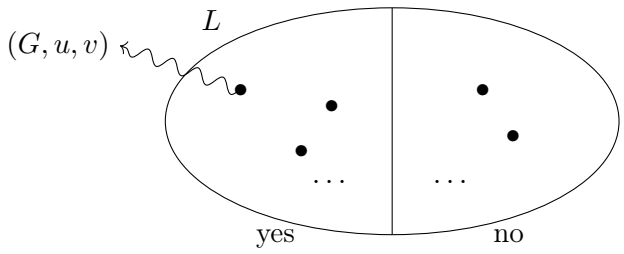
- **Decision problem** P : inputs $\to \{\text{yes}, \text{no}\}$
- **Function problem** computare una data funzione, ad esempio l'ordinamento di una lista
- **Optimization problem** tra tutti i possibili output, si vuole trovare quello che minimizza o massimizza una funzione di costo.

Esempio Sia $G = (V, E)$ un grafo, e $u, v \in V$ due nodi.

- decidere se esiste un cammino da u a v è un problema di decisione
- trovare un cammino da u a v è un problema di funzione
- trovare il cammino più corto da u a v è un problema di ottimizzazione

In questo corso ci concentreremo sui problemi di decisione. Se si ha una soluzione per un problema di funzione o di ottimizzazione, si possiede automaticamente una soluzione per il problema di decisione.

Immaginiamo tutti gli input possibili al problema dell'esempio precedente come ad un insieme infinito di tuple (G, u, v). Questo insieme si può dividere in due: il sottoinsieme dei yes di tutte le codifiche binarie di triple (G, u, v) tali che esiste un cammino in G da u a v, e, inversamente, il sottoinsieme no.



La codifica binaria di una tripla è una stringa del tipo $1011\ldots$. Più precisamente, è una stringa sull'alfabeto $\Sigma = \{0, 1\}$. L'insieme di tutte le possibili stringhe binarie è Σ^*. Questo insieme è quindi il linguaggio L sottoinsieme di Σ^*, ovvero $L \subseteq \Sigma^*$.

$$L = \{\text{bin}(G, u, v) | ln\ G\ u \to v\}$$

Esempio Consideriamo interi rappresentati in binario. Vogliamo decidere se un dato intero x è divisibile per 4.

$$\text{bin}(x) = 10\ldots11$$

in questo caso non è divisibile per 4. Un numero binario è divisibile per 4 se e solo se i due bit meno significativi sono 0.

$$\text{bin}(x) = x_n, x_{n-1}, \ldots, x_1, x_0 \quad \Leftrightarrow \quad x_0 = 0 \text{ and } x_1 = 0$$

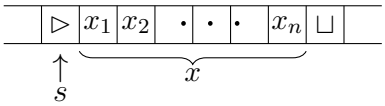
Il linguaggio indicato da questo problema di decisione è

$$L = \{x \in \{0,1\}^* | x = x_n, x_{n-1}, \ldots, x_1, x_0 \wedge x_0 = 0 \wedge x_1 = 0\}$$

Esempio: palindromo Decidere se una stringa è palindroma, con $\Sigma = \{0, 1\}$.

$$x_1, x_2, x_3, \ldots, x_3, x_2, x_1$$

Ad esempio, $x = 101$ è palindroma, mentre $x = 1010$ non lo è. Cerchiamo il linguaggio $L = \{x | x \text{ è palindroma}\}$. Utilizziamo una macchina di Turing.



Si parte dallo stato s e si vuole finire nello stato p solo quando x è palindroma. Per decidere se x è palindroma, si può leggere x_1, ricordarne il valore nello stato del puntatore, e poi confrontarlo con x_n. Se sono uguali, si ripete lo stesso procedimento con x_2 e x_{n-1}, e così via. Se si arriva a x_n e x_1 senza aver trovato una discrepanza, allora x è palindroma. Se invece si trova una discrepanza, allora x non è palindroma. Le transizioni sono le seguenti:

$$\begin{aligned}\delta(s, \triangleright) &= (q, \triangleright, \rightarrow)\\ \delta(q, 1) &= (q_1, \triangleright, \rightarrow)\\ \delta(q, 0) &= (q_0, \triangleright, \rightarrow)\end{aligned}$$

TODO: finire di scrivere le transizioni Questa macchina eseguirà un numero quadratico di passi per controllare se la stringa x è palindroma: $O(|x|^2)$.

Se si vuole controllare in C (o in un altro linguaggio) se una stringa è palindroma, si può scrivere un programma che confronta il primo e l'ultimo carattere, poi il secondo e il penultimo, e così via, eseguendo un numero lineare di passi. La complessità è $O(|x|)$. Questo è un esempio di come la complessità di un problema dipenda dal modello di computazione utilizzato.

5.1 Tesi di Church-Turing Estesa

La tesi di Church-Turing afferma che ogni cosa che può essere computata, può essere computata da una macchina di Turing.

La versione estesa afferma che tutti i modelli (ragionevoli) di calcolo sono correlati polinomialmente. Questo significa che se un problema è risolvibile in tempo polinomiale in un modello di computazione, allora è risolvibile in tempo polinomiale in ogni modello di computazione.

In altre parole, la tesi di Church-Turing estesa afferma che la complessità computazionale di un problema è indipendente dal modello di calcolo utilizzato per risolverlo.

$$\underset{\text{problema}}{P} \rightarrow \text{MdT } O(f(n)) \rightarrow \underset{\text{modello}}{M} \; O(p(f(n)))$$

Ma è vera anche la direzione contraria. TODO: ???

Capitolo 6

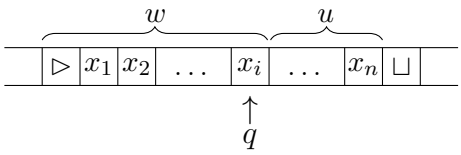
Macchine di Turing

6.1 Definizioni

Definizione 6.1.1 (Configurazione). Una configurazione è una tripla (q, w, u), con

- $q \in K \cup \{\text{yes}, \text{no}, \text{halt}\}$
- $w, u \in \Sigma^*$

Ad esempio, graficamente, una configurazione è



Definizione 6.1.2 (Configurazione Iniziale). La configurazione iniziale su una stringa x è una tripla

$$(1, \triangleright, x)$$

Definizione 6.1.3 (Configurazioni Finali). Le configurazioni finali su una stringa x sono una tripla

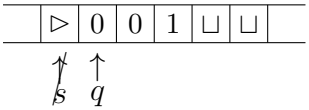
$$(H, w, u)$$

dove $H \in \{\text{yes}, \text{no}, \text{halt}\}$.

Definizione 6.1.4 (Passo di Computazione).

$$(q, w, u) \xrightarrow{\delta} (q', w', u')$$

Ad esempio, il passo di computazione è $(s, \triangleright, 001) \to (q, \triangleright 0, 01)$



Eseguito applicando $\delta(s, \triangleright) = (q, \triangleright, \to)$.

Definizione 6.1.5 (Time Complexity per una MdT $\mathcal{M}$ sull'input x). $\mathcal{M}$ ha time complexity t su x se dopo esattamente t passi si raggiunge una configurazione finale.

$$(s, \triangleright, x) \underbrace{\to \cdots \to}_{t \text{ passi}} (H, w, u)$$

Indicata in breve con $(s, \triangleright, x) \to^t (H, w, u)$.

$\mathcal{M}$ ha time complexity $f : \mathbb{N} \to \mathbb{N}$ se, $\forall x \in \Sigma^*$, $(s, \triangleright, x) \to^t (H, w, u)$ con $t \leq f(|x|)$.

La dimensione dell'input (bit length dell'input) è $|x|$. Questa è una complessità nel caso peggiore ($\leq$). Non stiamo utilizzando la notazione big-O.

6.2 Unlimited Register Machines

Una Unlimited Register Machine (URM) è una macchina di Turing con un numero illimitato di registri.

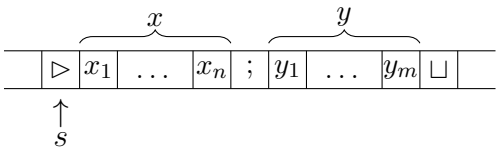
R_0	r_0
R_1	r_1
	$\dots$
R_m	r_m
	$\dots$

Ogni registro contiene un numero naturale. Quindi, il contenuto del registro R_m sarà $r_m \in \mathbb{N}$. Le operazioni possibili sono:

- **incremento** $S(i)$: $r_i := r_i + 1$
- **azzeramento** $Z(i)$: $r_i := 0$
- **trasferimento** $T(i, j)$: $r_j := r_i$, ovvero trasferisco il contenuto del registro R_i nel registro R_j
- **jump** $J(i, j, k)$: se $r_i = r_j$ allora salta all'istruzione k, altrimenti prosegue con l'istruzione successiva

Esempio Dati $x, y \in \mathbb{N}$, decidere se $x = y$.

MdT Si può utilizzare una macchina di Turing che contiene la rappresentazione binaria dei due interi, separati da un separatore.



Questa macchina richiede, nel caso peggiore, un numero quadratico di passi per terminare. La complessità è $\Theta(|x|^2)$.

URM Possiamo utilizzare una URM con x e y rispettivamente nei registri R_0 e R_1.

R_0	x
R_1	y

Alla fine, scriveremo 1 in R_0 se $x = y$, 0 altrimenti. Le istruzioni sono le seguenti:

1. $J(0,1,4)$
2. $Z(0)$
3. $J(0,0,100)$
4. $Z(0)$
5. $S(0)$

In questo caso, la complessità si può calcolare in due modi.

Definizione 6.2.1 (Time Complexity su URM)**.**

- *Uniform cost criterium* (criterio del costo uniforme): numero di istruzioni eseguite.
- *Logarithmic cost criterium* (criterio del costo logaritmico): ogni istruzione ha un costo proporzionale al numero di cifre coinvolte.

Quindi, per questa macchina, la complessità è

- utilizzando il criterio del costo uniforme: $\Theta(1)$
- utilizzando il criterio del costo logaritmico: $\Theta(|x| + |y|)$

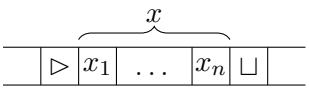
Nel secondo caso, ci si avvicina al costo per la macchina di Turing.

Mentre le macchine di Turing sono un modello di computazione sequenziale, nelle URM si ha l'istruzione *jump*. In altre parole:

- **MdT** 1 bit di informazione in ogni cella $\rightarrow$ tempo: numero di passi
- **URM** registri, un intero di lunghezza arbitraria (più bit) in ogni registro $\rightarrow$ tempo: numero di istruzioni (uniform time complexity)

Esempio Computare $x + 1$, $x \in \mathbb{N}$.

MdT Si ha una macchina di Turing che contiene x in binario.



Nel caso peggiore $x = 111\ldots1$, quindi la complessità è lineare $\Theta(n)$.

URM Si ha una URM con x nel registro R_0. È suffieciente una singola istruzione $S(0)$, quindi la complessità è $\Theta(1)$.

6.2.1 URM + Prodotto

Cambiamo il modello di computazione URM, considerando URM + prodotto. Oltre alle istruzioni $S(i)$, $Z(i)$, $T(i,j)$, e $J(i,j,k)$, aggiungiamo l'istruzione $P(i)$, che esegue l'operazione $r_i := r_i * r_i$.

Esempio di programma per URM + prodotto Abbiamo in input un numero x, che copiamo anche in R_1. Applichiamo il prodotto sul contenuto del registro R_0 per x volte. In altre parole, vogliamo calcolare x^{2^x}.

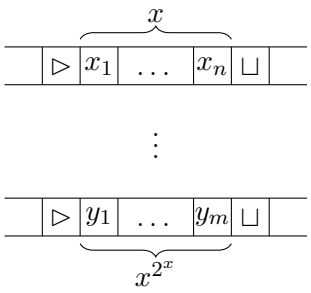
1. $J(1,2,5)$
2. $P(0)$
3. $S(2)$
4. $J(3,3,1)$

Pertendo da un input di x in R_0, x in R_1, e 0 in tutti gli altri registri. In un generico passo di iterazione i si avrà:

R_0	x		R_0	x^2		R_0	$(x^2)^2$		R_0	$(x^4)^2$					R_0	x^{2^i}
R_1	x		R_1	x		R_1	x		R_1	x					R_1	x
R_2	0	$\rightarrow$	R_2	1	$\rightarrow$	R_2	2	$\rightarrow$	R_2	3	$\rightarrow$	$\ldots$	$\rightarrow$	R_2	i
R_3	0		R_3	0		R_3	0		R_3	0					R_3	0
R_4	$\vdots$		R_4	$\vdots$		R_4	$\vdots$		R_4	$\vdots$					R_4	$\vdots$

Il numero di istruzioni è lineare $\Theta(n)$.

MdT Se si eseguisse la stessa computazione su una macchina di Turing, si avrebbe



Quindi $\Omega(\log(x^{2^x})) = \Omega(2^x \log(x))$.

Questo risultato sembra contraddire la tesi di Church-Turing estesa, che afferma che tutti i modelli **ragionevoli** di computazione sono correlati polinomialmente. Ma cosa significa *ragionevole*? Non si può avere una operazione che fa crescere "troppo" l'input (nell'esempio, il prodotto), si deve utilizzare il criterio logaritmico.

