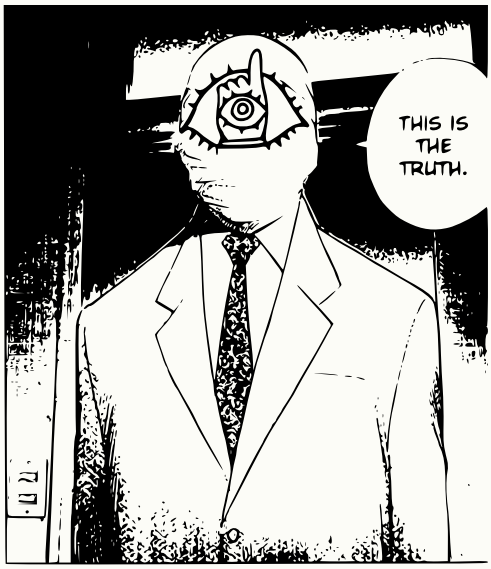
Marco Casu

Automi, Calcolabilità e Complessità

Sapienza
Università di Roma

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Questo documento è distribuito sotto la licenza GNU, è un resoconto degli appunti (eventualmente integrati con libri di testo) tratti dalle lezioni del corso di Automi, Calcolabilità e Complessità per la laurea triennale in Informatica. Se dovessi notare errori, ti prego di segnalarmeli.



INDICE

CAPITOLO

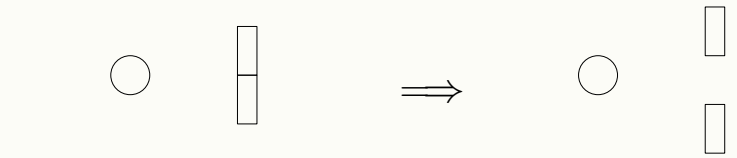
1

AUTOMI

1.1 Linguaggi Regolari

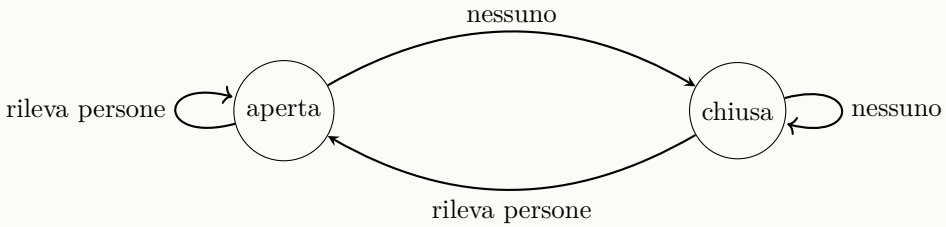
Un *automa a stati finiti* è, seppure limitato nella memoria e nella gestione dell'input, il più semplice modello di computazione. Un automa può interagire con l'input esclusivamente "scorrendolo" in maniera sequenziale.

Esempio : Si vuole modellare una semplice porta con sensore, che si apre quando qualcuno si trova nelle vicinanze.



una persona si trova davanti la porta — la porta si apre

Un automa che modella il problema è il seguente :



Un automa ha alcuni stati speciali, come quello iniziale, indicato con un apposita freccia, e degli stati detti *di accettazione*, ossia stati in cui deve necessariamente terminare la computazione per essere definita valida, vengono rappresentati con un doppio cerchio.

Il modello di calcolo degli automi è riconducibile al concetto di *linguaggio regolare*, che verrà formalizzato in seguito, segue ora una definizione formale di automa.

Definizione (DFA) : Un DFA (Deterministic Finite Automa) è una 5-tupla, $(Q, \Sigma, \delta, q_0, F)$ di cui

- Q è l'insieme degli stati possibili
- Σ è l'alfabeto che compone le stringhe in input

- δ è una mappa $Q \times \Sigma \to Q$ detta *funzione di transizione.*
- $q_0 \in Q$ è lo stato iniziale.
- $F \subseteq Q$ è l'insieme degli stati di accettazione.

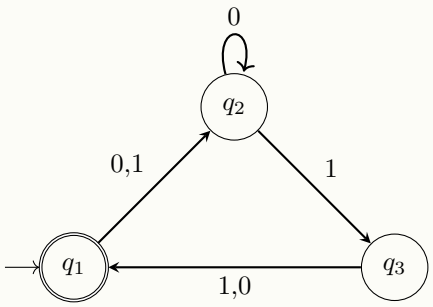


Figura 1.1: semplice automa

Nell'esempio in figura 1.1, si ha che

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- $F = \{q_1\}$
- $q_0 = q_1$
- $\delta =$

	0	1
q_1	q_2	q_2
q_2	q_2	q_3
q_3	q_1	q_1

Sia D un DFA, chiamiamo **linguaggio dell'automa**, e denotiamo $L(D)$, l'insieme delle stringhe che date in input a D fanno si che D termini su uno stato di accettazione. Per definire formalmente un linguaggio

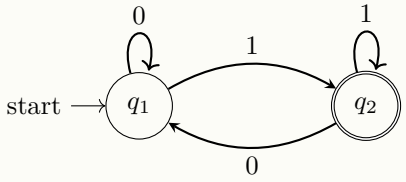


Figura 1.2: il linguaggio di tale automa risulta essere composto dalle stringhe che terminano con 1

di un automa, è necessario introdurre la **funzione di transizione estesa**:

$$\delta^*(q, \epsilon) = \delta(q, \epsilon)$$

$$\delta^*(q, ax) = \delta^*(\delta(q, a), x)$$

dove

$$a \in \Sigma, \quad x \in \Sigma^*, \quad \epsilon = \text{ stringa vuota}$$

Σ^* è l'insieme di tutte le stringhe formate dall'alfabeto Σ. Passiamo ora alla definizione di **configurazione**, essa rappresenta lo stato dell'automa ad un certo punto della computazione, è formata da una coppia

$$Q \times \Sigma^*$$

Rappresentante uno stato, ed una stringa di input rimanente da computare.

Un **passo della computazione** in un automa rappresenta una transizione da una configurazione ad un altra, è una relazione binaria $\vdash_D : Q \times \Sigma^*$ tale che

$$(p, ax) \vdash_D (q, x) \iff \delta(p, a) = q \quad \text{dove} \quad p, q \in Q, \;\; a \in \Sigma, \;\; x \in \Sigma^*$$

Si può estendere la definizione di passo di computazione, considerando la sua *chiusura transitiva* $\vdash_D^*$. Essa si ottiene aggiungendo a $\vdash_D$ tutte le coppie in $Q \times \Sigma^*$ che rendono la relazione chiusa rispetto la riflessività e rispetto la transitività.

$$(q, aby) \vdash_D (p, by) \;\wedge\; (p, by) \vdash_D (r, y) \implies (q, aby) \vdash_D^* (r, y)$$

Ad esempio, nell'automa in figura 1.2, risulta chiaro che

$$\begin{cases} (q_1, 011) \vdash_D (q_1, 11) \\ (q_1, 11) \vdash_D (q_2, 1) \\ (q_2, 1) \vdash_D (q_2, \epsilon) \end{cases} \implies (q_1, 011) \vdash_D^* (q_2, \epsilon)$$

Inoltre

$$\begin{gathered} \delta^*(q_1, 011) = \\ \delta^*(q_1, 11) = \\ \delta^*(q_2, 1) = \\ \delta^*(q_2, \epsilon) = q_2 \end{gathered}$$

Se non specificato diversamente, con ϵ verrà indicata la stringa vuota. Utilizzando le precedenti definizioni, è possibile definire formalmente quali sono gli input accettati da un DFA.

Definizione : Sia $D = (Q, \Sigma, \delta, q_0, F)$ un DFA, e sia $x \in \Sigma^*$ una stringa, essa è **accettata** da D se

$$\delta^*(q_0, x) \in F$$

Il **linguaggio riconosciuto** da D è

$$\text{☞} \quad L(D) = \{x \in \Sigma^* | \delta^*(q_0, x) \in F\} \quad \text{☜}$$

Definizione (Linguaggi Regolari) : L'insieme dei linguaggi regolari, denotato REG, contiene tutti i linguaggi, tali che esiste un DFA che li ha come linguaggi riconosciuti.

$$\text{☞} \quad REG = \{L \mid \exists D = (Q, \Sigma, \delta, q_0, F) \;\; \texttt{t.c.} \;\; L \subseteq \Sigma^* \wedge L(D) = L\} \quad \text{☜}$$

Uno fra gli scopi di questo corso riguarda il capire come progettare automi, e capire se, ogni linguaggio è regolare, o ce ne sono alcuni che non possono essere riconosciuti da alcun possibile DFA.

1.1.1 Esempi di DFA

Vediamo in questa sezione alcuni semplici esempi di DFA.

Esempio 1) Si vuole progettare un DFA che accetti il seguente linguaggio

$$\{x \in \{0, 1\}^* \mid w_h(x) \geq 3\}$$

Si ricordi come

$$w_h(x) = \text{ occorrenze di 1 in } x$$

Una volta progettato il DFA, è anche importante dimostrarne la correttezza, ossia dare una prova matematica che l'automa in questione accetti il linguaggio.

- Se $x \in L(D)$ allora D accetta x
- Se D accetta x allora $w_h(x) \geq 3$

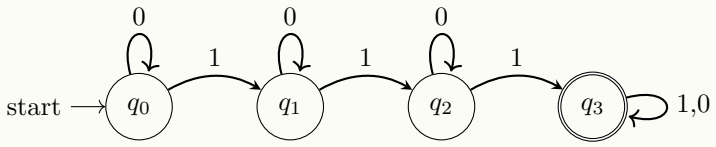


Figura 1.3: Esempio (1) di DFA

In questo, e nei seguenti casi, essendo i DFA estremamente semplici, risulta ovvio che accettino il dato linguaggio, in casi più avanzati, sarà necessario fornire una dimostrazione rigorosa.

Esempio 2) Si vuole progettare un DFA che accetti il seguente linguaggio

$$\{x \in \{0,1\}^* \mid x = 1y \wedge y \in \{0,1\}^*\}$$

Appunto sulla notazione : Se $a \in \Sigma^*$ e $b \in \Sigma^*$, allora con ab si denota la concatenazione di stringhe.

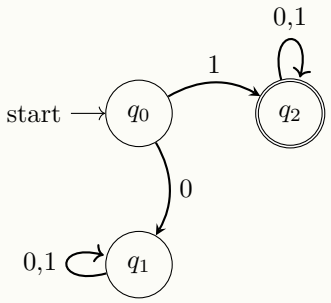


Figura 1.4: Esempio (2) di DFA

Nell'esempio (2), quando dallo stato q_0 il DFA riceve in input 0, la computazione cade su uno stato "buco nero", dalla quale non si può uscire a prescindere dall'input, l'operazione che fa cadere in questo stato è da considerarsi "non definita" in quanto non porterà mai la computazione a terminare su uno stato accettabile, è quindi comodo rimuovere tale stato dal diagramma.

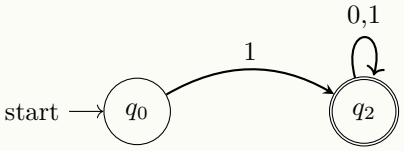


Figura 1.5: Esempio (2.1) di DFA

Anche in questo caso la dimostrazione della correttezza risulta banale.

Esempio 3) Si vuole progettare un DFA che accetti il seguente linguaggio

$$\{x \in \{0,1\}^* \mid x = 0^n 1, \;\; n \in \mathbb{N}\}$$

Con $0^n 1$ si intende una stringa che sia composta esclusivamente da 0, ma con un 1 come ultimo termine, ad esempio :

0000000000000001

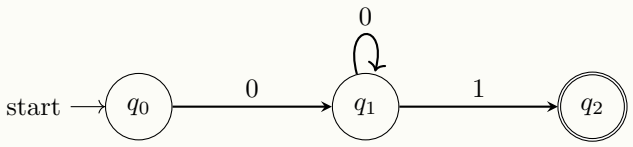


Figura 1.6: Esempio (3) di DFA

1.2 Operazioni sui Linguaggi

Lo studio delle proprietà dei linguaggi regolari può fornire opportune accortezze utili nella progettazione di automi, siccome i linguaggi sono insiemi di stringhe costruiti su un alfabeto Σ, essi godono delle operazioni insiemistiche.

Risulta utile definire formalmente la concatenazione fra stringhe, siano

$$x = a_1, a_2 \ldots, a_n \qquad y = b_1, b_2 \ldots, b_n$$

due stringhe, esse possono essere concatenate

$$xy = a_1, a_2 \ldots, a_n, b_1, b_2 \ldots, b_n$$

L'operazione di concatenazione non è commutativa, può essere definita ricorsivamente in tal modo :

$$x(ya) = (xy)a$$

dove

$$x, y \in \Sigma^* \qquad a \in \Sigma$$

Siano L_1, L_2 due linguaggi regolari in REG (per semplicità, definiti su uno stesso alfabeto Σ), e sia n un numero naturale, sono definite su di essi le seguenti operazioni :

- **unione** : $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$
- **intersezione** : $L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$
- **complemento** : $\neg L_1 = \{x \in \Sigma^* \mid x \notin L_1\}$
- **concatenazione** : $L_1 \circ L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$
- **potenza** : $L_1^n = \underbrace{L_1 \circ L_1 \circ L_1, \cdots \circ L_1}_{n \text{ volte}}$
- **star** : $L_1^* = \{x_1, x_2 \ldots, x_k \mid k \in \mathbb{Z}^+ \wedge x_i \in L_1\}$
  Si può definire anche diversamente
  $$L_1^* = \bigcup_{k=0}^{\infty} L_1^k$$

Esempio di concatenazione e potenza :

$$\Sigma = \{a, b\} \quad L_1 = \{a, ab, ba\} \quad L_2 = \{ab, b\} \quad L = \{a, ab, ba\}$$

$$L_1 \circ L_2 = \{aab, ab, abab, abb, baab, bab\}$$

$$L^2 = \{aa, aab, aba, abab, abba, baa, baba\}$$

Teorema (Chiusura di *REG*) : La classe dei linguaggi regolari REG, è chiusa rispetto a tutte le operazioni appena elencate, siano L_1 ed L_2 due linguaggi regolari, allora :

$$L_1 \cup L_2 \in REG \qquad L_1 \circ L_2 \in REG \qquad L_1 \cap L_2 \in REG$$

$$L_1^n \in REG \qquad \neg L_1 \in REG \qquad L_1^* \in REG$$

Dimostrazione (unione ed intersezione) : Siano L_1 ed L_2 due linguaggi regolari, considero due DFA, per semplicità, con lo stesso alfabeto

$$D_1 = (Q_1, \Sigma, \delta, q_1, F_1)$$

$$D_2 = (Q_2, \Sigma, \delta, q_2, F_2)$$

tali che

$$L(D_1) = L_1 \ \wedge \ L(D_2) = L_2$$

Si costruisce un DFA che simula contemporaneamente l'esecuzione di D_1 e D_2, in cui gli stati possibili saranno le possibili combinazioni di coppie di stati. Si definisce $D = (Q, \Sigma, \delta, q_0, F)$ tale che

- $Q = Q_1 \times Q_2 = \{(r_1, r_2) \| \ r_1 \in Q_1 \wedge r_2 \in Q_2\}$
- $\delta((r_1, r_2), a) = (\delta(r_1, a), \delta(r_2, a))$ dove $a \in \Sigma$ e $(r_1, r_2) \in Q$
- $q_0 = (q_1, q_2)$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2) = \{(r_1, r_2) \| \ r_1 \in F_1 \vee r_2 \in F_2\}$

Nel caso si dovesse dimostrare la proprietà dell'intersezione, si avrebbe che

- $F = F_1 \times F_2$

A questo punto risulta chiaro che

(i) $x \in L_1 \cup L_2 \implies x \in L(D)$

(ii) $x \in L(D) \implies x \in L_1 \cup L_2$ ■

Dimostrazione (complemento) : : Sia L un linguaggio regolare, e D un automa che lo accetta $L(D) = L$. Si vuole dimostrare che esiste un automa che accetti $L^C = \{w \in \Sigma^* \mid w \notin L\}$. Essendo

$$D = (Q, \Sigma, \delta, q_0, F)$$

considero

$$D' = (Q, \Sigma, \delta, q_0, F^C)$$

dove $F^C = Q \backslash F$. Supponiamo che $w \in L^C$, allora sicuramente, se data come input a D', la computazione terminerà in uno stato che non è in F, dato che, per definizione, se terminasse in F, sarebbe accettato da D, ma L^C contiene tutte le stringhe che non sono accettate da D, quindi

- $w \in L^C$
- la computazione termina in uno stato $q \in F^C$
- D' accetta w
- L^C è un linguaggio regolare. ■

Per dimostrare la proprietà di concatenazione, è necessario introdurre un nuovo concetto.