In altre parole, se l'algoritmo utilizza operazioni che in un numero polinomiale di passi fanno crescere l'input esponenzialmente, e queste sono utilizzate un numero di volte che dipende dalla dimensione dell'input, allora si deve utilizzare un criterio logaritmico. Quando non si è sicuri della potenza delle operazioni della macchina, il costo di ogni singola operazione dev'essere proporzionale al numero di bit manipolati.

istruzione	uniform	logarithmic
$S(i)$	$\Theta(1)$	$\Theta(\log(r_i))$
$Z(i)$	$\Theta(1)$	$\Theta(1)$
$T(i,j)$	$\Theta(1)$	$\Theta(\log(r_i))$
$J(i,j,k)$	$\Theta(1)$	$\Theta(\min(\log(r_i), \log(r_j)))$
$P(i)$	$\Theta(1)$	$\Theta((\log(r_i))^2)$

Con r_i contenuto del registro i. In particolare per $P(i)$, nella moltiplicazione di un numero x per se stesso si ha $x_1, x_2, \ldots, x_n \times x_1, x_2, \ldots, x_n$. Si hanno x^n bit operazioni, quindi $O((\log(x))^2)$.

6.3 Ulteriori Definizioni

Come abbiamo visto, nei problemi di decisione si ha un input $x \in \Sigma^*$ e un output in $\{\text{yes}, \text{no}\}$. Possiamo definire un linguaggio L come l'insieme di tutte le stringhe che hanno output yes.

$$L \subseteq (\Sigma \backslash \{\sqcup\})^*$$

Un problema P è una funzione

$$P : \Sigma^* \to \{\text{yes}, \text{no}\}$$

Definizione 6.3.1 (Linguaggio Ricorsivo)**.**

$$\text{Una macchina di Turing } \mathcal{M} \text{ decide un linguaggio } L$$
$$\Updownarrow$$
$$\forall x \in (\Sigma \backslash \{\sqcup\})^* \begin{cases} x \in L \to \mathcal{M}(x) = \text{yes} \\ x \notin L \to \mathcal{M}(x) = \text{no} \end{cases}$$

Il linguaggio L si dice **ricorsivo**.

Definizione 6.3.2 (Linguaggio Ricorsivamente Enumerabile)**.**

$$\text{Una macchina di Turing } \mathcal{M} \text{ accetta un linguaggio } L$$
$$\Updownarrow$$
$$\forall x \in (\Sigma \backslash \{\sqcup\})^* \begin{cases} x \in L \to \mathcal{M}(x) = \text{yes} \\ x \notin L \to \mathcal{M}(x) \uparrow \text{ (non termina)} \end{cases}$$

Il linguaggio L si dice **ricorsivamente enumerabile**.

Teorema 6.3.1.

$$L \text{ è ricorsivo } \Rightarrow \ L \text{ è ricorsivamente enumerabile}$$

Esempio Trovare un linguaggio L tale che L è ricorsivamente enumerabile ma non ricorsivo.
Nell'halting problem abbiamo

$$\mathcal{U}(\mathcal{M}; x) = \mathcal{M}(x)$$

L'halting language

$$H = \{(\text{bin}(\mathcal{M}); x) \mid \mathcal{M}(x) \downarrow\}$$

è ricorsivamente enumerabile ma non ricorsivo. Infatti, se $\mathcal{M}$ termina su x, allora $\mathcal{U}(\mathcal{M}; x) = \mathcal{M}(x) =$ yes, altrimenti $\mathcal{U}(\mathcal{M}; x) \uparrow$. Questo è un risultato qualitativo.

Esempio Sia

$$L = \{\text{bin}(\mathcal{M}) \mid \forall x \ \mathcal{M}(x) \downarrow \text{ in al massimo 100 passi}\}$$

L è ricorsivo. Infatti, la macchina $\mathcal{M}$ può eseguire al massimo 100 spostamenti a destra sul nastro. Quindi, tutte le macchine che terminano in al massimo 100 passi accettano input $\forall x \in |\Sigma|^n$ con $n \leq 100$.

Definizione 6.3.3 (Computazione di Funzioni)**.** Sia f una funzione $f : (\Sigma \backslash \sqcup)^* \to \Sigma^*$. Una macchina di Turing $\mathcal{M}$ computa f se

$$\forall x \in (\Sigma \backslash \sqcup)^* \qquad \mathcal{M}(x) \downarrow \text{ e alla fine } f(x) \text{ è sul nastro}$$

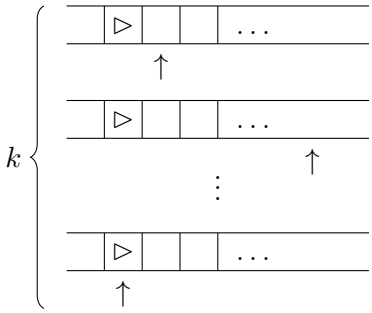
La funzione f è detta **ricorsiva**, o **computabile**.

6.4 Macchine di Turing a k-nastri e Input/Output

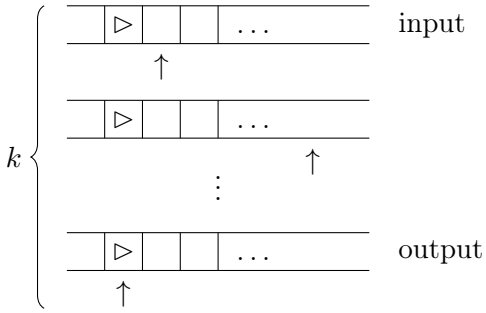
Definizione 6.4.1 (Macchina di Turing a k-nastri)**.** Una macchina di Turing a k-nastri è una tupla $\mathcal{M} = (K, \Sigma, \delta, s)$ con K, Σ, s definite come per una macchina di Turing, e

$$\delta : K \times \Sigma \to (K \cup \{\text{yes}, \text{no}, \text{halt}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$$

Una macchina di Turing a k-nastri è una macchina di Turing con un numero limitato di nastri, che possono essere utilizzati in parallelo. La funzione δ cambia perché si ha un puntatore per nastro.



Definizione 6.4.2 (Macchina di Turing a k-nastri con Input/Output)**.** Una macchina di Turing a k-nastri con I/O è una macchina di Turing a k-nastri con un nastro di input e un nastro di output. Il nastro di input è di sola lettura, il nastro di output è di sola scrittura.



Definizione 6.4.3 (Configurazione e Configurazione Iniziale)**.** Siano $w_i, u_i \in \Sigma^*$ stringhe. Una configurazione è una tupla

$$(q, w_1, u_1, w_2, u_2, \ldots, w_k, u_k) \to (q', w_1', u_1', w_2', u_2', \ldots, w_k', u_k')$$

Una configurazione iniziale su input x è una tupla

$$(s, \rhd, x, \rhd, \varepsilon, \ldots, \rhd, \varepsilon)$$

6.4.1 Complessità Temporale

Ricordiamo che L è deciso dalla macchina a k-nastri $\mathcal{M}$ se, $\forall x$,

$$\begin{aligned} x \in L &\to \mathcal{M}(x) = \text{yes} \\ x \notin L &\to \mathcal{M}(x) = \text{no} \end{aligned}$$

Inoltre,

$$\underbrace{(s, \triangleright, x, \triangleright, x, \dots)}_{\text{conf. iniziale}} \to^{\mathcal{M}_t} \underbrace{(H, \dots)}_{\text{conf. finale}} \qquad H \in \{\text{yes}, \text{no}\}, \quad \forall x \;\; t \leq f(|x|)$$

$\mathcal{M}$ indica L in tempo al massimo $f(n)$.

Esempio: palindromo Consideriamo $\mathcal{M}_1$ macchina di Turing a singolo nastro vs $\mathcal{M}_2$ macchina di Turing a k-nastri. Sia L il linguaggio di palindromi su $\{0, 1\}$. Le complessità temporali sono

- $\mathcal{M}_1$: $\Theta(n^2)$
- $\mathcal{M}_2$: $\Theta(n)$

Vedremo come il risultato generale indica che questo è il peggior incremento che si può ottenere.

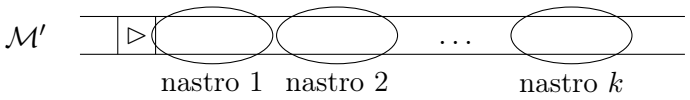
Teorema 6.4.1.

L è deciso da una macchina di Turing a k-nastri $\mathcal{M}$ in tempo $f(n)$

$\Downarrow$

$\exists$ una macchina di Turing a 1-nastro $\mathcal{M}'$ che decide L in tempo (al massimo) $O(f(n)^2)$

Dimostrazione Sia $\mathcal{M}$ macchina di Turing a k-nastri che decide L in tempo $f(n)$. Vogliamo costruire una macchina di Turing a 1-nastro $\mathcal{M}'$ in grado di simularla. Si possono rappresentare le informazioni dei k-nastri su un singolo nastro in questo modo:



Per imitare i vari cursori, si può ingrandire l'alfabeto:

$$\Sigma \cup \underline{\Sigma} \cup \{\triangleright'\} \qquad \underline{\Sigma} = \{\underline{\sigma} | \sigma \in \Sigma\}$$

In ogni momento la macchina $\mathcal{M}$ legge k simboli in $\mathcal{M}'$, bisogna leggere tutto il nastro per leggere (e memorizzare) il valore di tutti i k simboli, e poi un ulteriore scansione per cambiarli. In altre parole, per simulare un passo di $\mathcal{M}$ su $\mathcal{M}'$:

- Scansionare il nastro di $\mathcal{M}'$ per leggere i $\underline{\sigma}$ e memorizzarli nello stato di $\mathcal{M}'$
- Tornare indietro
- Cambiare i $\underline{\sigma}$
  - Ad esempio, se nel nastro in $\mathcal{M}$ cambio un simbolo e mi muovo a destra ($\to$), in $\mathcal{M}'$ passerò da $\boxed{}\boxed{\underline{\sigma}}\boxed{a}\boxed{}$ a $\boxed{}\boxed{\sigma'}\boxed{\underline{b}}\boxed{}$

Nel caso peggiore, ognuno dei k cambiamenti richiede di spostare tutto a destra per creare spazio sul nastro di $\mathcal{M}'$.

La domanda ora è: quanto è lungo al massimo il nastro di $\mathcal{M}'$ durante la computazione? Per $k-1$ nastri si ha un'occupazione che è al massimo $f(n)$, e per il primo nastro $n + f(n)$, con n lunghezza dell'input. Quindi, con l'ipotesi che $f(n) \geq n$, si ha

$$(k-1) \cdot O(f(n)) + O(n + f(n)) = kO(f(n)) = O(f(n))$$

Si può concludere che simulare un passo di $\mathcal{M}$ richiede tempo $O(f(n))$. Poiché ci vogliono $O(f(n))$ passi di $\mathcal{M}'$ per simulare un passo di $\mathcal{M}$, si ha che $\mathcal{M}'$ decide L in tempo $O(f(n)^2)$. □

A seguito di questo risultato, d'ora in poi si farà affidamento solo sulla complessità delle macchine di Turing a k-nastri.

Definizione 6.4.4 (Appartenenza ad una Classe di Complessità Temporale)**.**

$$L \in \text{TIME}(f(n)) \ \exists \text{ MdT a } k\text{-nastri } \mathcal{M} \text{ che decide } L \text{ in tempo } f(n)$$

Ad esempio, Palindromi $\in$ TIME$(3n+4)$. Questo insieme di linguaggi potrebbe essere diverso da, ad esempio, TIME$(2n+1)$. Non utilizziamo la notazione asintotica.

Teorema 6.4.2 (Speed-Up Theorem per TIME)**.**

$$L \in \text{TIME}(f(n)) \ \Rightarrow \ \forall \varepsilon > 0 \quad L \in \text{TIME}(f'(n))$$

con $f'(n) = \varepsilon \cdot f(n) + n + 2$.

Bisogna sempre pagare il tempo lineare $n+2$ per leggere l'input. Questo risultato ci dice che possiamo utilizzare la notazione asintotica.

Dimostrazione Sia $\mathcal{M}$ macchina di Turing a k-nastri che decide L in tempo $f(n)$. Vogliamo costruire una macchina di Turing a k-nastri $\mathcal{M}'$ che simula m passi di $\mathcal{M}$ in "un singolo passo" (nel libro sono fissati 6 passi). Ad esempio, se $f(n) = 3n^2$ e $\varepsilon = \frac{1}{3}$, allora $f'(n) = n^2$.

Vogliamo codificare blocchi di m passi in un singolo passo: $\Sigma' = \Sigma^m$. Quando $\mathcal{M}'$ legge una cella, sta in realtà leggendo m simboli.

- Si copia il nastro di input da $\mathcal{M}$ a $\mathcal{M}'$, e lo si traduce da Σ a Σ': questo costa $n+1$.
- Per quanto riguarda il puntatore, sappiamo che in $\mathcal{M}$ gli stati sono in K, mentre in $\mathcal{M}'$ sono in $K \times \{1, \dots, m\}^k$. In $\mathcal{M}'$ devo sapere dove sta il puntatore, ma se in $\mathcal{M}$ viene letto qualcosa nei successivi m passi (si cambia quindi di cella in $\mathcal{M}'$), gli stati di $\mathcal{M}'$ sono in $K \times \{1, \dots, m\}^k \times (\Sigma^m)^2 k$.

I passi di $\mathcal{M}'$ sono quindi $n + 2 + f(n) \cdot \varepsilon$. □

Abbiamo cambiato architettura, ciò che si può memorizzare in una singola cella. Ad esempio, se costruiamo una macchina in cui $m = 18$, 18 passi di $\mathcal{M}$ vengono simulati da 6 passi di $\mathcal{M}'$. Quindi i passi di $\mathcal{M}'$ sono $n + 2 + f(n) \cdot \frac{6}{m}$, con $\frac{6}{m} = \varepsilon$.

6.4.2 Complessità Spaziale

Una configurazione finale di una macchina di Turing a k-nastri è del tipo $(H, w_1, u_1, \dots, w_k, u_k)$. Include anche tutte le parti raggiunte durante la computazione (anche se tutti i cursori alla fine puntano a $\sqcup$). Si può utilizzare come complessità spaziale la lunghezza della configurazione finale.

Definizione 6.4.5 (Complessità Spaziale per una MdT a k-nastri su input x)**.** Si ha che

$$(s, \triangleright, x) \to^* (H, w_1, u_1, \dots, w_k, u_k)$$

con $H = \{\text{halt}, \text{yes}, \text{no}\}$. Lo spazio utilizzato è

$$\sum_{i=1}^{k} |w_i| + |u_i|$$

Se non vogliamo contare la dimensione dell'input e dell'output, bisogna utilizzare le macchine di Turing a k-nastri con I/O. L'input può essere solo letto, e l'output solo scritto.

Definizione 6.4.6 (Complessità Spaziale per una MdT a k-nastri con I/O)**.** Si ha che

$$(s, \triangleright, x) \to^* (H, w_1, u_1, \ldots, w_k, u_k)$$

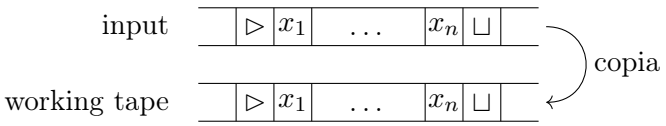
con $H = \{\text{halt}, \text{yes}, \text{no}\}$. Lo spazio utilizzato è

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

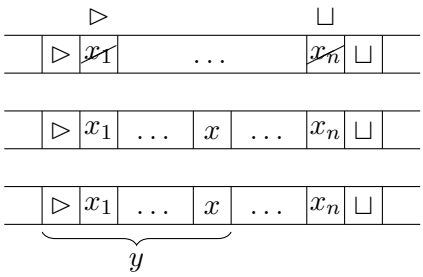
dove $\delta(q, \sigma_1, \ldots, \sigma_k) = (q', \sigma'_1, \ldots, \sigma'_k, \to)$.

Definizione 6.4.7 (Classi di Complessità Spaziale)**.** L è decidibile in spazio $f(n)$ se esiste una macchina di Turing a k-nastri con I/O $\mathcal{M}$ che decide L e, $\forall x$, $\mathcal{M}$ utilizza uno spazio al massimo $f(|x|)$.

Esempio: palindromo $L = \{x | x \text{ è palindroma}\}$. Si vuole trovare la macchina più efficiente in termini di spazio. La seguente macchina è efficiente nel tempo:



perché ha TIME $\Theta(n)$ e SPACE $\Theta(n)$. Mentre la seguente macchina è efficiente nello spazio:



con SPACE $\Theta(\log n)$.

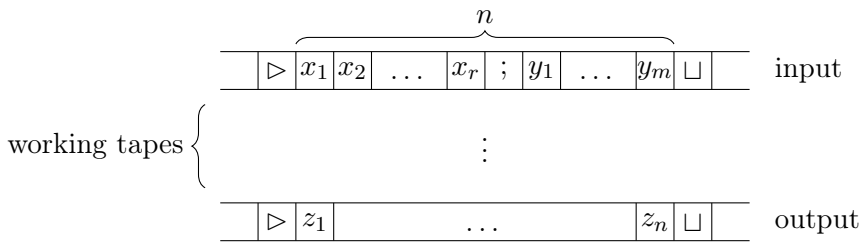
Definizione 6.4.8.

$$\text{TIME}(f(n)) = \{L \mid L \text{ può essere deciso in tempo } f(n)\}$$
$$\text{SPACE}(f(n)) = \{L \mid L \text{ può essere deciso in spazio } f(n)\}$$

In altre parole, SPACE$(f(n))$ è l'insieme di tutti i linguaggi che possono essere decisi in tempo $f(n)$ da una macchina di Turing a k-nastri con I/O. Per ogni input x tale che $|x| = n$, la macchina utilizza spazio al più $f(n)$.

Proprietà 6.4.1. Se esiste una macchina di Turing che decide L in tempo $f(n)$, e $f(n) \geq n$, allora esiste una macchina di Turing con I/O che decide L in tempo $O(f(n))$.

Esempio Calcola $x + y$.



Questo ha spazio lineare $\Theta(n)$ (molto male).

Definizione 6.4.9 (Classe P). Definiamo la classe P come

$$\mathrm{P} = \bigcup_{h \in \mathbb{N}} \mathrm{TIME}(n^h)$$

ovvero l'unione di tutti i problemi che possono essere risolti in tempo polinomiale.

La classe P ci piace così tanto perché abbiamo la tesi di Church-Turing estesa. Questa classe è **invariante** rispetto alla scelta del modello di computazione. Possiamo definire la classe EXP

$$\mathrm{EXP} = \bigcup_{h \in \mathbb{N}} \mathrm{TIME}(2^{n^h})$$

La classe $\mathbb{L}$, PSPACE, e EXPSPACE

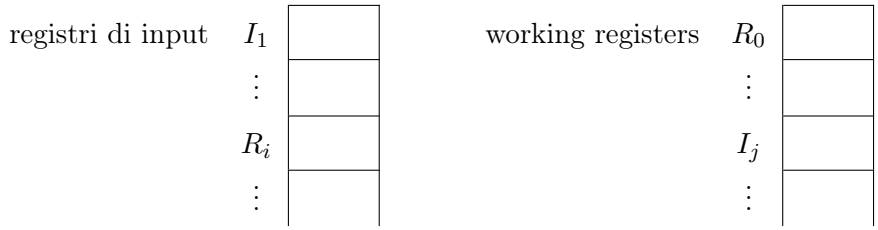
$$\begin{aligned} \mathbb{L} &= \mathrm{SPACE}(\log n) \\ \mathrm{PSPACE} &= \bigcup_{h \in \mathbb{N}} \mathrm{SPACE}(n^h) \\ \mathrm{EXPSPACE} &= \bigcup_{h \in \mathbb{N}} \mathrm{SPACE}(2^{n^h}) \end{aligned}$$

Proprietà 6.4.2.

$$\mathrm{TIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$$

6.5 Random Access Machines

Capitolo 2.6 del libro. Le random access machine (RAM), sono un modello di computazione sequenziale, composte da registri di input e registri di lavoro. Ogni registro contiene un intero.



Le operazioni possibili sono:

- `READ` j: $r_0 := i_j$
- `READ` $\uparrow$ j: $r_0 := i_{r_j}$ (vai al registro R_j, leggine il contenuto h, vai al registro I_h, copiane il contenuto in R_0)
- `STORE` j: $r_j := r_0$

- `STORE` $\uparrow j$
- `LOAD` j: $r_0 := r_j$
- `LOAD` $\uparrow j$
- `LOAD` $= j$: $r_0 := j$
- `ADD` j: $r_0 := r_0 + r_j$
- `ADD` $\uparrow j$: $r_0 := r_0 + r_{r_j}$
- `ADD` $= j$
- `SUB` j
- ...
- `HALF`: $r_0 := \left\lfloor \frac{r_0}{2} \right\rfloor$ (tolgo da r_0 l'ultimo bit)
- `JUMP` j: $k := j$ (contatore)
- `JPOS` j: if $r_0 > 0$ then $k := j$
- `JNEG` j
- `JZERO` j
- `HALT`

Il libro dimostra che

Teorema 6.5.1. RAM con complessità temporale uniforme e macchine di Turing con k-nastri sono correlate polinomialmente.

In particolare

$$\underbrace{\text{MdT}}_{f(n)} \to^{\text{simula}} \underbrace{\text{RAM}}_{O(f(n))}$$
$$\underbrace{\text{RAM}}_{f(n)} \to^{\text{simula}} \underbrace{\text{MdT a 7-nastri}}_{O((f(n))^3)}$$

Ad esempio, quando si sommano due numeri, si ottiene al massimo 1 bit in più dell'input maggiore.

6.6 Macchine Nondeterministiche

Si hanno

- Macchine deterministiche $\mathcal{M} = (K, \Sigma, \delta, s)$, con δ **funzione**

  $$\delta : K \times \Sigma^k \to (K \cup \{\text{yes}, \text{no}, \text{halt}\}) \times \Sigma^k \times \{\leftarrow, \rightarrow, -\}^k$$

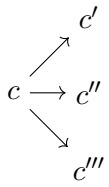
  la cui configurazione è del tipo

  $$c \to c'$$

- Macchine nondeterministiche $\mathcal{N} = (K, \Sigma, \Delta, s)$, con Δ **relazione**

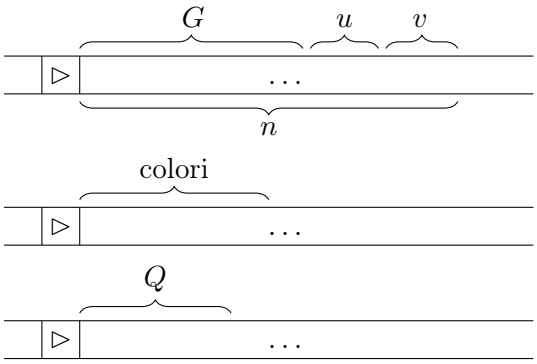
  $$\Delta \subseteq K \times \Sigma^k \times (K \cup \{\text{yes}, \text{no}, \text{halt}\}) \times \Sigma^k \times \{\leftarrow, \rightarrow, -\}^k$$

  quindi con una o più possibili transizioni. La configurazione $(q, u_1, w_1, \ldots, q_k, w_k)$ è del tipo



Esempio: Reachability Problem Dato un grafo diretto $G = (V, E)$, e due nodi $u, v \in V$, decidere se u raggiunge v (se esiste un cammino da u a v). Studiamo la complessità in tempo e spazio di questo problema utilizzando sia un modello deterministico che nondeterministico.

Modello deterministico Un possibile algoritmo per risolvere questo problema è BFS(G, u). Si costruisce un albero con radice u, e ad ogni livello si aggiungono i nodi raggiungibili in un passo. Utilizzando dei colori, alla fine della visita tutti i nodi visitati saranno neri, e quelli non raggiungibili grigi: è sufficiente controllare se v è nero. Se v è raggiungibile da u, allora v sarà raggiunto da u in un numero di passi $\leq |V|$. Quindi, la **complessità in tempo** è $O(|V| + |E|)$, ovvero lineare rispetto alla dimensione del grafo. Con una macchina di Turing:



con Q queue. Quindi, per la tesi di Church-Turing estesa, la complessità è $O(n^\alpha)$ per qualche $\alpha \in \mathbb{N}$. Si ha che

$$\text{Reachability} \in \text{P}$$

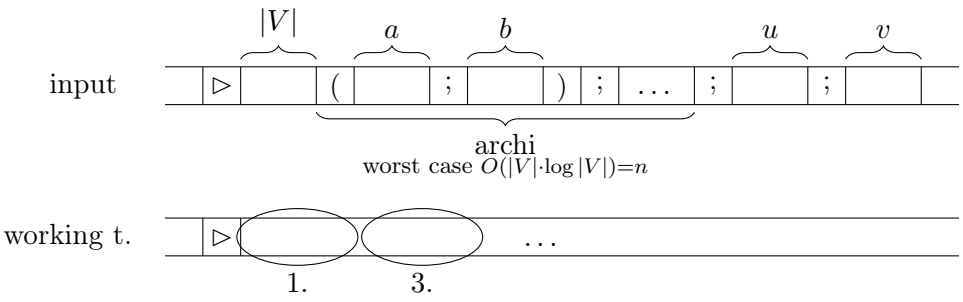
Per quanto riguarda la **complessità in spazio**, si ha che i colori e Q sono molto gradi, quindi

$$\text{Reachability} \in \text{PSPACE}$$

Esercizio: migliorare questo risultato. In particolare, Reachability $\in \mathbb{L} = \text{SPACE}(\log n)$? Reachability $\in \text{SPACE}((\log n)^2)$?

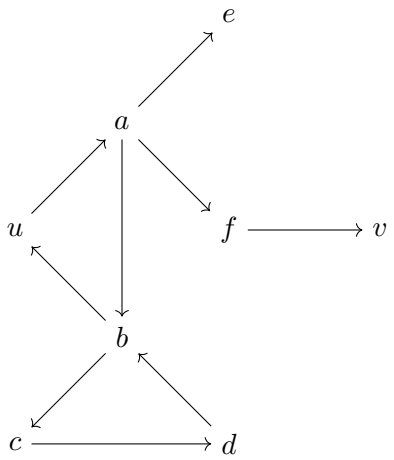
Modello nondeterministico Si hanno diversi possibili stati futuri. Immaginiamo un grafo con un cammino da u a v. Una macchina deterministica segue tutti i cammini uno ad uno, e per ognuno controlla se è quello corretto. Una macchina nondeterministica è in grado di "indovinare" il cammino corretto e di seguirlo.

Analizziamo la **complessità spaziale**. In questo caso non si ha bisogno dei colori. La macchina nondeterministica genera ad ogni passo un nuovo nodo nel working tape:



1. Genera il codice di un nodo (ad esempio, del nodo a)
2. Controlla se esiste un arcp da u ad a
   - Se non esiste, ritorna "no"
   - Se esiste, vai avanti
3. Aggiungi un altro nodo (ad esempio, il nodo b)
4. Controlla se esiste un arco da a ad b
   - Se non esiste, ritorna "no"
   - Se esiste, vai avanti
5. Sostituisci a con b, e aggiungi un altro nodo (ad esempio, il nodo c)
6. ...