1.3 Non Determinismo

Si può generalizzare il modello di DFA, in modo tale che la lettura di un input non scaturisca il passaggio da uno stato ad un'altro, ma da uno stato ad un insieme di stati, tale generalizzazione è detta *Non-Deterministic Finite Automa.*

Definizione (NFA) : Un NFA è una tupla $N = (Q, \Sigma, \delta, q_0, F)$ tale che

- Q, Σ, q_0, F condividono la definizione con i loro corrispettivi nel DFA
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$

- $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$
- $\mathcal{P}(Q)$ è l'insieme delle parti di Q

Una computazione in un NFA è paragonabile ad una computazione parallela, in cui un input può risultare in diversi *rami* di computazione. Una funzione di transizione di un NFA inoltre accetta la stringa vuota ϵ, se la computazione finisce in uno stato in cui è presente un arco di questo tipo, verrà considerata anche una diramazione verso quell'arco a prescindere dall'input.

Una computazione si può rappresentare graficamente con un albero, se una delle diramazioni possibili termina in uno stato accettabile, allora la stringa in input è accettata.

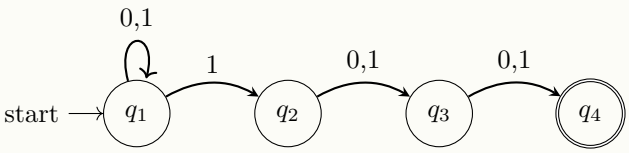


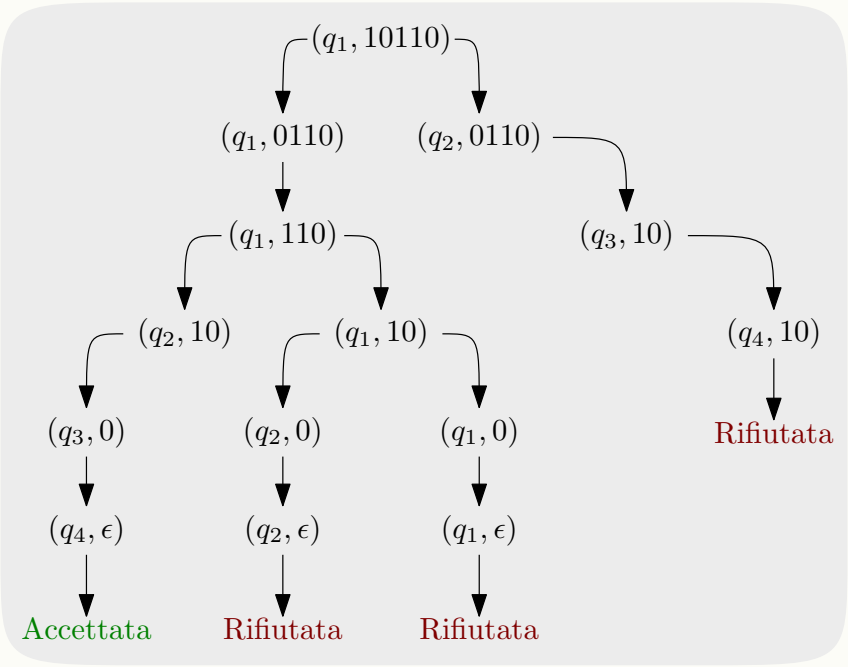
Figura 1.7: Esempio di NFA

Sia N l'NFA mostrato in figura 1.7, si ha

$$L(N) = \{x \in \{0,1\}^* \| \ x \text{ ha 1 come terz'ultimo valore } \}$$

Si può visualizzare il seguente albero di computazione data come input una stringa w :

$$w = 10110$$



Essendo che la traccia di sinistra accetta w, allora N accetta w.

È necessario estendere il concetto di *configurazione* per gli NFA, essa, rappresentante uno stato della computazione, sarà una coppia

$$(q, x) \in Q \times \Sigma_\epsilon$$

E diremo che

$$(p, ax) \vdash_N (q, x) \iff q \in \delta(p, a)$$

dove

$$p, q \in Q \quad a \in \Sigma_\epsilon \quad x \in \Sigma_\epsilon^*$$

Si considera ora la chiusura transitiva di $\vdash_N$, denotata $\vdash_N^*$, se w è una stringa ed N un NFA, si ha che

$$w \in L(N) \iff \exists q \in F \text{ tale che } (q_0, w) \vdash_N^* (q, \epsilon)$$

Consideriamo adesso l'unione di due NFA, risulta particolarmente semplice da definire, siano N_1 e N_2 due NFA, che per semplicità, condividono l'alfabeto

$$N_1 = \{Q_1, \Sigma_\epsilon, \delta_1, q_1, F_1\}$$

$$N_2 = \{Q_2, \Sigma_\epsilon, \delta_2, q_2, F_2\}$$

Definisco un nuovo NFA $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$ tale che

- $Q = Q_1 \cup Q_2$
- $F = F_1 \cup F_2$
- Siano $q \in Q$ e $a \in \Sigma_\epsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) \text{ se } q \in Q_1 \\ \delta_2(q,a) \text{ se } q \in Q_2 \\ \{q_1, q_2\} \text{ se } q = q_0 \ \wedge \ a = \epsilon \\ \emptyset \text{ se } q = q_0 \ \wedge \ a \neq \epsilon \end{cases}$$

Si avrà che $L(N) = L(N_1) \cup L(N_2)$

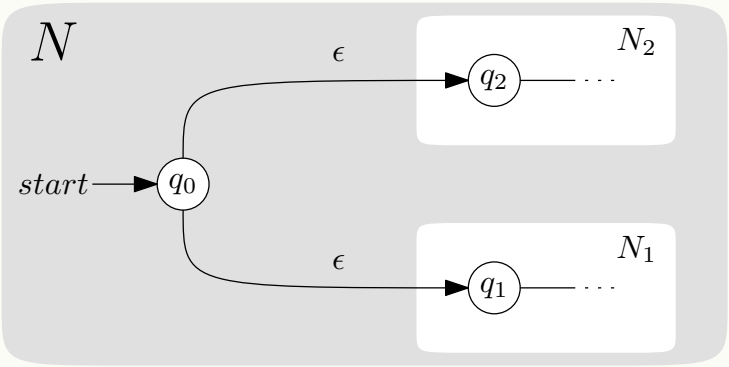


Figura 1.8: Unione di due NFA

Denotiamo $\mathcal{L}(DFA)$ l'insieme dei linguaggi accettati da un qualsiasi DFA, che per definizione è REG, e denotiamo, in maniera analoga $\mathcal{L}(NFA)$.

Teorema : L'insieme dei linguaggi accettati da un qualsiasi DFA, e quello dei linguaggi accettati da un qualsiasi NFA coincidono

$$\mathcal{L}(DFA) = \mathcal{L}(NFA) = REG$$

Dimostrazione : Il caso $\mathcal{L}(DFA) \subseteq \mathcal{L}(NFA)$ è banale e non verrà dimostrato. Si vuole dimostrare che se L è accettato da un generico NFA, allora esiste un DFA che accetta L, l'idea è quella di "simulare" un NFA tramite un DFA che rappresenti ogni possibile stato di computazione.

Sia $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$ un NFA, e sia $L = L(N)$. Si considera un DFA $D = (Q_D, \Sigma, \delta_D, q_{0_D}, F_D)$ tale che

- $Q_D = \mathcal{P}(Q_N)$
- $q_{0_D} = \{q_{0_N}\}$
- $F_D = \{R \in Q_D | R \cap F_N \neq \emptyset\}$, ovvero, D accetta tutti gli insiemi in cui compare almeno un elemento accettato da N.

- Sia $R \in Q_D$ e $a \in \Sigma$, si definisce $\delta_D(R, a) = \bigcup_{r \in R} \delta_N(r, a)$

Questo caso non tiene conto di un NFA in cui sono presenti degli ϵ-archi. Supponiamo che vi siano, sia $R \in Q_D$, si definisce la funzione estesa E definita come segue

$$E(R) = \{q \in Q_N \| \ q \text{ può essere raggiunto da un qualsiasi stato } r \in R \text{ attraverso zero o più } \epsilon\text{-archi } \}$$

Cambia la definizione del DFA utilizzato per la dimostrazione

- $q_{0_D} = E(\{q_{0_N})\}$
- $\delta_D(R, a) = \bigcup_{r \in R} E(\delta_N(r, a))$

È chiaro che D tiene traccia di tutte le possibili computazioni di N, ed accetta L, ossia $L(D) = L(N)$. ■

Esempio : Si consideri l'NFA N definito come segue

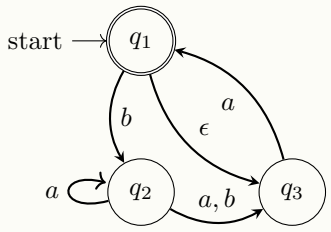


Figura 1.9: $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$

Si costruisce un DFA $D = (Q_D, \Sigma, \delta_D, q_{0_D}, F_D)$ con le seguenti specifiche (per comodità, l'elemento $\{q_i, q_j \ldots, q_k\}$ verrà denotato $p_{ij\ldots k}$), mostrato in figura 1.10

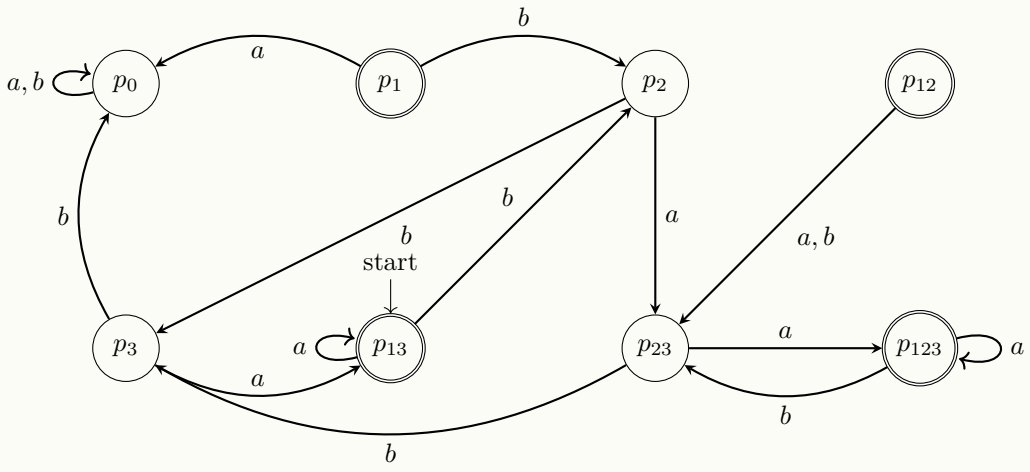


Figura 1.10: grafico di D

- $Q_D = \{p_0, p_1, p_2, p_3, p_{12}, p_{13}, p_{23}, p_{123}\}$
- $E(q_{0_N}) = \{q_1, q_3\} \implies q_{0_D} = p_{13}$
- $F_D = \{p_1, p_{12}, p_{13}, p_{123}\}$
- la funzione δ_D si definisce osservando il grafico di N
  - $\delta_N(q_2, a) = \{q_2, q_3\} \implies \delta_D(p_2, a) = p_{23}$
  - $\delta_N(q_2, b) = \{q_3\} \implies \delta_D(p_2, a) = p_3$

- $\delta_N(q_1, a) = \emptyset \implies \delta_D(p_1, a) = p_0$
- $\delta_N(q_1, b) = \{q_2\} \implies \delta_D(p_1, b) = p_2$
- etc...

L'introduzione degli automi non deterministici è stata necessaria in principio per la dimostrazione della chiusura di *REG* rispetto le operazioni di concatenazione e star.

Teorema : *REG* è chiusa per concatenazione.

Dimostrazione : Siano L_1 ed L_2 due linguaggi regolari, esistono quindi due NFA

$$N_1 = (Q_1, \Sigma_\epsilon, \delta_1, q_0^1, F_1) \qquad N_2 = (Q_2, \Sigma_\epsilon, \delta_2, q_0^2, F_2)$$

tali che $L(N_1) = L_1 \ \wedge \ L(N_2) = L_2$. Si costruisce un NFA $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$, l'idea è quella di concatenare le ramificazioni di N_1 ad N_2.

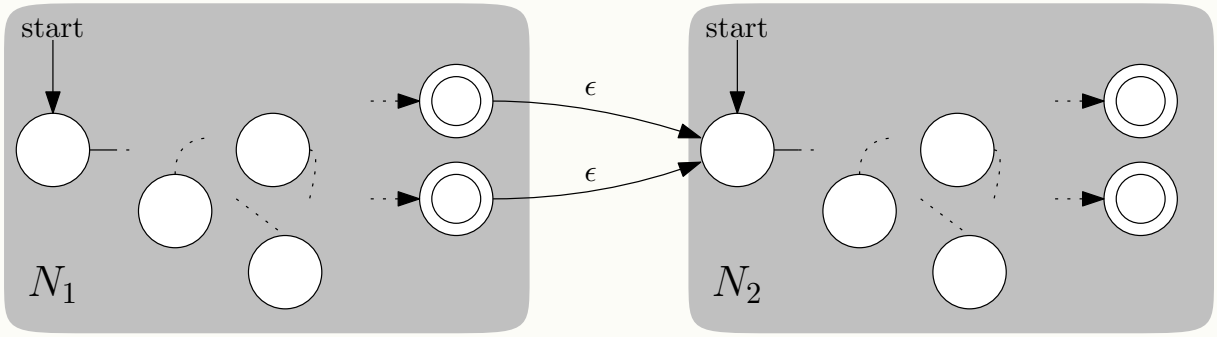


Figura 1.11: schema di N

Un NFA di questo tipo computerà una stringa in L_1, se finirà in uno stato di F_1, andrà nello stato iniziale di N_2, è chiaro che l'automa accetta solamente una concatenazione di stringhe fra L_1 ed L_2.

- $Q = Q_1 \cup Q_2$
- $q_0 = q_0^1$
- $F = F_2$
- per $a \in \Sigma_\epsilon$ e $q \in Q$ si ha $\delta(q, a) = \begin{cases} \delta_1(q, a) \text{ se } q \in Q_1 \wedge q \notin F_1 \\ \delta_1(q, a) \text{ se } q \in F_1 \wedge a \neq \epsilon \\ \delta_1(q, a) \cup \{q_0^2\} \text{ se } q \in F_1 \wedge a = \epsilon \\ \delta_2(q, a) \text{ se } q \in Q_2 \end{cases}$

Si ha quindi che $L(N) = L_1 \circ L_2$. ■

Teorema : *REG* è chiusa per star.

Dimostrazione : Sia $L \in REG$ e sia $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$ un NFA tale che $L(N) = L$. Considero un NFA $N^* = (Q^*, \Sigma_\epsilon, \delta^*, q_0^*, F^*)$, identico ad N, con opportune modifiche, lo stato q_0 iniziale di N non è iniziale in N^*, ed ogni stato finale ha una ϵ-arco verso q_0.