Esempio Consideriamo il grafo



Proviamo a trovare un cammino da u a v:

- Esiste un arco da u ad a? Sì, quindi ua
- Esiste un arco da a a e? Sì, quindi uae
- Esiste un arco da e a c? No, quindi non esiste un cammino $uaec$

Esiste invece una computazione che genera il cammino $uafv$? Sì. Si può notare come ci sia però un problema, ovvero i cicli (ad esempio $uabuabcdbcdb\ldots$). In realtà, questo non è un problema: ragionando sulla complessità, e non sulla computabilità, consideriamo solo macchine di Turing che terminano.

Un modo per evitare i cicli è quello di immagazzinare in un working tape un contatore di passi eseguiti. Quando tale contatore raggiunge $|V| + 1$, si può fermare la computazione e ritornare "no".

$$\text{Reachability} \in \mathrm{N}\mathbb{L} = \mathrm{NSPACE}(\log n)$$

Definizione 6.6.1 (Macchina Nondeterministica). Una macchina nondeterministica è una tupla $\mathcal{N}(K, \Sigma, \Delta, s)$, dove

- K è un insieme finito di stati, di cui $s \in K$ è quello iniziale
- Σ è un alfabeto finito, e $\triangleright, \sqcup \in \Sigma$.

- Δ è la relazione di transizione, definita come

$$\Delta \subseteq K \times \Sigma^k \times (K \cup \{\text{yes}, \text{no}, \text{halt}\}) \times \Sigma^k \times \{\leftarrow, \rightarrow, -\}^k$$

La relazione di transizione tra due configurazioni è definita come per le macchine deterministiche:

$$(q, u_1, w_1, \ldots, q_k, w_k) \to (q', u_1', w_1', \ldots, q_k', w_k')$$

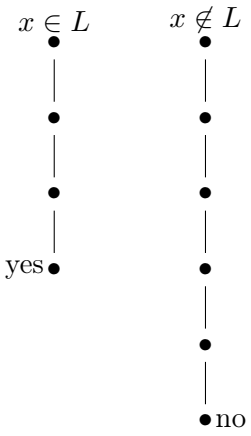
con $(q', u_1', w_1', \ldots, q_k', w_k')$ uno dei possibili risultati dell'applicazione di Δ.

Definizione 6.6.2 (Linguaggio deciso da una Macchina Nondeterministica)**.** Un linguaggio $L \subseteq (\Sigma \backslash \{\sqcup\})^*$ è deciso da una macchina nondeterministica $\mathcal{N}$ (con $\mathcal{N}(x)$ che termina sempre)

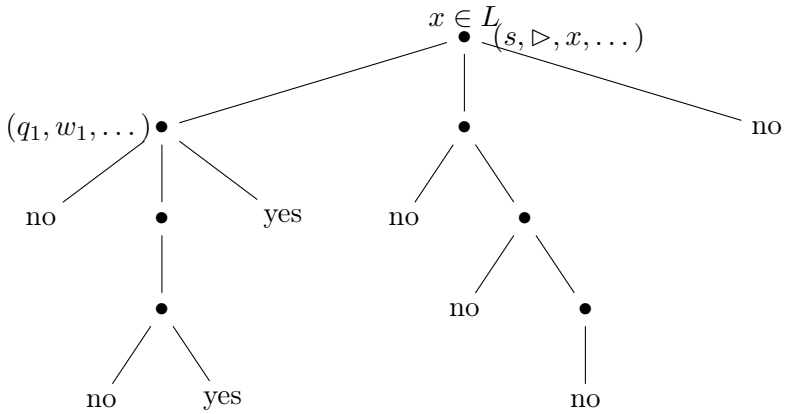
se $\forall x \in L(\Sigma \backslash \{\sqcup\})^*$

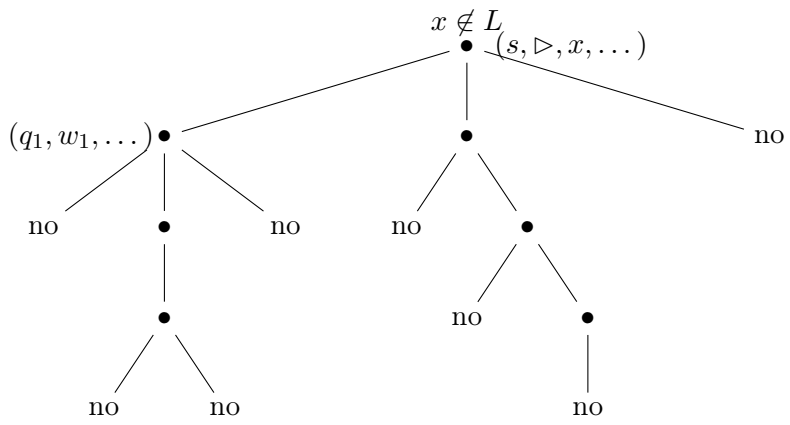
- $x \in L \Rightarrow$ esiste una computazione di $\mathcal{N}$ che inizia da $(s, \triangleright, x, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon)$ e termina in $(\text{yes}, \ldots)$
- $x \notin L \Rightarrow$ tutte le computazioni di $\mathcal{N}$ che iniziano da $(s, \triangleright, x, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon)$ terminano in $(\text{no}, \ldots)$

Se si ha una macchina che decide un linguaggio, bisogna definire la complessità spaziale e temporale su quella macchina. Per le macchine deterministiche, si ha che la complessità temporale è la lunghezza della computazione ne laso peggiore, per ogni possibile stringa di lunghezza n:



Ma nel caso nondeterministico si hanno degli alberi:





In questo caso, la complessità si può definire come l'altezza dell'albero.

Definizione 6.6.3 (Complessità Temporale di una Macchina Nondeterministica)**.** Una macchina di Turing nondeterministica $\mathcal{N}$ con input x richiede tempo t se ogni possibile computazione di $\mathcal{N}$ su x ha lunghezza al massimo t. $\mathcal{N}$ richiede tempo $f(n)$ se, per ogni x, $\mathcal{N}$ termina su x in tempo $f(|x|)$.

Esempio Immaginiamo che il grado del nondeterminismo di una macchina $\mathcal{N}$ sia 3, ovvero i nodi del suo albero hanno grado 3 ($d = 3$). L'altezza è $f(|x|)$, e il numero di foglie è $d^{f(|x|)}$ (molto grande).

Definizione 6.6.4 (NTIME($f(n)$))**.** $L \in \text{NTIME}(f(n))$ se esiste una macchina di Turing nondeterministica $\mathcal{N}$ che decide L in tempo $f(n)$.

Notare che, ad esempio, $\text{TIME}(n^5) \neq \text{NTIME}(n^5)$. Quest'ultimo è l'insieme di tutti i linguaggi che possono essere decisi in tempo n^5 da una macchina di Turing nondeterministica.

Proposizione 6.6.1.

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

Abbiamo che

$$\text{P} = \bigcup_{h \in \mathbb{N}} \text{TIME}(n^h) \qquad \text{NP} = \bigcup_{h \in \mathbb{N}} \text{NTIME}(n^h)$$

Quindi

$$\text{P} \subseteq \text{NP}$$

Abbiamo definito la complessità temporale di una macchina nondeterministica. Possiamo definire anche la complessità spaziale come la configurazione (nodo dell'albero) massima.

Definizione 6.6.5 (Complessità Spaziale di una Macchina Nondeterministica)**.** Una macchina di Turing nondeterministica con I/O $\mathcal{N}$ sull'input x utilizza spazio s se

$$s \geq \sum_{h=2}^{k-1} |w_h| + |u_h|$$

Ovvero s è il massimo su tutte le possibili computazioni di $\mathcal{N}$ su x

$$s = \max\left(\sum |w_h| + |u_h|\right)$$

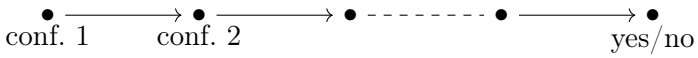
$\mathcal{N}$ lavora in NSPACE$f(n)$, $\mathcal{N}$ decide L in NSPACE$f(n)$.

$$\text{PSPACE} = \mathbb{L} \quad \text{spazio polinomiale su modello deterministico}$$
$$\text{NPSPACE} = \text{N}\mathbb{L} \quad \text{spazio polinomiale su modello nondeterministico}$$

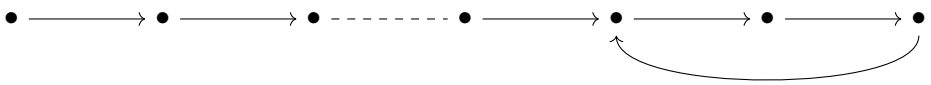
6.6.1 Simulazione di una Macchina Nondeterministica

In una macchina di Turing nondeterministica si possono avere tre possibili casi durante la computazione:

1. Si arriva ad una foglia con yes o no. In questo caso una macchina di Turing deterministica si comporta come



2. C'è un loop dal quale non si può uscire una volta entrati. In una macchina deterministica si ha



3. Si ha un loop dal quale si può uscire. In una macchina deterministica si ha un cammino infinito che continua a cambiare configurazione

Come già detto in precedenza, dato che questo è un corso sulla complessità, si considerano solo macchine di Turing che terminano sempre. Possiamo quindi procedere con la simulazione di una macchina nondeterministica da parte di una macchina deterministica. Ricordiamo che, in una macchina di Turing nondeterministica $\mathcal{N} = (K, \Sigma, \Delta, s)$, Δ è la relazione di transizione

$$\Delta \subseteq \underbrace{K \times \Sigma^k}_{\substack{\text{input della} \\ \text{relazione di} \\ \text{transizione}}} \times (K \cup \{\text{yes}, \text{no}, \text{halt}\}) \times \Sigma^k \times \{\leftarrow, \rightarrow, -\}^k$$

Definizione 6.6.6 (Grado di nondeterminismo)**.** Il grado di nondeterminismo di una macchina nondeterministica $\mathcal{N}$ è la ramificazione (il grado) massimo dell'albero di computazione di $\mathcal{N}$ per ogni $x \in \Sigma^*$.

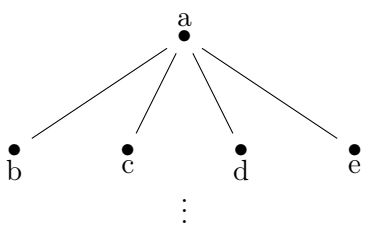
$$d(\mathcal{N}) = \max_{\substack{q \in K \\ \sigma_1, \ldots, \sigma_k \in \Sigma}} |\{(q, \sigma_1, \ldots, \sigma_k, q', \ldots) \in \Delta\}|$$

In particolare si ha

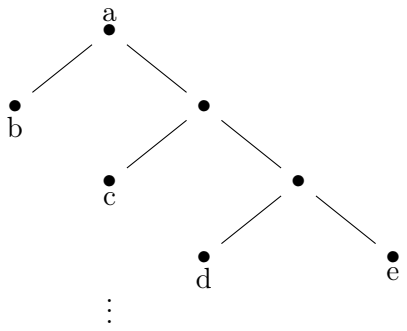
$$d(\mathcal{N}) = 1 \Leftrightarrow \mathcal{N} \text{ è deterministica}$$

Consideriamo solo macchine con grado di nondeterminismo $d(\mathcal{N}) \geq 2$. Sia $\mathcal{N}$ macchina con $d(\mathcal{N}) = d$. $\mathcal{N}$ decide L in tempo $f(n)$ (altezza dell'albero). Studiamo come la compessità temporale cambia restringendo il potere della macchina, ad esempio utilizzando $\mathcal{N}'$ con $d(\mathcal{N}') = 2$.

- In $\mathcal{N}$ l'altezza dell'albero è $f(n)$, e nel caso perggiore si hanno $d^{f(n)}$ foglie.



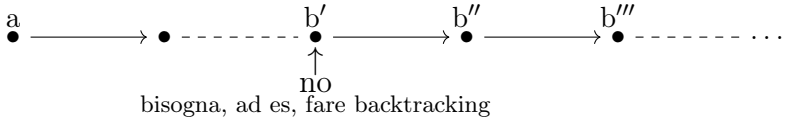
- In $\mathcal{N}'$ l'albero è del tipo



In questo caso l'altezza dell'albero è $\log_2(d^{f(n)}) = f(n)\log_2(d)$,

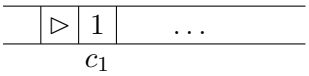
L'altezza cambia quindi solo di una costante, da $f(n)$ a $f(n)\log_2(d)$.

Se proviamo a fare la stessa trasformazione utilizzando una macchina deterministica $\mathcal{M}$, la lunghezza è $(d')^{f(n)}$.



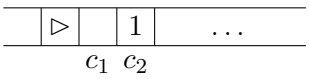
Teorema 6.6.1. Se L è deciso da una macchina di Turing nondeterministica $\mathcal{N}$ in tempo $f(n)$, allora L può essere deciso da una macchina di Turing deterministica $\mathcal{M}$ in tempo $O(c^{f(n)})$, con c costante che dipende da $\mathcal{N}$.

Dimostrazione (intuizione) La macchina $\mathcal{N}$ ha un input tape, una serie di working tape, e un output tape. La macchina $\mathcal{M}$ è costruita allo stesso modo, ma contiene un working tape in più. In ogni momento su questo nastro vengono registrate le scelte fatte dalla macchina $\mathcal{N}$. Ad esempio:

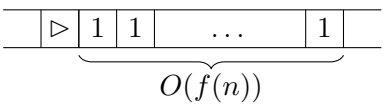


L'1 significa che viene preso il primo ramo dell'albero: viene simulata $\mathcal{N}$ con, come prima scelta, c_1.

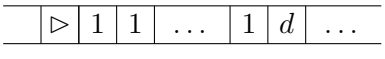
- Se $\mathcal{N}$ termina con yes, $\mathcal{M}$ termina con yes
- Se $\mathcal{N}$ termina con no, c_1 viene incrementato (di 1) e si procede nella simulazione
- Se $\mathcal{N}$ non termina, si genara $c_2 = 1$



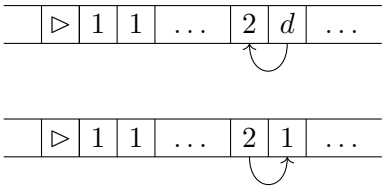
Il cammino più a sinistra di $\mathcal{N}$ sarà



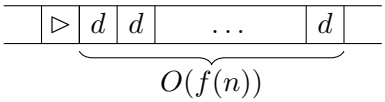
Se ci si trova in un caso in cui vengono percorsi tutti i rami uscenti da un nodo senza arrivare ad una soluzione yes, ad esempio



si farà backtracking:



Se ogni cammino finisce con no, si avrà



che corrisponde all'ultimo cammino nell'albero.

Per generare uno dei bit della sequenza (che vogliamo simulare) di scelte, $\mathcal{M}$ utilizza un numero di passi pari a $O\left(f(n) \cdot \log_2(d)\right)$. Quindi, per generarli tutti, si ha

$$O\left(d^{f(n)} \cdot f(n) \cdot \log_2(d)\right)$$

I passi di simulazione di $\mathcal{N}$ sono

$$O\left(d^{f(n)}\right)$$

Unendo tutto, abbiamo che

$$O\left(d^{f(n)} + d^{f(n)} \cdot f(n) \cdot \log_2(d)\right) = O\left(d^{f(n)} \cdot 2^{f(n)}\right) = O\left(c^{f(n)}\right)$$

Notiamo come questa simulazione può essere fatta senza conoscere $f(n)$. □

Corollario 6.6.1.1.

$$\text{NP} \subseteq \text{EXP}$$

Dimostrazione $\mathcal{N}$ decide L in tempo n^h. $\mathcal{N}$ può essere simulata da una macchina deterministica $\mathcal{M}$ in tempo

$$O\left(c^{n^h}\right) = O\left(2^{\log_2 c \cdot n^h}\right) = O\left(2^{n^{h+1}}\right)$$

che è incluso nella classe EXP. □

Nota $O(c^{n^h})$ non è dello stesso ordine di $\Theta(2^{n^h})$.

Capitolo 7

Relazioni tra Classi di Complessità

Capitolo 7 del libro di Papadimitriou. Finora abbiamo visto:

1. **Modelli di computazione**: macchine di Turing
2. **Modi di computazione**: deterministico, nondeterministico
3. **Risorse**: tempo, spazio
4. **Funzioni utilizzate come limiti**, ad esempio TIME$f(n)$, SPACE$f(n)$

In questo capitolo vedremo meglio il punto 4, e inoltre

- **Funzioni di complessità proprie**
- Due risultati fondamentali:
  - **Hierarchy Theorem**, ovvero se f è propria, allora $\text{TIME}(f(n)) \subsetneq \text{TIME}(f(n)^3)$. Esiste una gerarchia propria tra classi che sono tutte non uguali tra loro;
  - **Gap Theorem**, ovvero che esiste f non propria tale che $\text{TIME}(f(n)) = \text{TIME}(2^{f(n)})$.

7.1 Classi di Complessità

Definizione 7.1.1 (Funzione di Complessità Propria)**.** Una funzione $f : \mathbb{N} \to \mathbb{N}$ è una funzione di complessità propria se

1. f è non decrescente
2. Esiste $\mathcal{M}_f$ macchina di Turing deterministica con I/O tale che, per ogni x con $|x| = n$

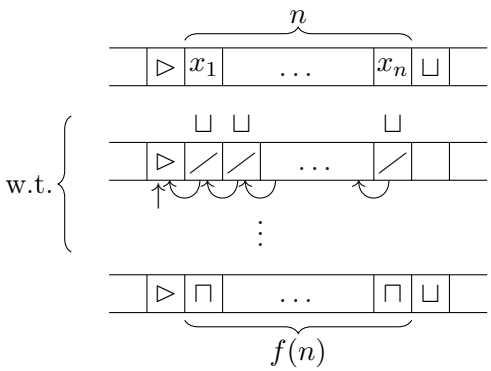
$$(s, \triangleright, x, \triangleright, \varepsilon, \dots, \triangleright, \varepsilon) \to^{(\mathcal{M}_f)t} (h, \underbrace{u, w}_{\triangleright x \sqcup}, \underbrace{\triangleright, \sqcup^{j_2}, \dots, \triangleright, \sqcup^{j_{k-1}}}_{\text{working tapes}}, \underbrace{\triangleright, \sqcap^{f(x)}}_{\text{output tape}})$$

con $\sqcap^{f(x)}$ rappresentazione unaria di $f(x)$. Inoltre,

- $t \in O(n + f(n))$
- $j_2, \dots, j_{k-1} \in O(f(n))$
- $t, j_2, \dots, j_{k-1}$ non dipendono da x

Nota In $t \in O(n + f(n))$, bisogna aggiungere n perché altrimenti non si potrebbe avere $\log(n)$ come funzione propria (bisogna leggere n).

Esempio Si ha $\mathcal{M}_f$



Su ogni working tape, viene eliminato tutto e si ferma a $\triangleright$. Lo spazio utilizzato in tutti i working tape è $O(f(n))$.

Esempi di funzioni proprie

$p(n)$ polinomio
$\log(n)$ logaritmo
c costante
$2^{p(n)}$ esponenziale
$2^{2^{\cdots^{2^{p(n)}}}}$ torre di esponenziali

La composizione di funzioni proprie è propria.

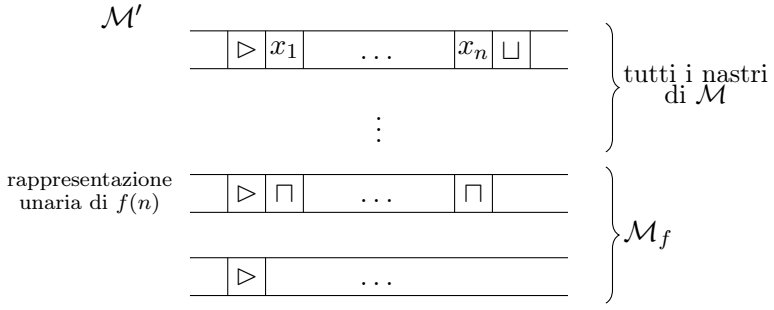
Definizione 7.1.2 (Macchine di Turing Precise)**.** Una macchina di Turing con I/O $\mathcal{M}$ è precisa se $\exists f, g$ tali che $\forall x \in \Sigma^*$, $\mathcal{M}$ su x termina in tempo $f(|x|)$ e utilizza spazio $g(|x|)$.

Ad esempio, merge sort è preciso, mentre heap sort non lo è.

Teorema 7.1.1.

L è decidibile in TIME($f(n)$) e f è propria
$\Downarrow$
$\exists$ una macchina di Turing precisa $\mathcal{M}$ che decide L in tempo $O(f(n))$

Dimostrazione (idea) (errore nel libro, proposizione 7.1.: macchina precisa in TIME e non in SPACE) Sappiamo che se $L \in$TIME($f(n)$), allora $\mathcal{M}$ decide L in $f(n)$ passi. Se f è propria, $\mathcal{M}_f$ computa $f(n)$.



□

7.1.1 Classi di Complessità Complemento

Una classe di complessità è un insieme di linguaggi (problemi) che possono essere decisi in un certo tempo, o spazio. Un linguaggio $L \subseteq \Sigma^*$ è un insieme di stringhe. Il complemento di L è definito come

$$\overline{L} = \{x \mid x \notin L\}$$

Definizione 7.1.3 (Complemento di una Classe di Complessità, co-$\mathcal{C}$)**.** Data una classe di complessità C, il complemento di C è

$$\text{co-}\mathcal{C} = \{\overline{L} \mid L \in C\}$$

Nota co-$\mathcal{C} \neq \overline{\mathcal{C}}$ (= complemento di tutte le classi $\mathcal{C}$).

Se $\mathcal{C}$ è deterministica (si utilizza una macchina deterministica), e $\Sigma = \{0, 1\}$, quando si prende un linguaggio appartenente a quella classe, ad esempio $L \in \text{TIME}(n \log n)$, allora esiste $\mathcal{M}$ macchina deterministica che decide L in tempo $n \log n$.

Per ottenere $\overline{L}$ si possono scambiare yes e no, e la macchina "complemento $\mathcal{M}$" decide $\overline{L} \in \text{TIME}(n \log n)$. Questo vale per tutte le classi deterministiche.

Proprietà 7.1.1.

$$\mathcal{C} \text{ è una classe di complessità deterministica } \Rightarrow \mathcal{C} = \text{co-}\mathcal{C}$$

Proprietà 7.1.2. Quando $\mathcal{C}$ è una classe di complessità deterministica

$$L \in \mathcal{C} \Rightarrow \overline{L} \in \mathcal{C} \Rightarrow \text{co-}\mathcal{C} \subseteq \mathcal{C}$$

Cosa succede se $\mathcal{C}$ è una classe di complessità nondeterministica?

Esempio Consideriamo $\text{NP} = \bigcup_{h \in \mathbb{N}} \text{NTIME}(n^h)$. Se $L \in \text{NP}$, allora esiste $\mathcal{N}$ macchina nondeterministica che decide L.

- Se $x \in L$, allora esiste un cammino nell'albero di computazione di $\mathcal{N}$ che arriva a yes.
- Se $x \notin L$, allora tutti i cammini nell'albero di computazione di $\mathcal{N}$ arrivano a no.

Consideriamo $\mathcal{N}'$ in cui le risposte di $\mathcal{N}$ sono invertite, che decide il linguaggio L'. Nel caso in cui $x \in L$, non è vero che $x \in L'$. Nell'albero di $\mathcal{N}'$ otteniamo alcuni yes non voluti, e quindi decide un linguaggio più ampio di $\overline{L}$. Quindi $\mathcal{N}'$ decide L' con $L' \subseteq \overline{L}$.

Satisfiability Problem Iniziamo con un esempio. Consideriamo la formula booleana

$$\varphi = (p \vee \neg q) \wedge (r \vee \neg p)$$

Quando si fissa una valutazione $v\{p, q, r\} \to \{0, 1\}$, si può calcolare il valore di φ con v. Ad esempio, se $v(p) = 1$, $v(q) = 0$, $v(r) = 0$, allora $\varphi[v] = 0$.

Proposizione 7.1.1 (Problema di Soddisfacibilità)**.** Data una formula booleana φ, decidere se esiste $v : Var(\varphi) \to \{0, 1\}$ tale che $\varphi[v] = 1$.

In altre parole si vuole trovare il linguaggio

$$L = \{\varphi \mid \varphi \text{ è soddisfacibile}\}$$

Studiamone la complessità. Una macchina nondeterministica efficiente per L è:

- In input si ha la formula

input		▷	p	$\vee$	$\neg$	q	$\wedge$	r	$\vee$	$\neg$	p	⊔

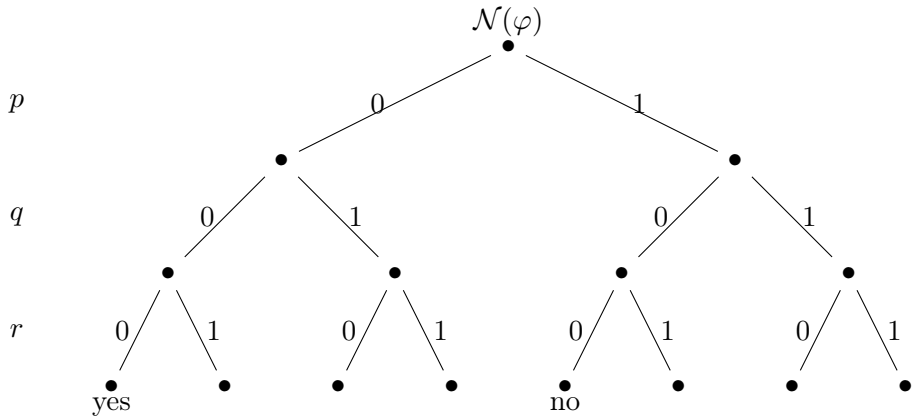
- Scansionare l'input per contare le variabili

w.t. 1		▷	p	q	r	

- Generare nondeterministicamente un assegnamento

w.t. 2		▷	0	0	0	

- Controllare il valore di verità di tale assegnamento



Quindi SAT $\in$ NP.