- $Q^* = Q \cup \{q_0^*\}$
- q_0^* è un nuovo stato
- $F^* = F \cup \{q_0^*\}$ questo perché in L^* è presente la stringa vuota

- per $a \in \Sigma_\epsilon$ e $q \in Q^*$ si ha $\delta^*(q,a) = \begin{cases} \delta(q,a) \text{ se } q \in Q \wedge q \notin F \\ \delta(q,a) \text{ se } q \in F \wedge a \neq \epsilon \\ \delta(q,a) \cup \{q_0\} \text{ se } q \in F \wedge a = \epsilon \\ \{q_0\} \text{ se } q = q_0^* \wedge a = \epsilon \\ \emptyset \text{ se } q = q_0^* \wedge a \neq \epsilon \end{cases}$

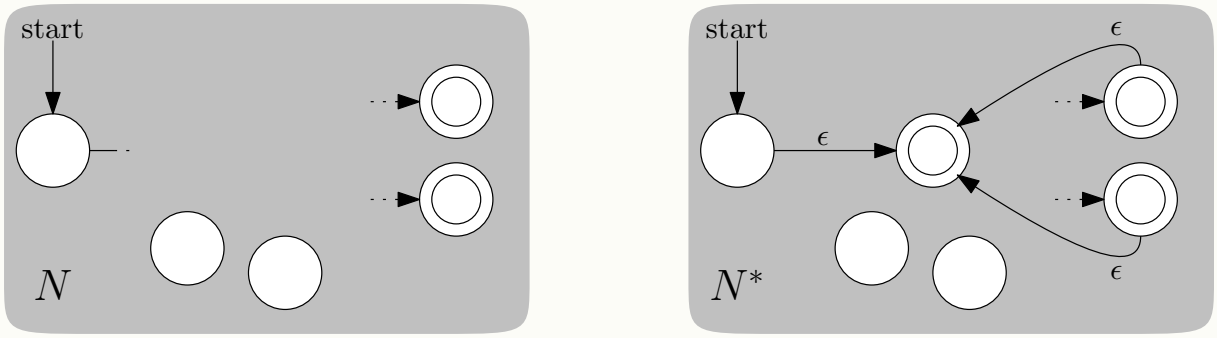


Figura 1.12: schema di N^*

1.4 Espressioni Regolari

Un espressione regolare è simile ad un espressione algebrica ma opera sulle stringhe, dato un alfabeto, un'espressione su tale alfabeto rappresenta un insieme di stringhe, un esempio è

$$(0|1)0^*$$

Dove $(0|1) \equiv \{0\} \cup \{1\} = \{0,1\}$ e $0^* \equiv \{0\}^*$ quindi $(0|1)0^* \equiv \{0,1\} \circ \{0\}^*$.

Definizione (espressione regolare) : Sia Σ un alfabeto, un espressione regolare r su Σ, denotata $r \in re(\Sigma)$, è definita per induzione

Caso base

- $r = \emptyset \in re(\Sigma)$
- $r = \epsilon \in re(\Sigma)$
- $r = a \in re(\Sigma)$ dove $a \in \Sigma$

Caso induttivo

- $r = r_1 \cup r_2$ dove $r_1, r_2 \in re(\Sigma)$
- $r = r_1 \circ r_2$ dove $r_1, r_2 \in re(\Sigma)$
- $r = r_1^*$ dove $r_1 \in re(\Sigma)$

L'insieme delle stringhe definite da $r \in re(\Sigma)$ è il *linguaggio* di r ed è denotato $L(r)$.

Esempio : Sia $\Sigma = \{0,1\}$

- $0^*10^* = \{w \mid w \text{ ha esattamente un } 1\}$
- $\Sigma^*1\Sigma^* = \{w \mid w \text{ ha almeno un } 1\}$
- $\Sigma^*001\Sigma^* = \{w \mid w \text{ ha la sottostringa } 001\}$

Per convenzione si definisce

$$1^*\emptyset = \emptyset \qquad \emptyset^* = \epsilon$$

Teorema Fondamentale : Sia $\mathcal{L}(DFA) = REG$ l'insieme dei linguaggi accettati da un qualsiasi DFA, e sia $\mathcal{L}(re)$ l'insieme dei linguaggi accettati da una qualsiasi espressione regolare, è vero che

$$\mathcal{L}(re) = \mathcal{L}(DFA) = REG$$

Dimostrazione : è necessario dimostrare due direzioni

$\boxed{\mathcal{L}(re) \subseteq \mathcal{L}(DFA)}$: Sia r un espressione regolare, si considera un DFA D_r definito come segue, a seconda dei casi

Caso base

- $r = \emptyset \implies D_r$ non accetta alcuna stringa
- $r = \epsilon \implies D_r$ accetta la stringa vuota
- $r = a \implies D_r$ accetta $a \in \Sigma$

Caso induttivo

- $r = r_1 \cup r_2$, esistono due automi D_{r_1} e D_{r_2} che accettano rispettivamente $L(r_1)$ e $L(r_2)$, ma allora esiste necessariamente un automa D_r che accetta $L(r_1) \cup L(r_2)$.
- $r = r_1 \circ r_2$, esistono due automi D_{r_1} e D_{r_2} che accettano rispettivamente $L(r_1)$ e $L(r_2)$, ma allora esiste necessariamente un automa D_r che accetta $L(r_1) \circ L(r_2)$.
- $r = r_1^*$, esiste un automa D_{r_1} che accetta $L(r_1^*)$, ma allora esiste necessariamente un automa D_r che accetta $L(r_1)$.

Tali tesi sono vere dato che la classe dei linguaggi regolari è chiusa per le operazioni di star, concatenazione ed unione. □

$\boxed{\mathcal{L}(DFA) \subseteq \mathcal{L}(re)}$: Sia L un linguaggio regolare, e sia N l'NFA tale che $L(N) = L$. Si costruisce un nuovo tipo di NFA che sarà equivalente ad N. Tale automa è detto *GNFA*, dove la G sta per "Generalizzato", tale automa ha una *forma canonica*, ossia, rispetta le seguenti proprietà

- Lo stato iniziale, ha solo archi uscenti
- Vi è un singolo stato finale, ed ha solo archi entranti
- Per ogni coppia di stati (non necessariamente distinti), c'è esattamente un arco.
- Ogni arco è etichettato da un espressione regolare.

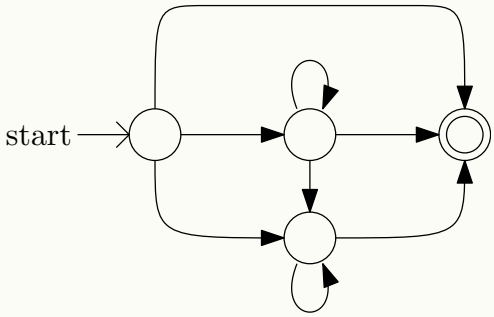


Figura 1.13: forma di un GNFA

Più precisamente, sia G un GNFA definito $G = (Q, \Sigma, \delta, q_{start}, q_{acc})$ dove

$$\delta : Q\backslash\{q_{acc}\} \times Q\backslash\{q_{start}\} \to re(\Sigma)$$

Dato un generico NFA, è possibile trasformarlo in un GNFA aggiungendo al più due stati (iniziale e finale), ed utilizzando gli ϵ-archi per riempire le coppie di stati che non sono collegate.

La funzione $Convert : GNFA \to GNFA$ modifica un GNFA restituendone uno equivalente, ma con uno stato in meno. Tale funzione è definita in tal modo, sia k il numero di stati/nodi di G, si esegue $Convert(G)$

- Se $k = 2$, allora esiste un solo arco fra questi etichettato con un espressione regolare r, la funzione restituirà r.
- Se $k > 2$, viene selezionato un qualsiasi nodo in $Q\backslash\{q_{start}, q_{acc}\}$, sia questo q_{rip}, si avrà $Convert(G) = G' = (Q\backslash\{Q_{rip}\}, \Sigma, \delta', q_{start}, q_{acc})$ dove

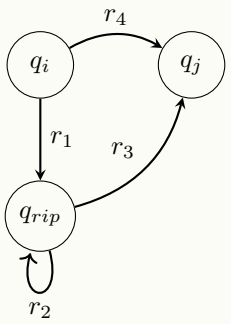
  $$\delta' : Q\backslash\{q_{acc}, q_{rip}\} \times Q\backslash\{q_{start}, q_{rip}\} \to re(\Sigma)$$

  Inoltre ogni etichetta di G' viene aggiornata secondo la seguente procedura, siano $q_i \in Q\backslash\{q_{acc}, q_{rip}\}$ e $q_j \in Q\backslash\{q_{start}, q_{rip}\}$ due stati qualsiasi

  $$\delta'(q_i, q_j) = (r_1 r_2^* r_3)|r_4$$

  Dove

  - $r_1 = \delta(q_i, q_{rip})$
  - $r_2 = \delta(q_{rip}, q_{rip})$
  - $r_3 = \delta(q_{rip}, q_j)$
  - $r_4 = \delta(q_i, q_j)$



Bisogna ora dimostrare che un generico GNFA G è equivalente a $Convert(G)$. Si dimostra per induzione su k numero di stati.

caso base $k = 2$: In tal caso la procedura $Convert$ restituisce l'espressione regolare r sull'unico arco che descrive ogni stringa accettata da G. $L(r) \equiv L(G)$

passo induttivo : si assume che G è equivalente a $Convert(G)$ per $k - 1$ stati.

- Se G accetta w, allora esiste un ramo di computazione $C = \{q_{start}, q_1 \ldots, q_{accept}\}$, se q_{rip} che è stato rimosso in $G' = Convert(G)$ non appartiene a C, allora la computazione non è alterata e G' accetta w, altrimenti ci sarà una differente sequenza di stati, ma gli stati q_i, q_j adiacenti a q_{rip} sono ora uniti da un arco etichettato da un espressione regolare che comprende le stringhe per andare da q_i a q_j passando per q_{rip}.
- Se G' accetta w, anche G lo accetta dato che per ogni coppia di stati in C si è aggiornata l'etichetta tenendo conto della transazione che porta da uno stato all'altro passando per q_{rip}.

Quindi $Convert$ restituisce un automa equivalente con $k - 1$ stati, quindi l'asserto è vero. ■

1.4.1 Esempi

Esempio 1) Si trasformi $r = (ab|a)^*$ in un NFA.

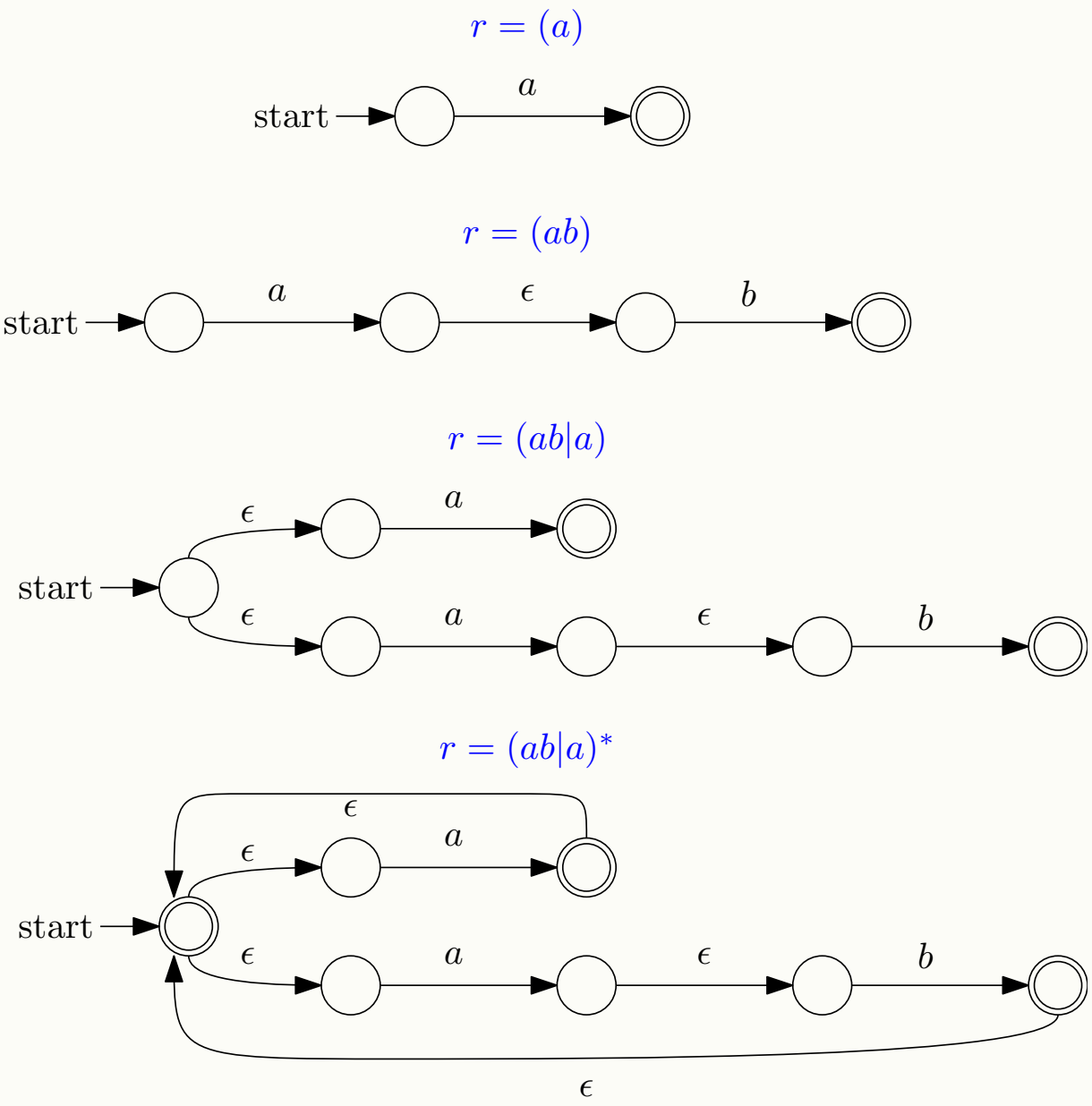
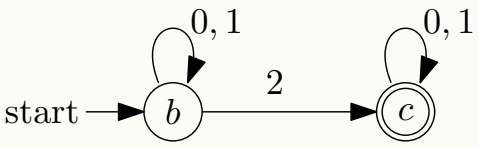
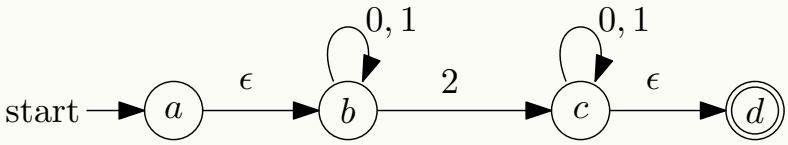


Figura 1.14: Esempio 1

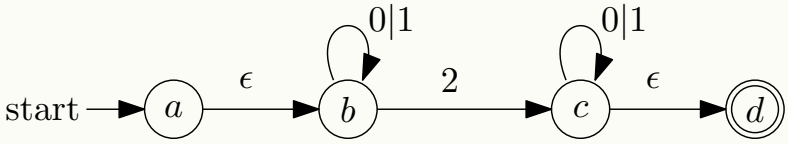
Esempio 2) Dato il seguente automa, si trovi l'espressione regolare associata



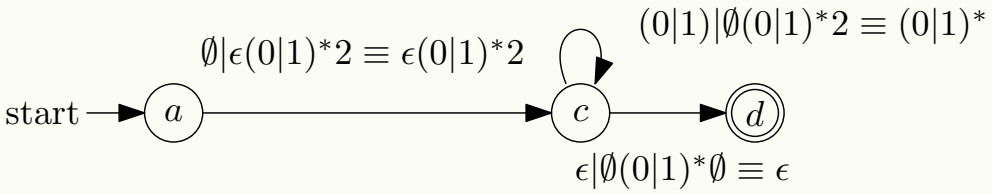
passo 1 : si trasforma in forma canonica (ignorati gli archi etichettati con $\emptyset$)



passo 2 : si trasformano le etichette in espressioni regolari



passo 3 : si rimuove b



passo 4 : si rimuove b

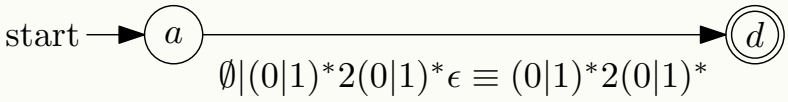


Figura 1.15: Esempio 2

1.5 Linguaggi non regolari

1.5.1 Il Pumping Lemma per i Linguaggi Regolari

A questo punto della lettura, è naturale porsi una domanda : Tutti i linguaggi sono regolari? Esistono linguaggi che non possono essere accettati da alcun DFA? Nel caso solamente un sottoinsieme dei linguaggi fosse regolare, quali proprietà soddisfa? Si consideri il seguente linguaggio

$$L = \{0^n 1^n \mid n \geq 0\}$$

Si può provare a disegnare un automa che accetti L, rendendosi ben presto conto che è *impossibile*, L non è regolare, esistono quindi dei linguaggi che non sono regolari. L'automa a stati finiti è un modello semplice, non può "ricordare" quanti caratteri di un certo tipo sono stati letti.

Essendo che solamente alcune stringhe possono essere accettate da un qualsiasi automa, è importante caratterizzare tali stringhe e definirne le proprietà in tal merito.

Appunto sulla notazione : Se w è una stringa, allora $|w|$ è il numero dei suoi caratteri.

Osservzione : Se un DFA con n stati legge una stringa di $k > n$ caratteri, allora ci sarà almeno uno stato che verrà considerato due volte durante la computazione.

Teorema (Pumping Lemma) : Sia L un linguaggio regolare, sia D l'automa tale che $L(D) = L$, si considera una stringa $w \in L(D)$, ed una sua decomposizione in 3 stringhe concatenate $w = xyz$. Esiste un intero $p \leq |w|$, denotato *pumping* tale che

1. $\forall i \geq 0$, $xy^i z \in L(D)$
2. $|y| > 0$
3. $|xy| \leq p$

Dimostrazione : Sia $D = (Q, \Sigma, \delta, q_{start}, F)$ un automa, e sia $p = |Q|$. Sia w una stringa su Σ di $n \geq p$ caratteri definita $w = w_1 w_2 \dots w_n$. Sia $\{r_1, r_2 \dots, r_{n+1}\}$ la sequenza di stati che D computa su input w, ossia

$$\delta(r_i, w_i) = r_{i+1}$$

Tale sequenza è lunga $n+1 \geq p+1$ stati, fra i primi $p+1$ elementi c'è necessariamente uno stato ripetuto, sia r_j la prima occorrenza di tale stato, e sia r_l la seconda.

Siccome la ripetizione avviene fra le prime $p+1$ computazioni, si ha che $l \leq p+1$. Si consideri la seguente scomposizione di w

- $x = w_1, w_2 \dots, w_{j-1}$
- $y = w_j, w_{j+1} \dots, w_{l-1}$
- $z = w_l, w_{l+1} \dots, w_n$

1. $xy^i z \in L(D)$ perché x parte da $r_1 = q_{start}$ e arriva a r_j, y^i parte da r_j e ritorna su r_l, che è lo stesso stato, e z porta da r_l allo stato finale di accettazione.
2. Essendo che $j < l$, allora la dimensione minima di y è 1, in quanto il valore minimo che può assumere l è $j + 1$, ne consegue che $|y| > 0$.
3. $l \leq p + 1$ ovvero $l - 1 = |xy| \leq p$.

I tre punti sono dimostrati. ■

Proposizione : Se L è un linguaggio regolare, ed L^* un sottoinsieme di L, allora L^* non è necessariamente regolare.

Alcuni esercizi al seguente link : Esercizi Linguaggi Regolari.

1.6 Grammatiche Acontestuali

Lo scopo delle grammatiche acontestuali è quello di estendere l'automa a stati finiti per ottenere un modello di computazione più potente. Tale automa al quale corrispondono le gramamtiche è detto *PDA*. Le grammatiche hanno applicazioni fondamentali, nei linguaggi di programmazione, precisamente, nel funzionamento dei compilatori.

Una definizione informale può essere la seguente : Una grammatica è composta da un insieme di *regole* su un alfabeto e delle variabili, tali regole sono annotate come segue

$$\begin{cases} A \longrightarrow 0A1 \\ A \longrightarrow B \\ B \longrightarrow \# \end{cases} \qquad \Sigma = \{0, 1, \#\}$$

Ciascuna regola contiene una variabile alla quale viene associata una stringa, composta da variabili e *terminali*, ossia i caratteri dell'alfabeto Σ. Una variabile è considerata iniziale, e per convenzione, è sempre quella presente nella prima regola.

Precisamente, una grammatica può generare stringhe

1. Si scrive la variabile iniziale
2. Si sostituisce applicando una delle regole
3. Si ripete ricorsivamente il procedimento finché la stringa contiene solo terminali.

Esempio : Considerando la grammatica con le regole prima elencate :

1. A si applica $A \longrightarrow 0A1$
2. $0A1$ si applica $A \longrightarrow 0A1$
3. $00A11$ si applica $A \longrightarrow 0A1$
4. $000A111$ si applica $A \longrightarrow B$
5. $000B111$ si applica $B \longrightarrow \#$
6. $000\#111$

Applicando le regole secondo un ordine arbitrario, è possibile generare qualsiasi stringa del tipo $0^n\#1^n$, ad una grammatica quindi corrisponde un linguaggio. Una 'computazione' di una grammatica può essere rappresentata con un albero sintattico, mostrato in figura 1.16.

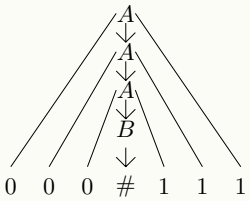


Figura 1.16: Albero Sintattico

Con una grammatica è anche possibile rappresentare una qualsiasi espressione algebrica

$$\begin{cases} E \longrightarrow E + E \\ E \longrightarrow E \times E \\ E \longrightarrow (E) \\ E \longrightarrow 0 \vee 1 \vee 2 \ldots, \vee 9 \end{cases} \qquad \Sigma = \{0, 1, 2, 3 \ldots, 9\}$$

Definizione (Grammatica Acontestuale) : Una **CFG** (Context Free Grammar) è una tupla $G = (V, \Sigma, R, S)$ dove

- V è un insieme di simboli dette variabili
- Σ è un insieme di simboli detti terminali
- $V \cap \Sigma = \emptyset$
- $S \in V$ è la variabile iniziale
- R è un insieme di regole

Le regole R possono essere rappresentate come una funzione $R : V \to (V \cup \Sigma)^*$, associa ad ogni variabile una stringa composta da terminali, variabili, o entrambe.

Sia uAv una stringa tale che $A \in V$, $u, v \in \Sigma \cup V$, e sia $w \in \Sigma \cup V$ diremo che uAv **produce** uwv, e denoteremo $uAv \Rightarrow uwv$ se e solo se

$$A \longrightarrow w \in R$$

Il simbolo $\Rightarrow$ rappresenta una relazione su $\Sigma \cup V$. Si può considerare la sua chiusura transitiva $\Rightarrow^*$

$$u \Rightarrow^* v \iff \exists\{u_1, u_2 \ldots, u_k\} \mid u \Rightarrow u_1 \Rightarrow u_2 \cdots \Rightarrow u_k \Rightarrow v$$

Sia $G = (V, \Sigma, R, S)$ una CFG, $L(G)$ è il **linguaggio della grammatica**, definito come segue

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Unione

Si consideri il seguente insieme di grammatiche

$$\{G_i = (V_i, \Sigma_i, R_i, S_i)\} \qquad i \in \{1, 2, 3 \ldots, n\}$$

È possibile considerare *l'unione delle grammatiche*, definita come segue

$$G = (\bigcup_i \{V_i\} \cup \{S\}, \bigcup_i \Sigma_i, R, S)$$

Dove

- S è una nuova variabile
- $R = \{\bigcup_i R_i\} \cup \{S \to S_1 \vee S \to S_2 \cdots \vee S \to S_n\}$

Proposizione : $L(G) = \bigcup_i L(G_i)$

Ambiguità

Una CFG soddisfa la proprietà di ambiguità se, una stessa stringa può essere generata seguendo sequenze di regole differenti. L'ambiguità di una grammatica può risultare problematica nelle applicazioni.

Definizione : Una stringa di una CFG ha una *derivazione a sinistra* se può essere ottenuta sostituendo ad ogni passo di produzione (applicazione delle regole) la variabile che si trova più a sinistra. Una stringa è *derivata ambiguamente* se ha 2 o più derivazioni a sinistra. Una CFG è ambigua se ha almeno una stringa derivata ambiguamente.

1.6.1 Forma Normale

In questa sezione verrà definita una forma canonica per le grammatiche, tale forma è fondamentale, soprattutto nelle applicazioni, se una grammatica è in tale forma, è possibile stabilire un tetto minimo di operazioni da eseguire per ottenere una determinata stringa.

Definizione : Una CFG $G = (V, \Sigma, R, S)$ è in **forma normale Chomsky** (o più comunemente, in forma normale) se ogni sua regola è della forma

$$A \longrightarrow BC$$
$$A \longrightarrow a$$

Dove

- $a \in \Sigma$
- $A, B, C \in V$
- $B \neq S, \quad C \neq S$
- La regola $S \longrightarrow \epsilon$ è permessa.

La variabile iniziale S non può mai essere nel termine destro di una regola.

Teorema : Per ogni CFG, esiste una CFG equivalente (che genera lo stesso linguaggio) in forma normale.

Dimostrazione : La dimostrazione consiste in una procedura, in cui vengono applicate delle trasformazioni alle regole di una generica CFG in modo che soddisfino la forma normale. Sia $G = (V, \Sigma, R, S)$ una CFG, non necessariamente in forma normale

- Si definisce una nuova variabile S_0, essa sarà considerata la nuova variabile iniziale della grammatica G, e verrà aggiunta la regola $S_0 \to S$, ciò garantisce che la variabile iniziale non compare mai come termine destro.
- Data una ϵ-regola, ossia le regole della forma

  $$A \longrightarrow \epsilon \qquad A \in V$$

  Si considera ogni occorrenza di A nel termine destro di una regola, e si definisce una nuova regola identica, dove A è assente. Infine, $A \longrightarrow \epsilon$ viene rimossa dalle regole

  $$\begin{cases} A \longrightarrow \epsilon \\ B \longrightarrow xAyA \end{cases} \implies \text{(diventa)} \begin{cases} B \longrightarrow xAyA \\ B \longrightarrow xyA \text{ (aggiunta)} \\ B \longrightarrow xAy \text{ (aggiunta)} \end{cases} \qquad \begin{matrix} B \in V \\ x, y \in (\Sigma \cup V)^* \end{matrix}$$

- Si considerano poi tutte le regole unitarie, ossia del tipo

  $$A \longrightarrow B$$

  con $B \in V$, una volta rimossa, per ogni altra regola $B \to u$, si aggiunge la regola $A \to u$, con $u \in (\Sigma \cup V)^*$, ripetendo ricorsivamente il procedimento.
- Infine si considerano le regole del tipo

  $$A \longrightarrow u_1 u_2 \ldots, u_k \qquad \begin{matrix} u_i \in (\Sigma \cup V) \\ k \geq 3 \end{matrix}$$

  Tale regola viene rimossa, e vengono spezzate in un set di regole come segue

  $$\begin{cases} A \longrightarrow u_1 A_1 & \text{nuova variabile } A_1 \text{ aggiunta a } V \\ A_1 \longrightarrow u_2 A_2 & \text{nuova variabile } A_2 \text{ aggiunta a } V \\ A_2 \longrightarrow u_3 A_3 & \text{nuova variabile } A_3 \text{ aggiunta a } V \\ \vdots & \\ A_{k-2} \longrightarrow u_{k-1} u_k & \text{nuova variabile } A_{k-2} \text{ aggiunta a } V \end{cases}$$

  Una volta fatto ciò, per ogni regola $A_i \longrightarrow u_j A_j$ in cui u_j è un terminale, si definisce una nuova variabile U_j ed una nuova regola $U_j \longrightarrow u_j$. Ad esempio, se u_1 è un terminale, viene rimossa $A \longrightarrow u_1 A_1$, e vengono definite le nuove regole

  $$\begin{cases} A \longrightarrow U_1 A_1 \\ U_1 \longrightarrow u_1 \end{cases}$$

Una volta eseguite le procedure, la nuova CFG sarà equivalente a quella originale, e sarà in forma normale Chomsky. ■

Esempio

Si consideri la seguente grammatica G

$$\begin{cases} S \longrightarrow ASA \vee aB \\ A \longrightarrow A \vee S \\ B \longrightarrow b \vee \epsilon \end{cases} \quad \text{si può scrivere anche} \quad \begin{cases} S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases}$$

- Passo 1 : variabile iniziale

$$\begin{cases} S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases}$$

- Passo 2 : ϵ-regole

$$\begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S \\ B \longrightarrow b|\epsilon \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S|\epsilon \\ B \longrightarrow b \end{cases}$$

$$\begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB \\ A \longrightarrow B|S|\epsilon \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB|SA|AS|S \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases}$$

- passo 3 : regole unitarie
  si rimuove $S_0 \longrightarrow S$

$$\begin{cases} S_0 \longrightarrow S \\ S \longrightarrow ASA|aB|SA|AS|S \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases}$$

  si rimuovono $A \longrightarrow B$ e $A \longrightarrow S$

$$\begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow B|S \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow b \\ A \longrightarrow ASA|aB|SA|AS \\ B \longrightarrow b \end{cases}$$

- passo 4 : convertire le ultime regole nella forma corretta

$$\begin{cases} S_0 \longrightarrow ASA|aB|SA|AS \\ S \longrightarrow ASA|aB|SA|AS \\ A \longrightarrow b \\ A \longrightarrow ASA|aB|SA|AS \\ B \longrightarrow b \end{cases} \Longrightarrow \begin{cases} U \longrightarrow a \\ S_0 \longrightarrow AA_1|UB|SA|AS \\ A_1 \longrightarrow SA \\ S \longrightarrow AA_1|UB|SA|AS \\ A \longrightarrow b \\ A \longrightarrow AA_1|UB|SA|AS \\ B \longrightarrow b \end{cases} \quad \text{variabili aggiunte : } A_1,\ U$$

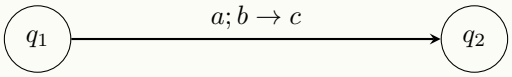
1.7 Push Down Automata

Si considera adesso un nuovo modello di computazione che estende il concetto di automa, vedremo in seguito che, i linguaggi regolari stanno agli NFA, come le grammatiche acontestuali stanno ai PDA. Introdurremo alcune caratteristiche dei PDA in modo informale, per poi darne la definizione.

Innanzitutto un PDA è sempre un automa, non deterministico, in particolare si differenzia dagli NFA/DFA per la presenza di una **pila** (di dimensione potenzialmente infinita) detta anche *stack*, il cui contenuto evolve dinamicamente durante la computazione, tale pila permette all'automa di *memorizzare*. L'accesso ad essa è limitato, esclusivamente LIFO, è possibile scrivere sulla pila una serie di caratteri appartenenti ad un determinato insieme, su di essa è possibile eseguire operazioni di

- pop
- top
- push

Se un NFA ha un solo alfabeto Σ, un PDA avrà due alfabeti, Σ e Γ, denotati *input* e *pila*. Anche la funzione δ sarà differente, infatti gli archi saranno marcati nel seguente modo



Dove b e c sono elementi di $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$, ed a è un elemento di Σ_ϵ. In particolare, la freccia in figura indica che :

> Se l'automa è nello stato q_1, legge in input a, ed in cima alla pila si trova b, allora si sposterà nello stato q_2, verrà rimosso b dalla pila e verrà aggiunto c

Il dominio della funzione di transizione sarà quindi $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$, mentre il codominio sarà l'insieme delle parti di $Q \times \Gamma_\epsilon$. Essendo non deterministico, ad ogni ramo di computazione sarà associata una differente evoluzione della pila.