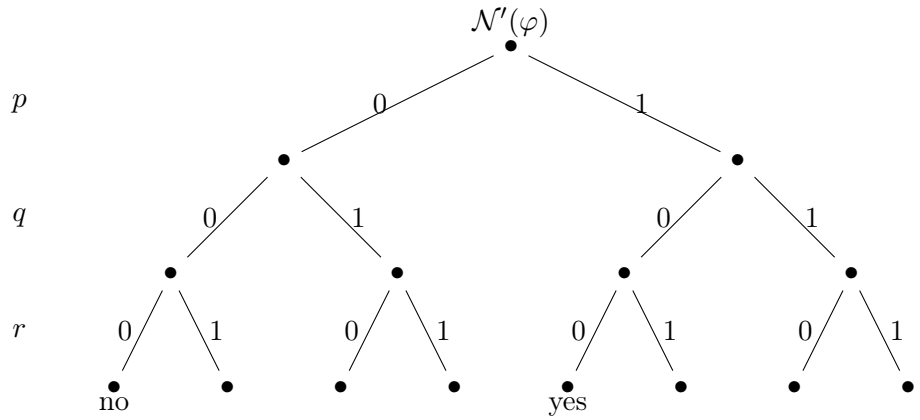
Unsatisfiability Problem Cosa succede per $\overline{L}$?

Proposizione 7.1.2 (Problema di Insoddisfacibilità)**.** Data una formula booleana φ, decidere se per ogni $v : Var(\varphi) \to \{0, 1\}$ si ha $\varphi[v] = 0$.

In altre parole si vuole trovare il linguaggio

$$\overline{L} = \{\varphi \mid \varphi \text{ non è una formula, o è insoddisfacibile}\}$$

Controllare che φ sia o meno una formula è banale, quindi ci concentriamo sulla sua insoddisfacibilità. Se scambiamo yes e no nelle foglie dell'albero di computazione, otteniamo questo:



$\mathcal{N}'$ accetta φ, ma φ non è insoddisfacibile. $\mathcal{N}'$ non decide $\overline{L}$. Quindi UnSAT $\in$ co-NP.

Per il momento non conosciamo la relazione tra NP e co-NP.

$$\text{NP ? co-NP}$$

Decidere unSAT ha la stessa complessità di decidere Validity, quindi Validity $\in$ co-NP.

7.2 Hierarchy Theorem

Utilizzeremo il teorema per dire che esiste un numero infinito di classi di compessità diverse tra loro, ognuna propriamente contenuta dentro l'altra.

Teorema 7.2.1 (Hierarchy Theorem)**.**

$$f \text{ è propria } \Rightarrow \text{ TIME}(f(n)) \subsetneq \text{TIME}(f(n)^3)$$

Dimostrazione Versione quantitativa della dimostrazione dell'Halting Theorem (Capitolo 3 del libro).

Teorema 7.2.2 (Halting Theorem)**.** Il linguaggio

$$H = \{\mathcal{M}; x \mid \mathcal{M}(x) \neq \uparrow\}$$

è ricorsivamente enumerabile, ma non ricorsivo.

Dimostrazione (intuizione) Si costruisce una macchina universale $\mathcal{U}$ che riceve in input la rappresentazione binaria della macchina $\mathcal{M}$ e della stringa x, e simula $\mathcal{M}$ su x. Ci sarà un working tape che contiene la configurazione di $\mathcal{M}$. Si può dimostrare che non si può modificare $\mathcal{U}$ in modo da dimostrare se $\mathcal{M}$ termina o meno su x (Se $\mathcal{M}$ termina su x, $\mathcal{U}$ accetta; se $\mathcal{M}$ non termina su x, $\mathcal{U}$ non termina).

Aggiungeremo un limite al numero di passi che si possono fare, dando quindi un risultato quantitativo.

7.2.1 Dimostrazione dello Hierarchy Theorem

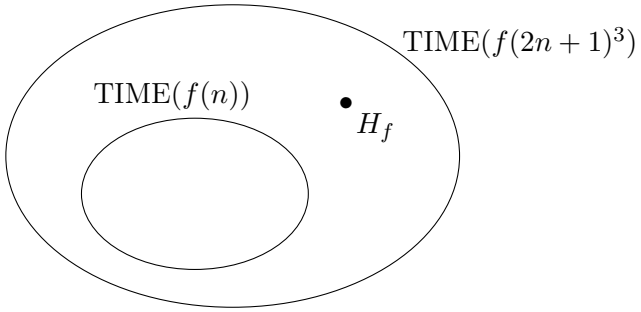
Consideriamo il linguaggio

$$H_f = \{\mathcal{M}; x \mid \mathcal{M} \text{ accetta } x \text{ in al massimo } f(|x|) + 5|x| + 4 \text{ passi}\}$$

Inoltre $\mathcal{M}(x)\downarrow$, e $\mathcal{M}(x) = \text{yes}$. Per lo Hierarchy Theorem, f è propria, e

$$H_f \in \text{TIME}\left(f(2n+1)^3\right) \qquad H_f \notin \text{TIME}(f(n))$$

Queste due classi di complessità sono diverse e sono una contenuta nell'altra.



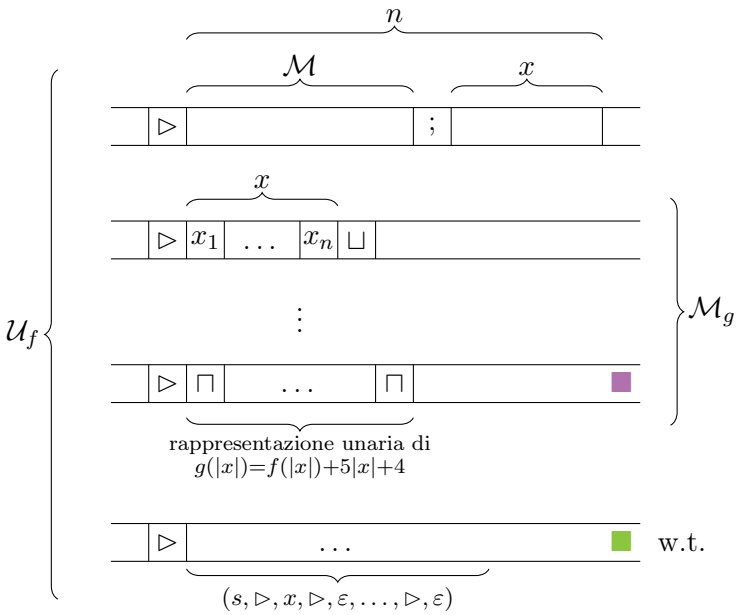
Quindi

$$\text{TIME}(f(n)) \subsetneq \text{TIME}\left(f(2n+1)^3\right)$$

Lemma 7.2.3.

$$H_f \in \text{TIME}(f(n)^3)$$

Dimostrazione Si costruisce una macchina universale $\mathcal{U}_f$. Se f è una funzione propria, allora anche la funzione $g(n) = f(n) + 5n + 4$ è propria.



In un dato tempo, si avrà la configurazione generica del tipo $(q, u_1, w_1, \ldots, u_k, w_k)$. Quindi, $\mathcal{U}_f$:

- Computa $f(|x|) + 5|x| + 4$ in unario
- Simula $\mathcal{M}$ su x, e per ogni passo simulato, elimina un $\sqcap$ dal nastro ■ (partendo dall'ultimo e sostituendolo con $\sqcup$)
- Se durante la simulazione, prima di aver eliminato tutti i $\sqcap$
  - $\mathcal{M}$ raggiunge $(\text{yes}, --)$, allora $\mathcal{U}_f$ termina con yes
  - $\mathcal{M}$ raggiunge $(\text{no}, --)$, allora $\mathcal{U}_f$ termina con no

  Se tutti i $\sqcap$ sono stati eliminati, $\mathcal{U}_f$ termina con no.

Quindi $\mathcal{U}_f$ decide H_f. Ma in quanto tempo? Ricordiamo che $f(n) \geq n$ ($f(n)$ è almeno lineare). $\mathcal{U}_f$ deve simulare al massimo $f(|x|) + 5|x| + 4$ passi di $\mathcal{M}$, con $f(|x|) + 5|x| + 4 = O(f(n))$, $n \geq |x|$.

Quanti passi di $\mathcal{U}_f$ sono necessari per simulare un singolo passo di $\mathcal{M}$? Un numero costante di scansioni dell'input e del nastro ■. Quanto è lungo al massimo il nastro ■? $O(f(n))$, perché è una configurazione ottenuta in al massimo $f(|x|) + 5|x| + 4 = O(f(n))$ passi.

Ciò che si ottiene è della forma

$$O\left(k' \cdot f(n) \cdot f(n)\right) = O\left(f(n)^3\right)$$

con k' costante che dipende da $\mathcal{M}$ (ad esempio, numero di nastri), e $k' \leq f(n)$. □

In questa dimostrazione non si ha bisogno di $f(|x|)+5|x|+4$: servirà nella dimostrazione successiva.

Esercizio (nel libro) Possiamo farlo in tempo $O(\log(f(n))^2 \cdot f(n))$, con $\log(f(n))^2 > f(n)$. Poiché f è propria, $\mathcal{M}_g$ raggiunge questo risultato in tempo $O(f(n))$.

$$O\left(k' \cdot f(n) \cdot f(n) + f(n)\right)$$

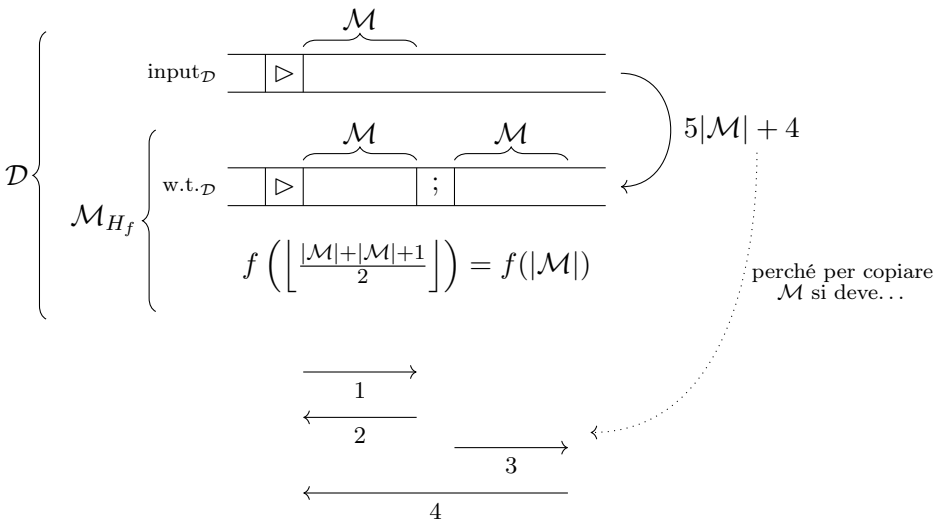
Lemma 7.2.4.

$$H_f \notin \text{TIME}\left(f\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right)$$

Dimostrazione Diagonalizzazione. Si supponga per contraddizione che esista $\mathcal{M}_{H_f}$ che decide H_f in tempo $f\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$. Si consideri la macchina $\mathcal{D}(\mathcal{M})$ tale che

$$\begin{gathered}\mathcal{D}(\mathcal{M}) \text{ termina con yes} \\ \Updownarrow \\ \mathcal{M}_{H_f}(\mathcal{M};\mathcal{M}) \text{ termina con no} \\ \Updownarrow \\ \mathcal{M}(\mathcal{M}) = \text{no} \quad \vee \quad \mathcal{M}(\mathcal{M}) \downarrow \text{ dopo più di } f(n)+5n+4 \text{ passi} \quad \vee \quad \mathcal{M}(\mathcal{M}) \uparrow\end{gathered}$$

Quanti passi richiede $\mathcal{D}$?



Quindi $\mathcal{D}$ lavora in tempo al massimo $f(n)+5n+4$.

Cosa succede a $\mathcal{D}(\mathcal{D})$?

- Se $\mathcal{D}(\mathcal{D}) = \text{no}$, allora $\mathcal{M}_{H_f}(D(\mathcal{D});\mathcal{D}) = \text{yes}$, e $\mathcal{D}(\mathcal{D}) = \text{yes}$, che è una contraddizione.
- Se $\mathcal{D}(\mathcal{D}) = \text{yes}$, allora $\mathcal{M}_{H_f}(D(\mathcal{D});\mathcal{D}) = \text{no}$, quindi una delle seguenti:
  - $\mathcal{D}(\mathcal{D}) = \text{no}$, che è una contraddizione
  - $\mathcal{D}(\mathcal{D}) \uparrow$, che è una contraddizione
  - $\mathcal{D}(\mathcal{D}) \downarrow$ dopo più di $f(n)+5n+4$ passi, che è una contraddizione (perché abbiamo visto che $\mathcal{D}$ lavora in tempo al massimo $f(n)+5n+4$)

□

Corollario 7.2.4.1.

$$\text{P} \subsetneq \text{EXP}$$

Dimostrazione Abbiamo che

$$\begin{aligned}\text{P} &= \bigcup_{h\in\mathbb{N}} \text{TIME}\left(n^h\right) \\ &\subseteq \text{TIME}(2^n) \\ &\overset{\text{Hie. Th.}}{\subsetneq} \text{TIME}\left(\left(2^{2^n+1}\right)^3\right) \\ &\subseteq \text{TIME}\left(2^{n^2}\right)\end{aligned}$$

$$\subseteq \bigcup_{h\in\mathbb{N}} \text{TIME}\left(2^{n^h}\right) = \text{EXP}$$

□

Abbiamo dimostrato che P $\subsetneq$ EXP e NP $\subseteq$ EXP, ma non sappiamo se P $\subsetneq$ NP

$$\text{P} \overset{?}{\subsetneq} \text{NP} \overset{?}{\subsetneq} \text{EXP}$$

7.2.2 Gap Theorem

vediamo ora un teorema che sembra contraddire lo Hierarchy Theorem. In realtà, la funzione f nel Gap Theorem non è propria, a causa dello Hie. Theo.