Definizione (PDA) : Un *Push Down Automata* è una tupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ dove

- Q, Σ, q_0, F sono definiti identicamente agli NFA/DFA
- Γ è un alfabeto finito rappresentante gli elementi che possono essere nello stack
- $\delta : Q \times \Gamma_\epsilon \times \Sigma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$

Considerato un PDA, supponiamo che

$$(q, c) \in \delta(p, a, b) \quad \text{con} \quad \begin{cases} q, p \in Q \\ a \in \Sigma_\epsilon \\ c, b \in \Gamma_\epsilon \end{cases}$$

Allora, a seconda dei valori di a, b e c, lo step di computazione assume il seguente significato

- $a, b, c \neq \epsilon \implies$ il PDA legge a, passa dallo stato p allo stato q e sostituisce b (che si trova nel top dello stack) con c.
- $a, c \neq \epsilon \wedge b = \epsilon \implies$ il PDA legge a, passa dallo stato p allo stato q indipendentemente dal valore del top dello stack, su cui andrà ad inserire c
- $a, b \neq \epsilon \wedge c = \epsilon \implies$ il PDA legge a, passa dallo stato p allo stato q ed eseguo una pop sullo stack rimuovendo b.

Passiamo alla descrizione di come viene *eseguita la computazione* su un PDA, si consideri una stringa w in input, composta dai caratteri

$$w = w_1 \dots w_n \text{ con } w_i \in \Sigma$$

consideriamo ora gli stati che verranno attraversati nella computazione :

$$r_0, r_1, r_2 \ldots, r_n$$

e si considerino le stringhe

$$s_0, s_2 \ldots, s_n \text{ con } s_i \in \Gamma^*$$

Lo stato $r_0 = q_0$ sarà lo stato iniziale, ed $s_0 = \epsilon$ rappresenterà la pila vuota, inoltre, $\forall i = 0, 1 \ldots, n$ si avrà che

$$(r_{i+1}, a) \in \delta(r_i, w_{i+1}, b)$$

Dove s_i rappresenterà lo stack nel momento in cui si è nello stato r_i, ed s_{i+1} rappresenterà lo stack nel momento in cui si è nello stato r_{i+1}, si avrà infatti che

- $s_i = bT$
- $s_{i+1} = aT$
- $T \in \Gamma_\epsilon^*$
- In breve, nello stack è stata sostituito il simbolo b con a

La computazione sarà andata a buon fine se e solo se $r_n \in F$, ossia lo stato finale è di accettazione.

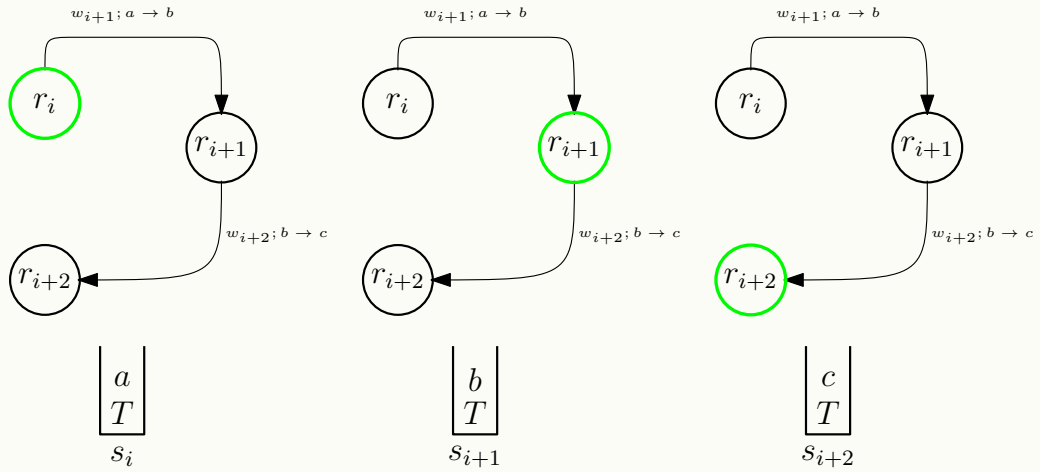


Figura 1.17: Lo stato verde rappresenta la computazione corrente

Gli stati consistenti nella computazione sono in relazione

$$(p, ax, by) \vdash (q, x, cy) \iff (q, c) \in \delta(p, a, b)$$

$$p, q \in Q \quad b, c \in \Gamma_\epsilon \quad a \in \Sigma_\epsilon \quad x \in \Sigma^* \quad y \in \Gamma^*$$

La chiusura transitiva $\vdash^*$ definisce la relazione di transizione estesa, utile per la seguente definizione.

Definizione : Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ un PDA, definiamo **linguaggio di** P, e denotiamo $L(P)$, l'insieme

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, \epsilon) \vdash^* (q, \epsilon, y) \wedge q \in F \wedge y \in \Gamma^*\}$$

Si può definire in manire equivalente assumendo che la computazione termini sempre con lo stack vuoto

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, \epsilon) \vdash^* (q, \epsilon, \epsilon) \wedge q \in F\}$$

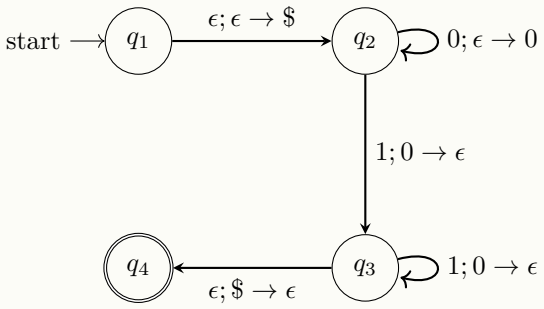
Il fatto che le due definizioni siano equivalenti, implica che ogni PDA può essere trasformato in un PDA equivalente in cui ogni stringa accettata termini la computazione con lo stack vuoto.

1.7.1 Esempi

Esempio 1

Sia $L = \{0^n 1^n | n \geq 0\}$ un linguaggio, si vuole determinare un PDA P tale che $L(P) = L$.

L'idea è quella di utilizzare lo stack, aggiungendo un valore ogni volta che si legge uno zero, e togliendone uno ogni volta che si legge un 1, se al termine la pila sarà vuota, allora la stringa letta apparterrà al linguaggio L. Definiamo $\Gamma = \{0, \$\}$, il simbolo \$ è utile per marcare il fondo dello stack, verrà aggiunto automaticamente all'inizio.

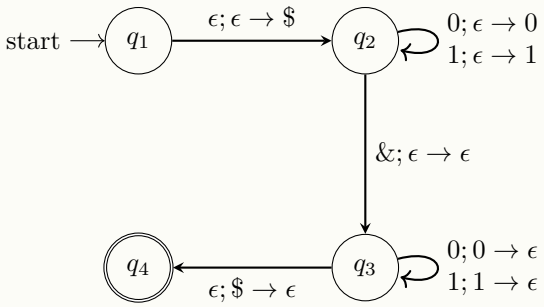


Esempio 2

Data una stringa w, definiamo w^R la stringa w specchiata, ad esempio

$$w = a12hgf \implies w^R = fgh21a$$

Sia $L = \{w\&w^R | w \in \{0,1\}^*\}$, ossia l'insieme di tutte le stringhe binarie palindrome, al cui centro è presente il carattere &.



1.7.2 PDA e Linguaggi Acontestuali

Tale sezione si concentrerà sul seguente teorema.

Teorema : Un linguaggio è acontestuale (generato da una CFG) se e solo se esiste un PDA che lo riconosce. Per rendere più leggibile e chiara la dimostrazione del teorema, essa verrà scomposta in due lemma, che rappresentano le due implicazioni della dimostrazione.

Lemma [$\implies$] : Se un linguaggio è acontestuale, esiste un PDA che lo riconosce.

Dimostrazione [$\implies$] : Sia $G = (V, \Sigma, R, S)$ una generica CFG, il cui linguaggio è $L(G)$. L'idea è quella di costruire un PDA P che sfrutti il non determinismo per accettare una generica stringa $w \in L(G)$ usando tutte le possibili derivazioni (applicazioni delle regole) di G.

Lo stack di P conterrà le variabili ed i terminali di G che verranno appositamente sostituiti con le regole. Possono esserci differenti alberi di computazione, il comportamento del PDA può essere scritto dal seguente algoritmo

◇ si inserisce $ nella pila

◇ while(true){

◇ Se nel top dello stack vi è una variabile, si eseguono varie diramazioni in base a tutte le regole che comprendono tale variabile

◇ Se nel top dello stack c'è un terminale, viene rimosso dallo stack e si confronta con il prossimo carattere in input, se identici la computazione continua, altrimenti rifiuta.

◇ Se nel top dello stack c'è $ (la pila è svuotata), si accetta se e solo se è stata letta tutta la stringa in input.

◇ }

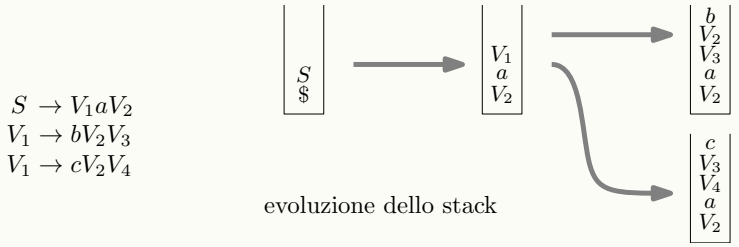
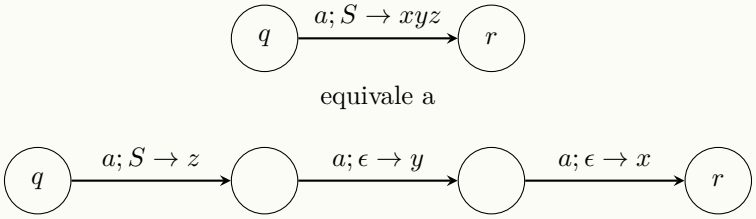


Figura 1.18: Esempio di computazione

Notazione : Per comodotià, verrà utilizzata una notazione ridotta nel diagramma del PDA in questione, in particolare, si introduce l'inserimento di una stringa nello stack, piuttosto che di un solo carattere, ovviamente un push down di questo tipo equivale a più push down, che richiedono stati intermedi che verranno omessi.



Sia $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$ l'automa che deve accettare il linguaggio $L(G)$, l'insieme degli stati sarà

$$Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$$

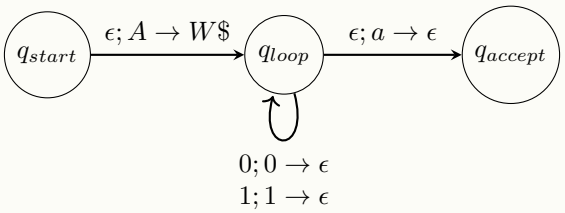
Dove E è l'insieme degli stati intermedi, ossia quelli che verranno omessi nella notazione. Siano

- $a \in \Sigma$ un carattere generico della grammatica e dell'alfabeto del PDA corrispondente
- $A \in V$ una generica variabile della grammatica

Si ricordi che gli elementi del codominio di δ sono insiemi, dato che il PDA è non deterministico. La funzione di transizione del PDA P sarà definita come segue

- $\delta(q_{start}, \epsilon, \epsilon) = \{(q_{loop}, S\$)\}$ si ricordi che S è la variabile iniziale della grammatica e $ è il simbolo per marcare il fondo dello stack
- $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, W) \mid A \to W \in R\}$, ossia $A \longrightarrow W \in R$ è una generica regola della grammatica
- $\delta(q_{loop}, a, a) = \{(q_{loop}, \epsilon)\}$
- $\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$

In generale (ignorando gli stati in E, in notazione abbreviata), il PDA assumerà una forma canonica del tipo



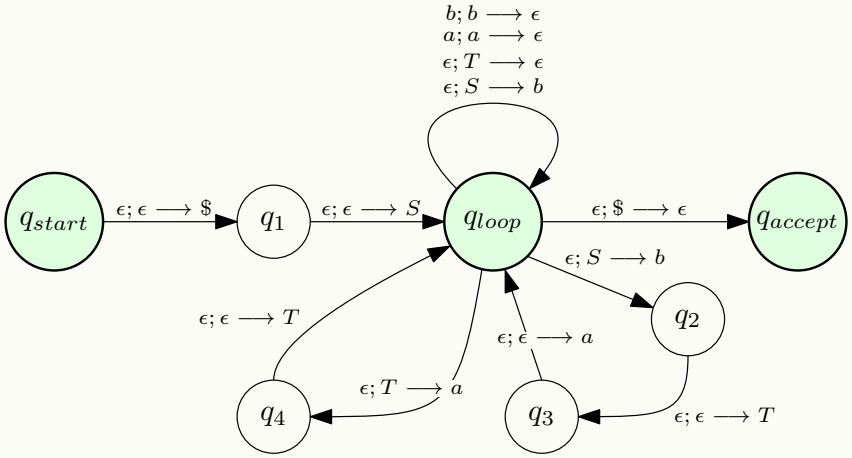
■

Esempio

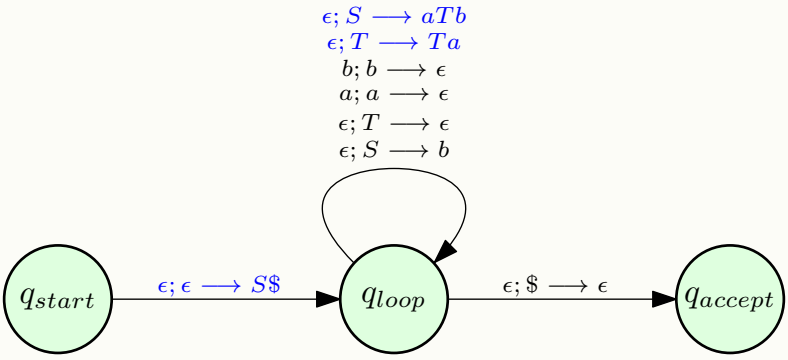
Si consideri la seguente grammatica

$$\begin{cases} S \longrightarrow aTb|b \\ T \longrightarrow Ta|\epsilon \end{cases} \qquad \begin{matrix} \Sigma = \{a, b\} \\ V = \{S, T\} \end{matrix}$$

Il PDA corrispondente avrà pila $\Gamma = \{\$, S, a, T, b\}$, precisamente



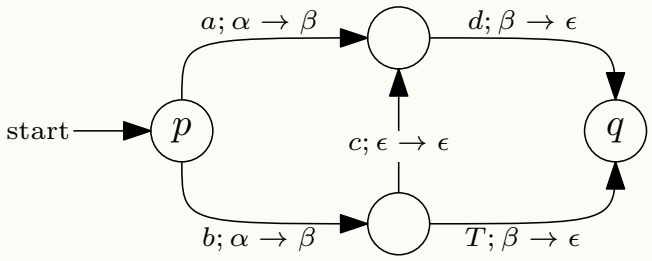
Gli stati $E = \{q_1, 1_2, q_3, q_4\}$ sono quelli di transizione che nella notazione abbreviata verrebbero omessi, sostituendo le etichette con gli archi.



notazione abbreviata

Lemma [$\Longleftarrow$] : Dato un PDA, esiste una CGF le cui stringhe generate sono il linguaggio del PDA.

Dimostrazione [$\Longleftarrow$] : Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ il PDA in questione. L'idea è quella di considerare, per ogni coppia di stati p, q in Q, una variabile denotata A_{pq} che genera tutte le stringhe che permettono al PDA di passare da p a q lasciando la pila vuota, come nell'esempio in figura 1.19.



si avrà

$A_{pq} \longrightarrow ad$
$A_{pq} \longrightarrow bcd$
$A_{pq} \longrightarrow bT$

Figura 1.19: esempio

Forma Canonica : Durante la dimostrazione, si assumerà che P soddisfi certe proprietà, ossia che assuma una *forma canonica*, tale forma non fa perdere di generalità alla dimostrazione in quanto ogni PDA può essere trasformato in un PDA in tale forma lasciando invariato il linguaggio che accetta. Un PDA è nella forma canonica se

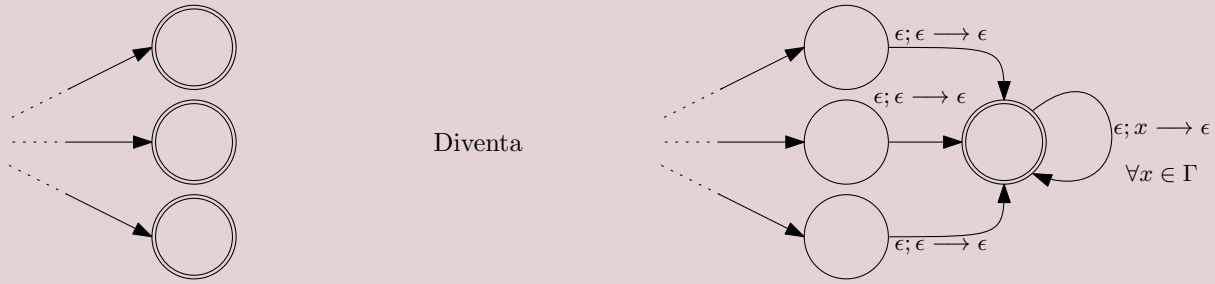
1. Inizia con la pila vuota, ed termina la computazione sempre con la pila vuota
2. Ha un solo stato accettante $|F| = 1$
3. Il PDA ad ogni passo computazionale, non sostituisce mai un elemento della pila con un altro, o ne elimina uno, o ne aggiunge uno, la sostituzione non avviene mai in un solo passo di computazione, formalmente, cambia la definizione della funzione di transizione

$$\delta_1 : Q \times \Gamma_\epsilon \times \Sigma_\epsilon \to \mathcal{P}(Q \times \{\epsilon\})$$

$$\delta_2 : Q \times \{\epsilon\} \times \Sigma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$$

$$\delta = \delta_1 \cup \delta_2$$

È chiaro che ogni PDA può essere portato in tale forma, se ci sono più stati di accettazione, si creerà un nuovo stato alla quale tutti rimandano, rendendolo l'unico stato di accettazione. Quest'ultimo stato inoltre si occuperà di svuotare la pila automaticamente grazie agli ϵ-archi. Inoltre, ogni arco che esegue una sostituzione nello stack sarà diviso in due archi che eseguono in sequenza un pop ed un push, facendo uso di uno stato intermedio.



Bisogna definire le regole dela grammatica, la variabile iniziale S sarà $A_{q_0 q_{accept}}$, quindi per definizione S genererà tutte le stringhe che accetta il PDA. L'insieme delle regole R della grammatica G sarà definito come segue

- Siano $p, q, r, s \in Q$, $u \in \Gamma$, $a, b \in \Sigma_\epsilon$, se

$$(r, u) \in \delta(p, a, \epsilon) \wedge (q, \epsilon) \in \delta(s, b, u)$$

allora ci sarà la regola

$$A_{pq} \longrightarrow aA_{rs}b$$

Figura 1.20: situazione del PDA

È ovvio che che si possa arrivare da p a q in tal modo, il carattere a porta da p ad r (aggiungendo u), il carattere b porta da s a q (rimuovendo u), la variabile A_{rs} deriva[1] i caratteri che portano da r ad s con pila vuota.

- Per ogni tripla $p, q, r \in Q$ si pone la regola $A_{pq} \longrightarrow A_{pr}A_{rq}$.
- Per ogni $p \in Q$, si pone la regola $A_{pp} \longrightarrow \epsilon$

La prova della dimostrazione segue dal fatto che A_{pq} genera x *se e solo se* x porta l'automa P da p a q con pila vuota, ne segue che, se $S = A_{q_0 q_{accept}}$ deriva x, allora $x \in L(P)$. Tale dimostrazione sarà suddivisa in due claim.

Claim 1 : Se A_{pq} deriva x, allora x porta da p a q con pila vuota.

Dimostrazione claim 1 : Verrà dimostrato per induzione sul numero di produzioni di x in G.

- **caso base** : x è generata da una regola di G per $k = 1$ produzione, allora l'unica regola che può generare x è del tipo $A_{pp} \longrightarrow \epsilon$, allora $x = \epsilon$, e la stringa vuota porta l'automa P da p a p con pila vuota.
- **ipotesi induttiva** : Per ogni x generata da una regola di G per $k > 1$ produzioni, tale stringa congiunge i due stati in questione lasciando la pila vuota.
- **passo induttivo** : Supponiamo che A_{pq} produca x in $k+1$ produzioni, allora, data la definizione delle regole di G, la regola in questione può essere di due tipi :
  - La regola in questione è $A_{pq} \longrightarrow aA_{rs}b$, allora x ha la forma ayb, ne consegue che A_{rs} deriva y in al più k passi, quindi, per ipotesi, la stringa y porta l'automa P dallo stato r allo stato s lasciando la pila vuota. Inoltre, per definizione delle regole è vero che

    $$(r, u) \in \delta(p, a, \epsilon) \wedge (q, \epsilon) \in \delta(s, b, u) \quad \text{per qualche } u \in \Gamma$$

    Quindi è chiaro che x porti P da p a q lasciando la pila vuota.
  - La regola in questione è $A_{pq} \longrightarrow A_{pr}A_{rq}$„ allora la stringa x sarà del tipo yz, quindi A_{pr} deriva y, mentre A_{rq} deriva z. Se A_{pq} deriva x in $k+1$ produzioni, allora le derivazioni di y e z avvengono in al più k produzioni, ma allora per ipotesi

    $$\begin{cases} y \text{ porta } P \text{ da } p \text{ a } r \text{ lasciando la pila vuota} \\ z \text{ porta } P \text{ da } r \text{ a } q \text{ lasciando la pila vuota} \end{cases} \implies x = yz \text{ porta } P \text{ da } p \text{ a } q \text{ lasciando la pila vuota}$$

□

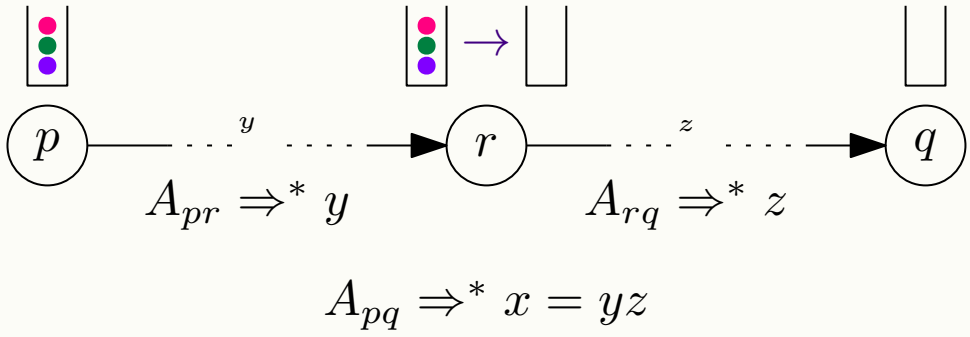
Claim 2 : Se x porta da p a q con pila vuota, allora A_{pq} deriva x.

Dimostrazione claim 2 : Verrà dimostrato per induzione sul numero di passi di computazione dell'automa P.

- **caso base** : L'automa compie 0 passi, la computazione inizia e finisce in p, l'input x è quindi la stringa vuota, siccome per definizione delle regole, G ha la regola $A_{pp} \longrightarrow \epsilon$, allora A_{pp} deriva x.

[1] con *deriva*, si intende l'atto di generare tale stringa attraverso diverse produzioni, ossia la chiusura transitiva della relazione di produzione, precedentemente denotata $\Rightarrow^*$

- **ipotesi induttiva** : Si ipotizza che per ogni computazione di $k > 0$ passi (che lasci vuota la pila) da p a q con stringa in input x, la regola A_{pq} deriva x.
- **passo induttivo** : Supponiamo che x porti da p a q lasciando la pila vuota in $k+1$ passi di computazione, precisamente, ci sono due possibili casi
  - La pila è vuota solo negli stati p e q, negli stati intermedi è stata riempita dal primo passo, e verrà poi svuotata all'ultimo passo. Sia u, tale carattere inserito all'inizio e rimosso al termine, e sia a il simbolo che porta p allo stato successivo r (primo passo di computazione), e b il simbolo che porta dal penultimo stato s a q (ultimo passo di computazione), proprio come mostrato in figura 1.20. Per definizione, la grammatica G contiene la regola $A_{pq} \longrightarrow aA_{rs}b$, quindi x è del tipo ayb. La variabile A_{rs} deriva y, quindi y porta da r a s lasciando la pila vuota con al più $k-1$ passi. Ricapitolando
    - a porta da p ad r aggiungendo u
    - y porta da r ad s senza fare operazioni sulla pila
    - b porta da s a q rimuovendo u
    - $x = ayb$ porta da p a q lasciando la pila vuota
    - A_{pq} deriva x.
  - La pila viene svuotata negli stati intermedi fra p e q, sia r lo stato in cui la pila si svuota, essendo che la computazione da p a q richiede $k+1$ passi, allora le computazioni da p ad r e da r a q richiederanno al più k passi. Sia $x = yz$, y porta da p ad r, e z porta da r a q, per ipotesi A_{pr} deriva y e A_{rq} deriva z, ma allora, essendo che esiste la regola $A_{pq} \longrightarrow A_{pr}A_{rq}$ ne consegue che A_{pq} deriva $x = yz$.



□

I due claim dimostrano il lemma $\Longleftarrow$. ■
Essendo entrambi i lati dimostrati, il teorema sull'equivalenza fra PDA e CFG è dimostrato.

1.7.3 Il Pumping Lemma per le Grammatiche Acontestuali

Il teorema presentato in questa sezione mostra che esistono anche dei linguaggi non acontestuali, ossia, per i quali non esiste alcuna grammatica che li genera. Un esempio di linguaggio acontestuale è il seguente

$$L = \{0^n 1^n 2^n \mid n \geq 1\}$$

Prima di presentare il teorema, è necessario introdurre il seguente fatto.

Claim : Sia G una CFG in forma normale Chomsky, data la forma delle regole, ogni albero di derivazione di G è binario, da ciò ne deriva che, preso un qualsiasi albero, se il cammino più lungo di tale albero è lungo i, allora allora la stringa generata sarà lunga al più 2^{i-1}.

Dimostrazione claim : Si dimostra per induzione su i

- **caso base** : $i = 1$, allora la derivazione è composta da una sola produzione, la stringa è lunga 1, infatti $2^{i-1} = 2^{1-1} = 2^0 = 1$.
- **ipotesi induttiva** : si assume sia vero per un generico $i > 1$.

- **passo induttivo** : si consideri un cammino lungo $k = i + 1 = (i > 1) + 1 \geq 3$, la prima regola applicata deve essere necessariamente del tipo

$$S \longrightarrow BC$$

i sotto alberi generati da B e C hanno il cammino più lungo (denominato da ora in poi *altezza*) grande al più i, generano quindi stringhe lunghe al più 2^{-1}, quindi S genera stringhe lunghe al più $2 \cdot 2^{i-1} = 2^i = 2^{k-1}$. ■

Teorema (Pumping Lemma per le CFG) : Sia L un linguaggio acontestuale, allora esiste un numero p tale che, presa una qualsiasi stringa w lunga almeno p, ($|w| \geq p$), esiste una sua suddivisione

$$w = uvxyz$$

tale che

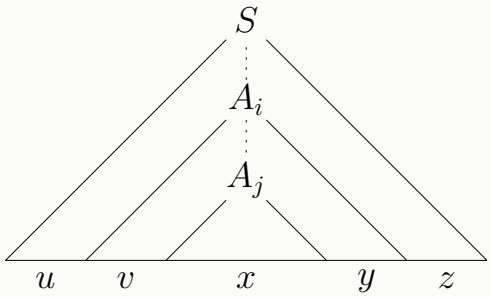
1. $\forall i \geq 0 \quad uv^ixy^iz \in L$
2. $|vy| > 0$
3. $|vxy| \geq p$

Dimostrazione (Pumping Lemma per le CFG) : Sia $G = (V, \Sigma, R, S)$ la CFG associata ad L, ossia $L(G) = L$, si assume che G sia in forma normale Chomsky (senza perdita di generalità), quindi ogni suo albero di derivazione sarà binario. Sia $m = |V|$ il numero di variabili distinte di G, identifichiamo come *pumping* il numero $p = 2^m$. Sia w una generica stringa tale che $|w| \geq p = 2^m$, allora w avrà un albero di derivazione di altezza almeno $m + 1$, ed il numero di nodi nell'albero sarà almeno $m + 2$, di cui 1 nodo è necessariamente un terminale, e gli altri $m + 1$ sono variabili.

Sia $w = uvxyz$, siccome le variabili distinte sono m, esiste una variabile A_i che si ripete nell'albero, in particolare

- L'albero con radice A_i genera vxy
- L'albero con radice $A_j = A_i$ genera x

Mentre u e z sono generate nella derivazione da S ad A_i.



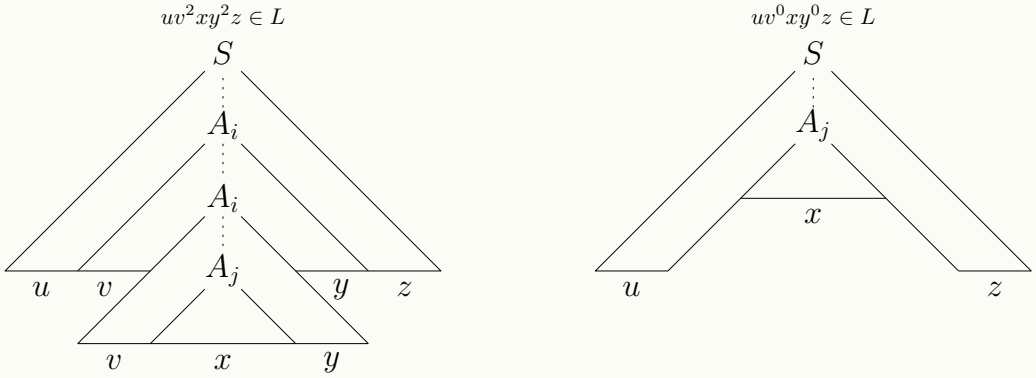
Siccome G è in forma normale, un sotto albero contiene, o un singolo terminale, o due variabili. Quindi, essendo che il sotto albero di A_i non può essere un terminale, avendo A_j, sarà composto da due variabili.

$$A_i \longrightarrow BC \in R$$

Una delle due variabili fra B e C genererà A_j che a sua volta deriverà x. L'altra variabile deriverà vy, quindi $vxy \neq x$, ciò implica che necessariamente $|vy| > 0$ (punto (ii) dimostrato).

Essendo che il cammino più lungo nell'albero ha lunghezza $m+1$, le stringhe generate avranno lunghezza minore o uguale a $2^{(m+1)-1} = 2^m = p$, quindi $|vxy| \geq p$ (punto (iii) dimostrato).

Le variabili A_i e A_j possono essere sostituite nell'albero di computazione.



In generale, è chiaro come $\forall i\ uv^ixy^iz \in L$. (punto (i) dimostrato), la dimostrazione del teorema è completa. ■

1.7.4 Esercizi ed Ultime Proprietà sulle CFG

In questa sezione verrà utilizzato il pumping lemma per dimostrare che alcuni linguaggi non sono acontestuali.

Esercizio 1

Si consideri il linguaggio

$$L = \{0^n1^n2^n \mid n \geq 1\}$$

Tale linguaggio non è acontestuale, non esiste nessuna CFG che lo genera. Per assurdo, si assuma che L sia acontestuale, allora $\exists p$ tale che, per ogni stringa $w \in L$ di lunghezza al più p, valgono le condizioni del pumping lemma.

Si prende in esame la stringa $w = 0^p1^p2^p$, bisogna considerare tutte le scomposizioni di w del tipo

- $w = uvxyz$ con
- $|vy| > 0$
- $|vxy| \leq p$

$$w = \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}\ \underbrace{22\ldots2}_{p \text{ volte}}$$

Se la stringa vxy è lunga al più p conterrà 1 oppure 2 caratteri distinti, dato che, se contenesse 3 caratteri distinti (sia 0 che 1 che 2) allora per costruzione di w sarebbe lunga più di p. Si consideri il punto (1) del pumping lemma, preso $i = 0$ la stringa

$$\hat{w} = uv^0xy^0z$$

Deve essere contenuta in L. Siccome la stringa vxy per definizione non era vuota, la stringa $\hat{w} = uxz$ avrà un numero di elementi minore di w dato che v ed y sono state rimosse ed insieme erano composte da almeno 1 carattere.