Teorema 7.2.5 (Gap Theorem). $\exists f$ funzione ricorsiva tale che

$$\text{TIME}(f(n)) = \text{TIME}(2^{f(n)})$$

Dimostrazione L'idea è quella di definire $f(n)$ in modo che se una computazione termina in al massimo $2^{f(n)}$ passi, allora termina in al massimo $f(n)$ passi.

Si considerino un'enumerazione di macchine di Turing $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_h, \ldots$ e il predicato $P(i,k)$ definito come

$$\begin{aligned} P(i,k) \quad = \quad & \forall \mathcal{M}_h \text{ con } h \le i \qquad \forall x \quad |x| = i \\ & \mathcal{M}_h(x) \text{ termina in al massimo } k \text{ passi, o} \\ & \mathcal{M}_h(x) \text{ termina in più di } 2^k \text{ passi, o} \\ & \mathcal{M}_h(x) \text{ non termina} \end{aligned}$$

Questo predicato è ricorsivo (decidibile). Per definire $f(i)$ si considerino i seguenti intervalli:

$$\underset{k_1=2i}{[} \qquad \underset{k_2=2^{k_1}+1}{)[} \qquad \cdots \qquad \underset{k_j=2^{k_{j-1}}+1}{)[} \qquad \cdots$$

Zoommando in fuori, si può osservare un numero infinito di intervalli sempre più lunghi:

$$[\,)[\quad)[\qquad)[\qquad\quad)[\qquad\quad)[\qquad\qquad)\ldots$$

Sia

$$N(i) = \sum_{h=0}^{i} |\sigma_h|^i$$

con σ_h alfabeto di $\mathcal{M}_h$ ($|\sigma_h|^i$ è il numero di input di lunghezza i per $\mathcal{M}_i$). $N(i)$ è un numero molto grande. Ad esempio, $|x| = i$ input per $\mathcal{M}_3$, $\mathcal{M}_3 \downarrow$.

Si considerino $N(i)+1$ intervalli della forma $[k_j, k_{j+1})$. Per il principio della piccionaia, $\exists [k_l, k_{l+1})$ tale che nessuna delle macchine $\mathcal{M}_0, \ldots, \mathcal{M}_i$ termina in un numero di passi che cade nell'intervallo $[k_l, k_{l+1})$. In altre parole, $f(i) = k_l$, e $P(i, f(i))$ è vera.

Dobbiamo dimostrare che $\text{TIME}(2^{f(n)}) \subseteq \text{TIME}(f(n))$, ovvero che $\forall L$, se $L \in \text{TIME}(2^{f(n)})$ allora $L \in \text{TIME}(f(n))$. Sia $L \in \text{TIME}(2^{f(n)})$. Allora $\exists \mathcal{M}_j$ che decide L in al massimo $2^{f(n)}$ passi *.

$\forall x$ tali che $|x| \ge j$, la macchina $\mathcal{M}_j$ in $\mathcal{M}_0, \ldots, \mathcal{M}_j, \ldots, \mathcal{M}_{|x|}$ è una di queste. Ma $P(|x|, f(|x|))$ è vera, quindi

- $\mathcal{M}_j(k)$ termina in al massimo $f(|x|)$ passi, o
- $\mathcal{M}_j(k)$ termina in più di $2^{f(|x|)}$ passi *

(poichè $\mathcal{M}_j$ decide un linguaggio, non c'è un terzo caso). A causa di *, possiamo eliminare il secondo caso *. Quindi $\mathcal{M}_j$ decide L in $\text{TIME}(f(n))$.

Ma abbiamo detto che $\forall x$ t.c. $|x| \ge j$, dobbiamo dimostrarlo per ogni input. Per risolvere questo, possiamo aggiungere una tabella che, per ognuno dei $|x| < j$, ha output lineare. Quindi $\mathcal{M}_j$ modificato sugli input più corti di j decide L in $\text{TIME}(f(n))$. □

7.3 Reachability Method

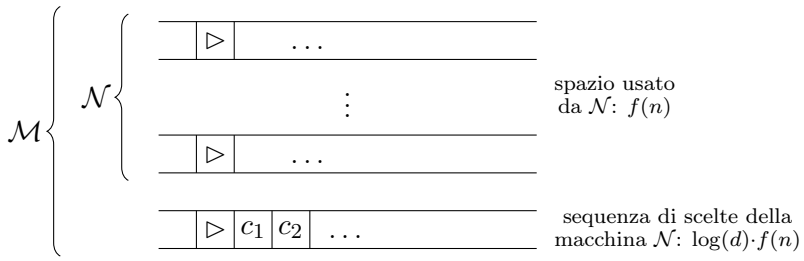
I teoremi appena visti ci dicono come classi dello stesso tipo (tempo deterministico, spazio deterministico) si relazionano tra loro quando variamo la funzione che rappresenta il limite di complessità. Risultati simili, sebbene molto più difficili da dimostrare, sono noti per classi di complessità non deterministiche. Tuttavia, le domande più interessanti nella teoria della complessità riguardano la relazione tra classi di tipo diverso, come P vs NP.

Teorema 7.3.1. Sia f una funzione propria. Allora

1. $\text{SPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$
2. $\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n))$
3. $\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$
4. $\text{SPACE}(f(n)) \subseteq \text{TIME}(c^{f(n)+\log(n)})$
5. $\text{NTIME}(f(n)) \subseteq \text{SPACE}(f(n))$
6. $\text{NSPACE}(f(n)) \subseteq \text{TIME}(c^{f(n)+\log(n)})$

Abbiamo già dimostrato 1, 2, 3 (segue da 2 e 5), e 4 (segue da 1 e 6). Dimostriamo 5 e 6.

Dimostrazione 5 Vogliamo dimostrare che se $L \in \text{NTIME}(f(n))$, allora $L \in \text{SPACE}(f(n))$. Questo significa che esiste $\mathcal{N}$ macchina di Turing non deterministica che decide L in al massimo $f(n)$ passi. Possiamo simularla con una macchina di Turing deterministica $\mathcal{M}$ che decide L in tempo $c^{f(n)}$, e quindi in spazio $f(n)$.



$\text{SPACE}f(n)$. □

Dimostrazione 6: Reachability Method Vogliamo dimostrare che se $L \in \text{NSPACE}(f(n))$, allora $L \in \text{TIME}(c^{f(n)+\log(n)})$. Questo significa che esiste $\mathcal{N}$ macchina di Turing non deterministica I/O che decide L in spazio $f(n)$. Sia $G_{\mathcal{N}}(x)$ il grafo di configurazioni di $\mathcal{N}$ su x definito come

$$G_{\mathcal{N}}(x) = (V_{\mathcal{N}}(x), E_{\mathcal{N}}(x))$$

con $V_{\mathcal{N}}(x)$ configurazione di $\mathcal{N}$ durante la computazione su x. Uniamo tutte le configurazioni che finiscono in yes (o no) in un unico nodo etichettato con yes (o no).

$$x \in L \ (\mathcal{N} \text{ accetta } x) \ \Leftrightarrow \ \text{in } G_{\mathcal{N}}(x) \text{ la configurazione iniziale raggiunge yes}$$

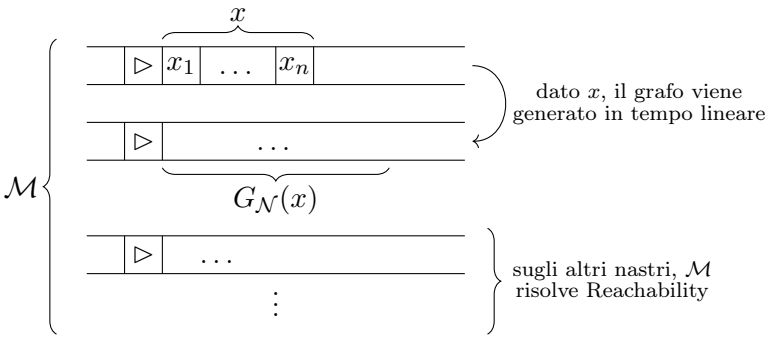
che è la complessità temporale di Reachability su $G_{\mathcal{N}}(x)$. Utilizzando una visita, a complessità (in pseudocodice) è $O(|G_{\mathcal{N}}(x)|)$. Utilizzando una macchina di Turing, Reachability $\in$ P (ad esempio, $O(|G_{\mathcal{N}}(x)|^{\beta})$). Vogliamo trovare il numero di nodi di questo grafo, ovvero $|V_{\mathcal{N}}(x)|$. Una configurazione è del tipo

$$(q, \underbrace{u_1, w_1}_{\text{input}}, \ldots, \underbrace{u_k, w_k}_{\text{output}})$$

$\mathcal{N}$ è I/O, quindi possiamo non considerare l'output nella configurazione. Inoltre, poiché l'input non può essere modificato (è sempre $\triangleright x \sqcup$), può essere rappresentato con (q, j), dove q è sempre 0, e l'indice $0 \leq j \leq |x| + 1$. Quindi

$$(q, j, u_1, w_1, \ldots, u_{k-1}, w_{k-1})$$

con $|u_i|, |w_i| \leq f(n)$, $\forall i = 2, \ldots, k-1$. L'idea è quella di considerare una macchina deterministica $\mathcal{M}$ che, sull'input x, genera $G_{\mathcal{N}}(x)$, e testa su $G_{\mathcal{N}}(x)$ se la configurazione iniziale raggiunge yes o no.



Il numero di configurazioni in questa forma, ovvero il numero di nodi di $G_{\mathcal{N}}(x)$, è

$$\begin{aligned} |V_{\mathcal{N}}(x)| &= (|k|+2) \cdot (|x|+2) \cdot \prod_{i=2}^{k-1} |\Sigma|^{2 \cdot f(|x|)} \\ &= (|k|+2) \cdot (|x|+2) \cdot |\Sigma|^{2 \cdot f(|x|) \cdot (k-2)} \end{aligned}$$

con +2 per yes e no, e k numero di nastri della macchina. I termini che non dipendono dalla macchina sono $|x| + 2$ e $f(|x|)$. Quindi

$$\begin{aligned} (|k|+2) \cdot (|x|+2) \cdot |\Sigma|^{2 \cdot f(|x|) \cdot (k-2)} &\leq \alpha^{f(|x|)+\log(|x|)} \\ |G_{\mathcal{N}}(x)| \leq \left(\alpha^{f(|x|)+\log(|x|)}\right)^2 &\leq \gamma^{f(|x|)+\log(|x|)} \end{aligned}$$

Richiede

$$\text{TIME}\left(\left(\gamma^{f(|x|)+\log(|x|)}\right)^{\beta}\right) = \text{TIME}\left(c^{f(n)+\log(n)}\right)$$

□

Corollario 7.3.1.1. Per lo Hierarchy Theorem,

$$\text{SPACE}(\log(n)) = \mathbb{L} \subsetneq \text{N}\mathbb{L} \subsetneq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{NPSPACE} \subseteq \text{EXP} \subseteq \text{NEXP} \ldots$$

Vedremo che PSPACE = NPSPACE. Un problema aperto è $\mathbb{L} \stackrel{?}{=} \text{N}\mathbb{L}$.

7.3.1 Spazio Nondeterministico

Dato un grafo $G(V, E)$, e $u, v \in E$, il problema della raggiungibilità (Reachability) chiede se esiste un cammino da u a v in $G(V, E)$. Vedremo come questo problema, Reachability, è in $\text{SPACE}((\log n)^2)$.

Teorema 7.3.2 (Savitch).

$$\text{Reachability} \in \text{SPACE}((\log n)^2)$$

Dimostrazione Sia

$$\text{Path}(a, b, i) \text{ true} \Leftrightarrow \exists \text{ cammino da } a \text{ a } b \text{ di lunghezza al massimo } 2^i$$

Il caso base è

$$\text{Path}(a, b, 0) \text{ true} \quad \Leftrightarrow \quad \begin{cases} a = b \\ (a, b) \in E \text{ (esiste un arco da } a \text{ a } b) \end{cases}$$

Continuando,

$$\text{Path}(a, b, i+1) \text{ true}$$

$$\Updownarrow$$

$$a \xrightarrow{\overbrace{\quad\quad}^{\text{al max } 2^i}\ \overbrace{\quad\quad}^{\text{al max } 2^i}} b$$

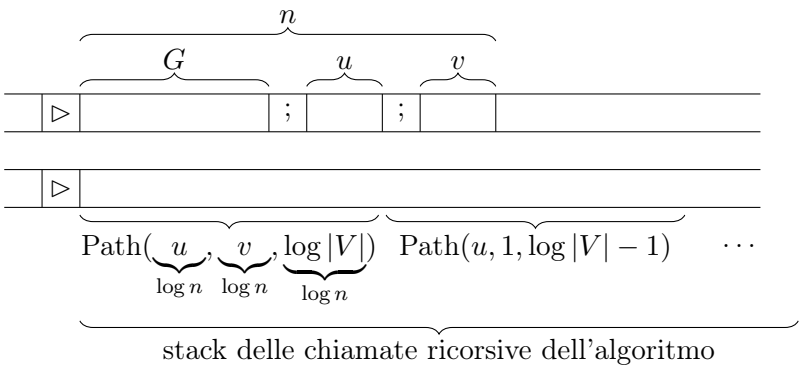
$$\underbrace{\qquad\qquad}_{\text{al massimo } 2^{i+1}}$$

$$\Updownarrow$$

$$\exists z \quad \text{Path}(a, z, i) \ \wedge \ \text{Path}(z, b, i)$$

In G, ci possono volere al massimo $|V|$ passi per raggiungere v da u. Quindi

$$\text{In } G, \ u \text{ raggiunge } v \ \Leftrightarrow \ \text{Path}(u, v, \log |V|) \text{ true}$$



Nel caso peggiore si ha la tripla $(\log n, \log n, \log n)$ ripetuta $\log n$ volte, quindi SPACE$((\log n)^2)$. □

È inefficiente in termini di tempo, ma efficiente in spazio.