La rimozione di vy comporta il cambio del numero di occorrenze di 1 (oppure 2) simboli in $\hat{w}$ rispetto a w, quindi $\hat{w}$ avrà 1 (oppure 2) simboli le cui occorrenze differiscono dal/dai restante/restanti, ma allora non è del tipo $0^n1^n2^n,\ \ n \geq 1$, quindi non è in L, ma allora non è vero che L è acontestuale.

Esercizio 2

Si consideri il linguaggio

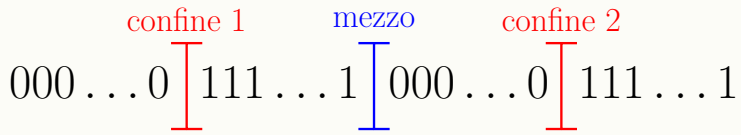
$$L = \{ww \mid w \in \{0,1\}^*\}$$

Tale linguaggio non è acontestuale, non esiste nessuna CFG che lo genera. Per assurdo, si assuma che L sia acontestuale, allora $\exists p$ tale che, per ogni stringa $w \in L$ di lunghezza al più p, valgono le condizioni del pumping lemma.

Si considera la stringa

$$w = 0^p 1^p 0^p 1^p$$

di cardinalità $|w| = 4p$, bisogna considerare ogni possibile modo di scomporre la stringa, come prima cosa, si individuano le seguenti sezioni nella stringa



La stringa può essere scomposta in diversi modi

- *caso 1* : vxy non si trova a cavallo fra i confini e non si trova a cavallo nel mezzo, è quindi confinata fra una delle 4 metà della stringa, è quindi composta di soli 0 oppure di soli 1. Secondo il punto (1) del pumping lemma, preso $i = 2$ si ha la stringa

  $$\hat{w} = uv^2xy^2z$$

  ma allora $\hat{w}$ non appartiene ad L.

- *caso 2* : vxy si trova a cavallo fra uno dei due confini, ma non tocca il mezzo. Si considera $i = 0$:

  $$\hat{w} = uv02xy^0z = uxz$$

  eliminando vy, si crea un "buco" rispetto alla stringa originale, tale buco ha dimensione al più p, essendo che una delle due metà della stringa cambia, il mezzo si sposta, supponiamo che vxy si trovava sul confine destro (la dimostrazione è analoga al sinistro), allora il centro si sposterà verso destra di $\geq p$ posizioni, e la metà destra della stringa terminerà necessariamente con un 1, ma la metà sinistra terminerà necessariamente con uno zero, quindi le due metà non sono uguali e $\hat{w} \notin L$.

- *caso 3* : vxy si trova a cavallo sul mezzo ma non oltrepassa alcun confine. Si considera, preso $i = 0$

  $$\hat{w} = uxz$$

  In tal modo, la seconda e la terza sezione di w verrà modificata, ed una delle due parti avrà meno di p caratteri

  $$w = \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}\ \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}$$

  $$\hat{w} = \underbrace{00\ldots0}_{p \text{ volte}}\ \underbrace{11\ldots1}_{k \text{ volte}}\ \underbrace{00\ldots0}_{j \text{ volte}}\ \underbrace{11\ldots1}_{p \text{ volte}}$$

  $$k \neq p \vee j \neq p$$

  Ma allora $\hat{w} \notin L$.

Durante dimostrazioni di questo tipo può risultare utile l'applicazione di una proprietà delle CFG, ossia la loro **chiusura per concatenazione**, siano $G_1 = (V_1, \Sigma, R_1, S_1)$ e $G_2 = (V_2, \Sigma, R_2, S_2)$ due CFG, la grammatica $G = (V_1 \cup V_2, \Sigma, R, S)$ dove

- S è un nuovo stato
- $R = R_1 \cup R_2 \cup \{S \to S_1S_2\}$

é ancora una CFG.

Differentemente, le CFG non sono chiuse per intersezione, basta dare un singolo contro esempio per dimostrare la tesi, si considerino i seguenti linguaggi

$$L_1 = \{0^n1^n2^i \mid n \geq 0 \wedge i \geq 0\}$$
$$L_2 = \{0^k1^n2^n \mid n \geq 0 \wedge k \geq 0\}$$

Entrambi i linguaggi sono acontestuali, infatti sono generati dalle regole

$$R_1 = \begin{cases} S \longrightarrow S_1S_2 \\ S_1 \longrightarrow 0S_11|\epsilon \\ S_2 \longrightarrow 2S_2|\epsilon \end{cases} \qquad R_2 = \begin{cases} S \longrightarrow S_1S_2 \\ S_1 \longrightarrow 0S_1|\epsilon \\ S_2 \longrightarrow 1S_22|\epsilon \end{cases}$$

L'intersezione dei due linguaggi $L = L_1 \cup L_2 = \{0^n1^n2^n \mid n \geq 0\}$ sappiamo essere un linguaggio acontestuale, quindi le CFG **non sono chiuse per intersezione**.

Esercizio 4

Si consideri il seguente linguaggio

$$L = \{a, b\}^* \setminus \{ww \mid w \in \{a, b\}^*\}$$

Comprende tutte le stringhe che non possono essere scritte come una stessa stringa ripetuta due volte. Nell'esercizio 1.7.4 si è visto che il complementare di L, non è un linguaggio acontestuale, si vuole dimostrare che invece L lo è. Sicuramente, tutte le stringhe di cardinalità dispari saranno in L. Le regole che generano il linguaggio sono le seguenti

$$R = \begin{cases} S \longrightarrow A|B|AB|BA \\ A \longrightarrow a|aAa|aAb|bAa|bAbB \longrightarrow b|aBa|aBb|bBa|bBb \end{cases}$$

- A genera le stringhe dispari con una a al centro
- B genera le stringhe dispari con una b al centro

Dimostrazione : : Si vuole dimostrare che le stringhe di lunghezza pari generate da R sono in L. Si dimostreranno entrambi i versi.

$\boxed{\implies}$: Se $x \in L$, allora $S \Rightarrow^* x$. Se $x \in L$, allora esiste almeno una lettera che si differenzia fra le due sotto stringhe di eguale lunghezza che compongono x, sia $n = |x|$

$$\exists\ i \mid x_i \neq x_{n/2+1}$$

x può essere scritta come unione di due stringhe di lunghezza dispari $x = uv$. Si pongono

- $u = x_1x_2 \dots x_{2i-1}$
- $v = x_{2i}x_{2i+1} \dots x_n$

$S \longrightarrow AB$ e $A \longrightarrow u \wedge B \longrightarrow v \implies S \Rightarrow^* x$.

$\boxed{\impliedby}$: Se $S \Rightarrow^* x$ allora $x \in L$. Siccome $|x| = n = 0 \mod 2$, può essere generata a partire da $S \longrightarrow AB$ oppure $S \longrightarrow BA$, si suppone che sia generata dalla prima (il procedimento è analogo), allora $x = uv$ con $u \neq v$ e

- $A \Rightarrow^* u$
- $B \Rightarrow^* v$

Sia $|u| = l$ e $|v| = n - l$, le lettere centrali di u e v sono

- $u_{\frac{l+1}{2}}$
- $v_{\frac{n-2+1}{2}}$
- dove $u_{\frac{l+1}{2}} \neq v_{\frac{n-2+1}{2}}$

Scrivendo le lettere in funzione di x si ha

$$u_{\frac{l+1}{2}} = x_{\frac{l+1}{2}} \qquad v_{\frac{n-2+1}{2}} = x_{\frac{n+l+1}{2}}$$

dimostrazione non completa

CAPITOLO

2

CALCOLABILITÀ

2.1 Macchina di Turing

Esistono linguaggi che nessuna CFG può accettare, si vuole estendere il modello di calcolo ad uno più potente. Negli anni 30' del ventesimo secolo fu introdotta la **Macchina di Turing** (alla quale ci riferiremo come TM), estendendo gli automi dandogli una memoria illimitata, tale modello è una semplice astrazione dei calcolatori odierni, e corrisponde alla nozione di algoritmo.

Caratteristiche di una TM

- Una TM possiede un *nastro di lavoro*, rappresenta una memoria (illimitata) sulla quale il modello può scrivere dei caratteri.
- Una *testina di lettura* che identifica la precisa posizione attuale sul nastro, che può spostarsi a destra o a sinistra.
- Vi sono poi degli *stati di accettazione* (la stringa in input è accettata dalla TM) e degli *stati di rifiuto* (la stringa in input è rifiutata dalla TM). Quando la TM passa su uno stato di rifiuto, la computazione è immediatamente terminata.
- È possibile che per alcuni input una TM entri in uno stato di *loop* in cui non termina.

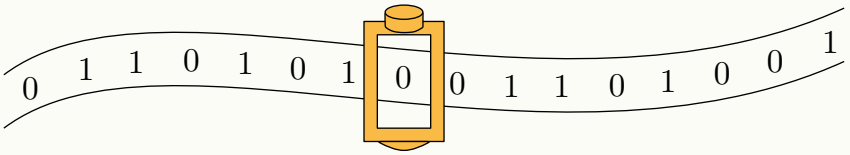
Definizione (Turing Machine) : Una TM $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ è una tupla tale che

- Q è l'insieme degli stati
- Σ è l'alfabeto delle stringhe in input
- Γ è l'insieme dei caratteri che possono essere scritti sul nastro, solitamente $\Sigma \subseteq \Gamma$. Inoltre in Γ vi è sempre un carattere speciale $\sqcup$ (denominato "blank") che rappresenta il carattere vuoto. Inoltre $\sqcup \notin \Sigma$. δ è la funzione di transizione definita

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

  L'insieme $\{L, R\}$ rappresenta i possibili spostamenti della testina a sinistra o a destra
- q_0 è lo stato iniziale
- q_{acc} è lo stato (unico) di accettazione
- q_{rej} è lo stato (unico) di rifiuto

Canonicamente, la configurazione iniziale di una TM prevede l'intera stringa in input contenuta nel nastro, seguita dal carattere ⊔.



Una TM computa la stringa in input seguendo le regole definite dalla δ, come per gli automi, per una TM è defintia la configurazione ad un certo passo nella configurazione, essa determina il contenuto del nastro, la posizione della testina, e lo stato attuale. Una configurazione si denota

$$uqav$$

dove

1. $u, v \in \Gamma^*$
2. $q \in Q$
3. uav è il contenuto del nastro, $a \in \Gamma$ è il carattere su cui si trova la testina.

Data in input una stringa w, la configurazione iniziale sarà $q_0 w$. Si può rappresentare graficamente una configurazione come segue

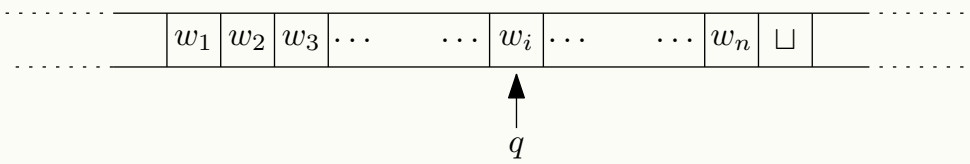


Figura 2.1: la testina è sul carattere w_i e lo stato attuale è q

Per definire il concetto di accettazione, bisogna stabilire la relazione di *produzione* :

$$uaq_ibv \text{ produce } uq_jacv$$
$$\text{se e solo se}$$
$$\delta(q_i, b) = (q_j, c, L)$$

Vuol dire che la TM, nello stato q_i, leggendo con la testina il carattere b si sposta a sinistra. Può essere analogamente definito per lo spostamento a destra.

$$uaq_ibv \text{ produce } uacq_jv$$
$$\text{se e solo se}$$
$$\delta(q_i, b) = (q_j, c, R)$$

Diremo che una TM **accetta** w se e solo se esiste una sequenza di configurazioni

$$C_1 \to C_2 \to C_3 \cdots \to C_k$$

dove

- $C_1 = q_0 w$
- $\forall i \quad C_i$ produce C_j
- lo stato della configurazione C_k è lo stato accettante $C_k = uq_{acc}av$

Definizione (Riconoscibilità) : Un linguaggio L è **turing riconoscibile** se esiste una TM M che *accetta* ogni sua stringa, si dice che L è il linguaggio di M.

Se una TM deve computare una stringa che non è nel suo linguaggio, può

- rifiutare
- andare in loop

Una TM M per cui, data una qualsiasi stringa, non vai mai in loop, viene detta **decisore**.

Definizione (Decidibilità) : Un linguaggio L è **turing decidibile** se esiste una TM M che è un decisore per L, ossia, per ogni sua stringa, non va mai in loop. Si dice che M *decide* L.

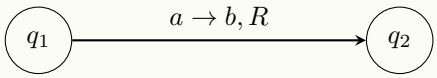
Data una TM M si hanno gli insiemi

- $L(M) = \{w \in \Sigma^* \mid M \text{ accetta } w\}$
- $R(M) = \{w \in \Sigma^* \mid M \text{ rifiuta } w\}$
- Generalmente $L(M) \cup R(M) \subseteq \Sigma^*$
- Se $L(M) \cup R(M) = \Sigma^*$ allora M è un decisore.

Un linguaggio che non ha decisori *non è decidibile*, la definizione di decidibilità stabilisce i limiti della computabilità, esistono infatti dei linguaggi (astrando, dei problemi) che non possono essere decisi (non possono essere risolti), ciò si lega inevitabilmente con i *teoremi di incompletezza* di Gödel, esisteranno sempre delle proposioni per cui è *impossibile* stabilire se sono vere o false.

2.1.1 Esempi di TM

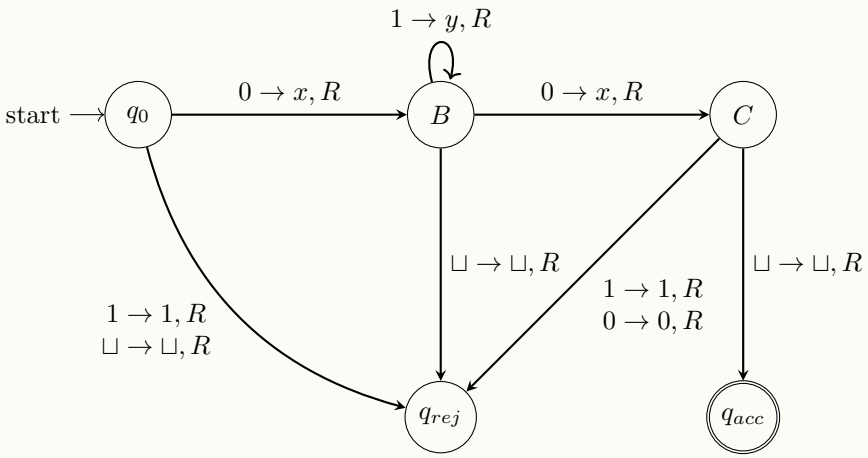
Le TM verranno rappresentato in maniera compatta sottoforma di grafi proprio come per gli automi.



Il grafo rappresentato in figura descrive la seguente situazione : Se la TM si trova nello stato q_1, e la testina si trova sul carattere a, allora si sostituisce il carattere a nel nastro con il carattere b, si sposta la testina a destra (se al posto di R ci fosse stato L si sarebbe andati a sinistra) e la TM si sposta nello stato q_2.