Esempio Il grafo G ha 8 nodi, quindi $|V| = 8$, $V = \{1, 2, \ldots, 8\}$, $u = 1$, $v = 8$. Se scriviamo Path$(1, 8, 3)$, si sta cercando un cammino da u a v lungo al massimo 2^3.

Corollario 7.3.2.1.

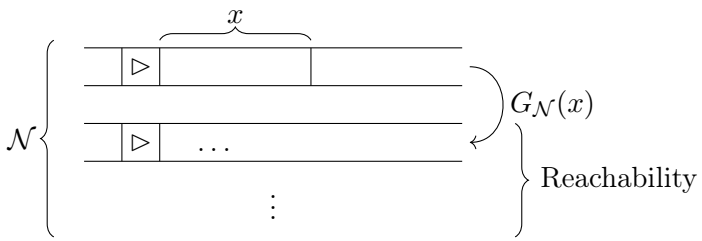
$$\text{NSPSACE}(f(n)) \subseteq \text{SPACE}(f(n)^2)$$

Dimostrazione Sia $\mathcal{N}$ macchina che decide L in spazio $f(n)$. Allora

$$|G_{\mathcal{N}}(x)| = \gamma^{f(|x|)+\log(|x|)}$$

Per Savitch

$$\text{SPACE}\left(\log(\gamma^{f(|x|)+\log(|x|)})^2\right) = \Theta(f(|x|)^2)$$



Questo significa che Reachability può essere risolto senza memorizzare il grafo $G_{\mathcal{N}}(x)$ in un working tape. □

Corollario 7.3.2.2.

$$\text{NPSPACE} = \text{PSPACE}$$

Corollario 7.3.2.3.

$$\text{N}\mathbb{L} = \text{SPACE}\left((\log n)^2\right)$$

Sappiamo che $\mathbb{L} \subseteq \text{N}\mathbb{L}$. Se dimostrassimo che Reachability $\in \text{SPACE}(\log n)$, allora $\mathbb{L} = \text{N}\mathbb{L}$.

Sappiamo che

$$\text{co-}l = \{\overline{L} \mid L \in l\}$$

Quindi

$$\text{NPSPACE} = \text{PSPACE} \;\Rightarrow\; \text{co-NPSPACE} = \text{PSPACE}$$

Inoltre

$$\text{co-NPSPACE}(f(n)) \subseteq \text{SPACE}(f(n)^2)$$

Dato $G_{\mathcal{N}}(x)$,

$x \in L \;\Leftrightarrow$ la configurazione iniziale raggiunge yes (Reachability)
$x \in \overline{L} \;\Leftrightarrow$ la configurazione iniziale non è in grado di raggiunge yes (Unreachability)

Vogliamo studiare la relazione tra co-NPSPACE($f(n)$) e NPSPACE($f(n)$). Qual è la complessità spaziale nondeterministica di Unreachability? Dato un grafo G e un nodo $u \in V$, contare il numero di nodi raggiungibili da u, con un algoritmo non deterministico e in spazio logaritmico.

Teorema 7.3.3 (Immerman Szelepcsényi)**.** Dato $G = (V, E)$ e $x \in V$, il numero di nodi raggiungibili da x è in NSPACE($\log n$).

Per un problema di decisione, questo significa che nell'albero di computazione di $\mathcal{N}(x)$ è sufficente uno yes nelle foglie per dire $x \in L$.

Per la computazione di $f : \Sigma \to \mathbb{N}$, i rami non corretti devono terminare con no. Quando un ramo termina con un numero ($f(x)$), siamo certi che il risultato è corretto.

Dimostrazione Indichiamo con $S(i)$ l'insieme di nodi raggiungibili da x in al massimo i passi. Vogliamo calcolare $|S(|V| - 1)|$. Iniziamo da

$$S(0) = \{u\}$$

e quindi $|S(0)| = 1$. Poi

for $k = 1$ to $|V| - 1$
    compute $|S(k)|$ from $|S(k-1)|$

Come si può calcolare $|S(k)|$?

```
l := 0
for u = (1, 2, ..., |V|)
    if u ∈ S(k) then l := l+1
```

Come si può controllare $u \in S(k)$? L'idea è che

$$u \in S \quad \Leftrightarrow \quad \exists b \in S(k-1) \ \wedge \ \underbrace{(u = v \ \vee \ (v,u) \in E)}_{G(v,u)}$$

Quindi

$$\underbrace{\underbrace{x \rightsquigarrow v}_{\leq x-1} \longrightarrow u}_{\leq x}$$

Conosciamo $|S(k-1)|$, possiamo controllare se $u \in S(k)$

```
m := 0
reply := false  % diventa true se u ∈ S(k)
for v = (1, 2, ..., |V|)
    if v ∈ S(k − 1)
        m := m+1
        if G(v, u) then reply := true
if (m = |S(k − 1|)) then reply
else "no"
```

Il passaggio $v \in S(k-1)$ viene eseguito nondeterministicamente.

```
w_0 := x
for (p = (1, 2, ..., k − 1))
    guess w_p
    check G(w_{p−1}, w_p)
if (w_{k−1} = v) then true
else false
```

Durante la computazione bisogna memorizzare tutte le variabili $|S(k-1)|$, k, l, m, reply, u, v, w_p, w_{p-1}, p. Queste variabili sono tutte $\in [0, |V|]$. Ognuna di loro richiede spazio $O(\log |V|)$. Poiché non viene mai memorizzato un intero cammino, lo spazio richiesto è proprio $O(\log |V|)$. □

Corollario 7.3.3.1.

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

con $f(n) \geq \log n$, f propria.

Dimostrazione Utilizzando il Reachability method. $\mathcal{N}$ decide L. Dato $G_{\mathcal{N}}(x)$, $x \in \overline{L}$ sse yes non è raggiungibile (dalla configurazione iniziale, ovvero tutti i nodi finiscono in no). □

Capitolo 8

Riduzione e Completezza

Capitolo 8 del libro. Alcuni problemi catturano la difficoltà di un'intera classe di complessità. La logica gioca un ruolo centrale in questo fenomeno.

8.1 Riduzioni

Si vuole risolvere un problema A "simile" al problema B, e si possiede un algoritmo efficiente per B. Si possono eseguire una serie di trasformazioni:

$$\underset{\text{input per } A}{x} \to \underset{\text{input per } B}{x'} \to \text{algoritmo} \to \underset{\text{output per } B}{y'} \to \underset{\text{output per } A}{y}$$

Se una trasformazione ha una complessità molto minore del problema, la complessità rimane la stessa. In particolare, la trasformazione più interessante è quella da x a x', e prende il nome di **riduzione**.

Definizione 8.1.1 (Riduzione). Una riduzione da L_1 a L_2 è

$$R : \Sigma_1^* \to \Sigma_2^* \quad \text{tale che} \quad x \in L_1 \Leftrightarrow R(x) \in L_2$$

La complessità principale deriva dall'algoritmo per decidere L_2.

R dev'essere computabile in spazio logaritmico.

Esempio (Riduzione) Si consideri il Graph 3-Coloring (L_1): dato un grafo $G = (V, E)$, decidere se è possibile definire $Col : V \to \{r, b, y\}$ tale che se $(u, v) \in E$ allora $Col(u) \neq Col(v)$.

Si consideri SAT (L_2): data una formula booleana, decidere se è soddisfacibile o meno.

$$R : \text{Graph 3-Coloring} \to \text{SAT}$$

ovvero

$$R(G) \text{ è soddisfacibile } \Leftrightarrow \ G \text{ è 3-colorabile}$$

Abbiamo che

$$\begin{aligned} R(G) \quad = \quad & \bigwedge_{u \in V} [(r_u \vee b_u \vee y_u) \wedge (r_u \to \neg b_u \wedge \neg u_u) \wedge (b_u \to \neg r_u \wedge \neg y_u) \wedge (y_u \to \neg r_u \wedge \neg b_u)] \wedge \\ & \bigwedge_{(u,v) \in E} [(r_u \to \neg r_v) \wedge (b_u \to \neg b_v) \wedge (y_u \to \neg y_v)] \end{aligned}$$

Se in una macchina di Turing con I/O si ha G sul nastro di input e $R(G)$ sul nastro di output, la riduzione deve utilizzare spazio logaritmico sui working tape.

Definizione 8.1.2 ($L_1 \leq L_2$)**.** L_1 può essere ridotto a L_2 ($L_1 \leq L_2$) se esiste una riduzione R da L_1 a L_2.

Proprietà 8.1.1. ($\circ$ è la composizione di funzioni)

$$R_1 \text{ riduzione da } L_1 \text{ a } L_2$$
$$R_2 \text{ riduzione da } L_2 \text{ a } L_3$$
$$\Downarrow$$
$$R_2 \circ R_1 \text{ riduzione da } L_1 \text{ a } L_3$$

Questo significa che $L_1 \leq L_2$ e $L_2 \leq L_3$ implica $L_1 \leq L_3$. Inoltre, $L_1 \leq L_2$ significa che, in termini di complessità, L_1 è al più difficile di L_2.

Definizione 8.1.3 (Chiusura per Riduzione)**.** Sia $\mathcal{C}$ una classe di complessità. $\mathcal{C}$ è chiusa per riduzione se

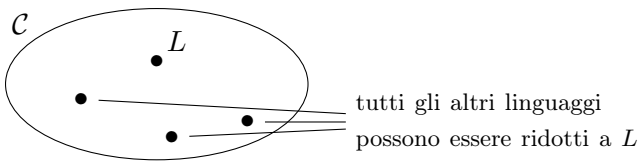
$$L_1 \leq L_2 \text{ e } L_2 \in \mathcal{C} \Rightarrow L_1 \in \mathcal{C}$$

Si può dimostrare che, ad esempio, P, NP, EXP, $\mathbb{L}$, N$\mathbb{L}$, PSPACE sono tutte chiuse per riduzione. *Esercizio: trovare un esempio di una classe di complessità non chiusa per riduzione.*

8.1.1 Completezza

Definizione 8.1.4 (Completezza di una Classe di Complessità)**.** Un linguaggio L è completo per una classe $\mathcal{C}$ se

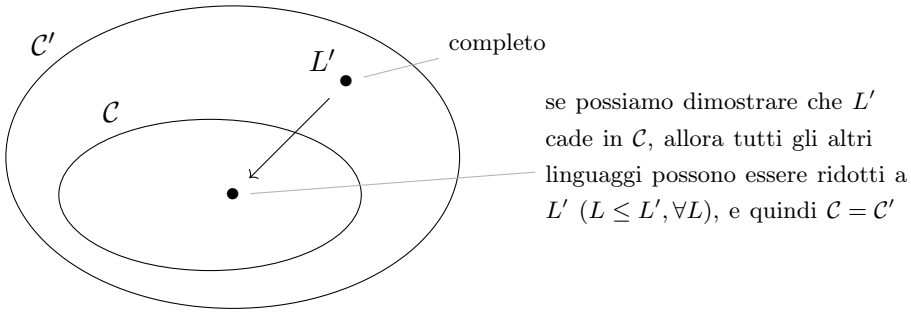
1. $L \in \mathcal{C}$
2. $\forall L' \in \mathcal{C}$, $L' \leq L$



Proprietà 8.1.2. Se $\mathcal{C}$ e $\mathcal{C}'$ sono classi di complessità

- chiuse per riduzione, e
- $\mathcal{C} \subseteq \mathcal{C}'$, e
- L' è completo per $\mathcal{C}'$, e
- $L' \in \mathcal{C}$

allora $\mathcal{C} = \mathcal{C}'$.



8.1.2 Problema P-Completo: Circuit value

Una **formula booleana** è coposta da variabili booleane $x_1, \ldots, x_n$, da costanti $0, 1$, e da operatori $\wedge, \vee, \neg$. Una **funzione booleana** è

$$\varphi : \{0,1\}^m \to \{0,1\}$$

Esempio m=3

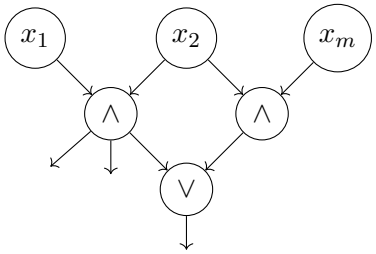
$$\left.\begin{array}{l} \varphi(\overset{x_1}{0}, \overset{x_2}{0}, \overset{x_3}{0}) = 1 \\ \varphi(0,1,0) = 1 \\ \ldots \\ \varphi(1,1,1) = 0 \end{array}\right\} \text{dominio } |\{0,1\}^3| = 8$$

Questa funzione booleana ha valore 1 solo in due casi, ed è quindi equivalente alla formula booleana

$$(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge 2 \wedge \neg x_3)$$

Formule booleane ed espressioni booleane hanno lo stesso potere espressivo.

I **circuiti booleani** (boolean circuits) sono equivalenti a formule booleane ed espressioni booleane. Un circuito è composto da *gates*, ovvero nodi di un grafo. I gates sono di tre tipi: variabili, costanti e operazioni.



TODO: finire lezione 20