Esempio 1

Si consideri il seguente linguaggio $L = \{01^*0\}$. La TM che decide L è la seguente

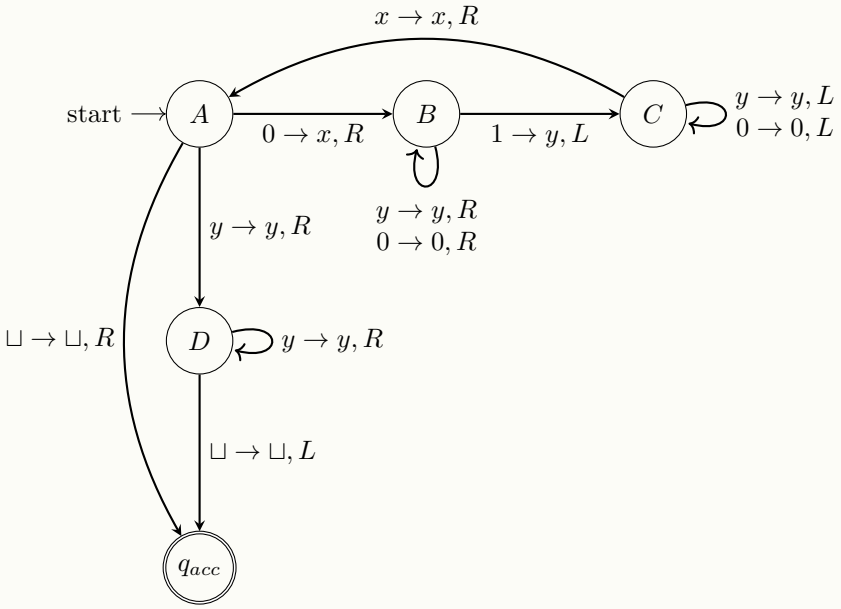


Esempio di computazione su una stringa di L:

$$\begin{array}{lll} \bar{0}1110\sqcup & x\bar{1}110\sqcup & xy\bar{1}10\sqcup \\ xyy\bar{1}0\sqcup & xyyy\bar{0}\sqcup & xyyyx\bar{\sqcup} \end{array}$$

Esempio 2

Si consideri il seguente linguaggio $L = \{0^n 1^n \mid n \geq 0\}$. La TM che decide L è la seguente (stato di rifiuto omesso)



Nel capitolo precedente si è visto come il linguaggio $L = \{0^n 1^n 2^n \mid n \geq 0\}$ non è acontestuale, è possibile computarlo tramite le TM, de facto, basterà unire 2 TM che si comportino come quella vista nell'esempio 2.1.1, infatti nella prima computazione si occuperà di controllare che la stringa in input abbia i primi $2n$ caratteri del tipo $0^n 1^n$.

$$00\ldots0 \quad 11\ldots1 \quad 22\ldots2 \quad \text{viene trasformata} \quad xx\ldots x \quad yy\ldots y \quad 22\ldots2$$

La seconda TM si occuperà di controllare se gli ultimi $2n$ caratteri sono del tipo $y^n 2^n$.

$$xx\ldots x \quad yy\ldots y \quad 22\ldots2 \quad \text{viene trasformata} \quad xx\ldots x \quad xx\ldots x \quad yy\ldots y$$

Senza perdita di generalità, è possibile astrarre le TM considerano due nuovi modelli equivalenti

- **TM multinastro**
- **TM non deterministica**

Prima di introdurli, si consideri il seguente esempio di astrazione di una TM, ossia di una macchina che, ad ogni passo di computazione, piuttosto che spostarsi necessariamente a destra o a sinistra, può rimanere ferma, la denominiamo STM.

La definiamo M', e la sua funzione di transizione sarà del tipo

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

Dove S (che sta per 'stay') indica l'azione del restare fermi senza muovere la testina.

Proposizione : Per ogni STM esiste una TM classica equivalente.

Dimostrazione : ,Sia M' la STM, l'idea è quella di considerare una TM M che gestisca le transizioni di stato in cui la testina non si muove, a tal scopo, basta simulare tale azione tramite una sequenza di 2 movimenti che fanno uso di uno stato ausiliario q_s.

$$\delta'(q, a) = (p, b, S)$$
è equivalente alle azioni
$$\begin{cases} \delta(q, a) = (p, q_s, L) \\ \delta(q_s, *) = (p, *, R) \end{cases}$$

dove $*$ rappresenta un qualsiasi elemento di Γ.

2.1.2 TM multi nastro

Introduciamo la macchina di Turing con più nastri, che verrà denotata MTM, sia k il numero di nastri, ogni nastro avrà una testina proprietaria, uno stato sarà quindi rappresentato dalle stringhe scritte su tutti e k i nastri, e la relativa posizione delle testine.

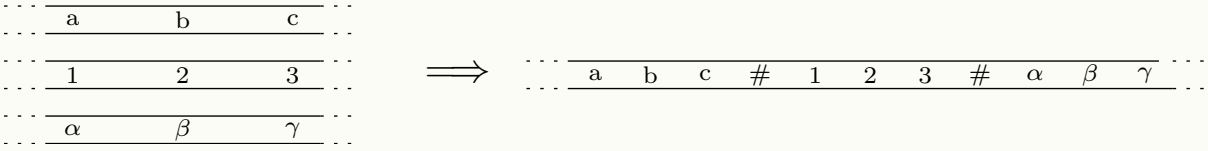
La funzione di transizione prenderà le decisioni valutando le posizioni ed il valore di tutte le testine

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

Teorema (MTM≡TM) : Per ogni MTM esiste una TM equivalente.

Dimostrazione : La dimostrazione, piuttosto che formale, mostrerà la costruzione di una TM che si comporta come la generica MTM.

Sia MM la macchina multi nastro, ed M la macchina classica che deve simularla, si introduce un nuovo carattere $\# \notin \Gamma$ che verrà utilizzato nel nastro di M come separatore per i differenti nastri originari di MM.



Per simulare le k testine, si implementa la possibilità per M di *marcare* i caratteri (in tal caso, con un punto come apice). I caratteri marcati indicano che la testina è posta su di essi. Sia Γ' l'alfabeto dei nastri di MM, e Γ l'alfabeto del nastro di M.

$$\forall a \in \Gamma', \quad \exists \dot{a} \in \Gamma$$

Si descrive ora la computazione di M, data una stringa in input w la configurazione iniziale è la seguente:

$$\#\dot{w}_1 w_2 w_3 \ldots, w_n \#\underbrace{\dot{\sqcup}\#\dot{\sqcup}\ldots\#\dot{\sqcup}}_{k-1 \text{ volte}}$$

Nel *passo di computazione*, si scansiona una prima volta il nastro partendo dal primo simbolo #, leggendo tutti i caratteri marcati, considerando poi la funzione di transizione, si esegue una seconda scansione aggiornando i valori delle testine (le marcature dei caratteri) ed il contenuto dei k caratteri in questione. Se la testina su uno dei nastri deve spostarsi su #, allora M sposta il contenuto dell'intero nastro a destra di 1 posizione.

Corollario : L è turing riconoscibile/decidibile se e solo se esiste una MTM che lo riconosce/decide.

2.1.3 TM non deterministiche

Si arricchisce il modello della macchina di turing introducendo il non determinismo, denotandolo NTM, la funzione di transizione cambia definizione

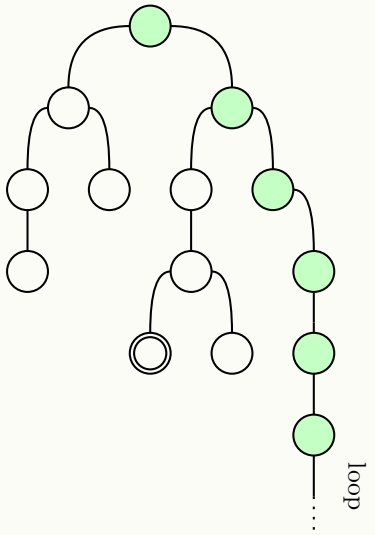
$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Per ogni input si diramano più vie di computazione, una NTM accetta una stringa in input se almeno un ramo di computazione è accettante.

Teorema (NTM≡TM) : Per ogni NTM esiste una TM equivalente.

Dimostrazione : La dimostrazione, piuttosto che formale, mostrerà la costruzione di una TM che si comporta come la generica NTM.

Sia N la NTM in questione, e sia M la TM classica che la vuole simulare. M, per essere equivalente, dovrà esplorare ogni ramo di computazione, precisamente, deve esplorarlo in altezza. L'esplorazione in profondità in presenza di un loop su un ramo di computazione, comporterà il loop su M, anche se l'originaria N aveva uno stato accettante su un differente ramo.



Idea algoritmica : M utilizzerà 3 nastri

1. il primo nastro conterrà la stringa in input che non verrà modificata (il riferimento va mantenuto)
2. il secondo nastro sarà il nastro di lavoro che verrà modificato per esplorare un cammino emulando la NTM.
3. il terzo nastro conterrà l'indirizzo del nodo dell'albero che si sta esplorando.

Nell'i-esimo step, identificato da una tripla

$$(M, w, i)$$

usando l'input del primo nastro si percorre il ramo simulando N, se viene trovato uno stato accettante, M accetta, altrimenti si incremente l'indice i e si ripercorre.

- Se un ramo di N è accettante, M accetta
- Se ogni ramo di N rifiuta, M rifiuta
- Se almeno un ramo di N va in loop, e nessun ramo accetta, M va in loop.

  Se una NTM è un decisore, tutti i cammini hanno lunghezza finita

■

2.1.4 L'Enumeratore

L'enumeratore è un modello (in particolare, una variante di una TM) capace di generare un preciso insieme di stringhe, in un qualsiasi ordine e con eventuali ripetizioni, in particolare, per ogni linguaggio turing-riconoscibile, esiste un enumeratore che lo genera.

Un enumeratore è una TM che "stampa" delle stringhe, eventualmente, infinite.

Teorema : Un linguaggio è turing riconoscibile se e solo se esiste un enumeratore che lo genera.

Dimostrazione : Si dimostrano separatamente le due direzioni.

$\boxed{\Longleftarrow}$: Dato un enumeratore E, definiamo una TM M come segue (sia w l'input)

- Si esegue E, se stampa una stringa, si confronta con w
- Se w è uguale ad una stringa stampata da E, allora M accetta.

$\boxed{\Longrightarrow}$: Sia M una TM, si considera un enumeratore E che deve stampare $L(M)$. Sia Σ l'alfabeto di $L(M)$, identifichiamo come segue

$$\Sigma^* = \{\epsilon, s_1, s_2 \dots\}$$

la lista di tutte le possibile stringhe su tale alfabeto. E sarà definito come segue

- Si ripetono i passi per $i = 1, 2 \dots$
  - Si esegue M per i passi su ogni input $s_1, s_2 \dots s_i$
  - Se una qualsiasi computazione su s_j accetta, E stampa s_j

■

2.2 Decidibilità dei Linguaggi

Come esistono linguaggi non regolari e linguaggi non acontestuali, risulta naturale porsi la domanda : Esistono dei linguaggi non turing riconosibili? Il modello della TM corrisponde al concetto di algoritmo, un linguaggio non riconoscibile corrisponde ad un problema che *nessun* algoritmo può risolvere.

Una TM può ricevere in input un qualsiasi oggetto matematico, come un polinomio, un DFA, o direttamente un'altra TM, è importante che tale oggetto O, venga **codificato** in binario, denotandolo $< O >$ prima di essere computato dalla TM.

Vediamo come prima cosa un esempio di linguaggio decidibile, si consideri

$$A_{DFA} = \{< D, w > \mid D \in DFA \wedge w \in L(d)\}$$

con $D \in DFA$ si indica che D è un DFA.

Proposizione : A_{DFA} è decidibile.

Dimostrazione : Si vuole costruire un decisore per A_{DFA}, la TM in questione avrà il seguente comportamento

1. Su input $< D, w >$ interpreta D come un DFA e w come stringa, se non riconosce l'input, rifiuta
2. Simula l'esecuzione di D con input w
3. Se D accetta w, allora la TM accetta, altrimenti rifiuta

■

In maniera simile, si può dimostrare che anche $A_{NFA} = \{< N, w > \mid D \in NFA \wedge w \in L(d)\}$ è decidibile, utilizzando una TM non deterministica, oppure, si potrebbe trasformare l'input in un DFA ed eseguire l'algoritmo appena trattato.

Si consideri il linguaggio

$$E_{DFA} = \{< D > \mid D \in DFA \wedge L(D) = \emptyset\}$$

Ossia di tutte le codifiche di DFA che hanno linguaggio vuoto, ossia che non accettano nessuna stringa.

Proposizione : E_{DFA} è decidibile.

Dimostrazione : La TM che deciderà il linguaggio codificherà il DFA come un grafo, in gli stati sono i nodi, e vi è un arco da un nodo ad un altro solo se esiste una transizione fra i due stati. In particolare la TM seguirà il seguente comportamento

1. marca lo stato iniziale, ed inizia a fare una passeggiata sul grafo
2. marca ogni stato raggiungibile
3. se almeno uno stato di accettazione è marcato, la TM rifiuta, altrimenti accetta ■

Un altro esempio interessante è il seguente

$$EQ_{DFA} = \{< D_1, D_2 > \mid D_1, D_2 \in DFA \wedge L(D_1) = L(D_2)\}$$

L'insieme delle coppie (codificate) di DFA che accettano lo stesso linguaggio.

Proposizione : EQ_{DFA} è decidibile.

Dimostrazione : Dati due DFA si definisce l'operatore di differenza simmetrica Δ come segue

$$L(D_1)\Delta L(D_2) = (L(D_1) \cap \overline{L(D_2)}) \cup (L(D_2) \cap \overline{L(D_1)})$$

Fatto : $L(D_1) = L(D_2) \iff L(D_1)\Delta L(D_2) = \emptyset$. Dati D_1 e D_2 si definisce D tale che $L(D) = L(D_1)\Delta L(D_2)$ e si esegue l'algoritmo per decidere E_{DFA} su D. ■

La seguente proposizione sarà necessaria a dimostrare la decidibilità di un linguaggio.

Proposizione : Se G è una CFG in forma normale Chomsky, e $w \in L(G)$, se $|w| = n$ per produrre w saranno necessarie $2n - 1$ derivazioni.

Dimostrazione : Il caso base per $n = 1$ è banale. Se $n \geq 2$ la regola di partenza sarà

$$S \to AB$$

A produrrà u e B produrrà v, con $w = uv$. Allora

$$|u| = k \quad |v| = n - k$$

Quindi per ipotesi induttiva per generare w saranno necessari

$$1 + 2k - 1 + 2(n - k) - 1 = 1 + 2k - 1 + 2n - 2k - 1 = 2n - 1$$

passi. ■

Si consideri

$$A_{CFG} = \{< G, w > \mid G \in CFG \wedge G \text{ produce } w\}$$

Proposizione : A_{CFG} è decidibile.

Dimostrazione : Si può assumere che G sia in forma normale Chomsky. Si costruisce una TM che considera tutte le derivazioni lunghe $2n - 1$ di G. Se trova w accetta, altrimenti rifiuta. ■

Anche il linguaggio

$$E_{CFG} = \{< G > \mid G \in CFG \wedge L(G) = \emptyset\}$$

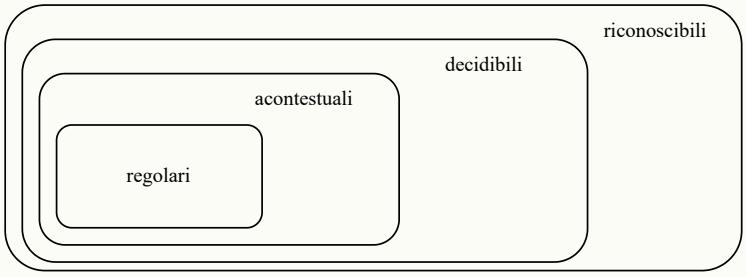
è decidibile, se ne da un *idea di dimostrazione*, la TM può procedere nel seguente modo

1. data G marca ogni terminale
2. marca ogni variabile A se esiste la regola $A \longrightarrow U_1U_2 \dots U_k$ e se esiste un U_i marcato.
3. accetta se e solo se S non è marcata, altrimenti rifiuta.

2.3 Linguaggi non Decidibili

Un linguaggio può essere regolare, acontestuale, decidibile e riconoscibile, in generale vi è una relazione di inclusione fra gli insiemi delle classi dei linguaggi.

- regolari $\subset$ acontestuali
- acontestuali $\subset$ decidibili
- decidibili $\subset$ riconoscibili



2.3.1 Macchina di Turing Universale

Vedremo che il linguaggio contenente le codifiche delle macchine di turing ed una qualsiasi stringa che accettano, è riconoscibile ma non decidibile.

$$A_{TM} = \{< M, w > \mid M \in TM \wedge w \in L(M)\}$$

Teorema : A_{TM} è turing-riconoscibile.

Dimostrazione : Si definisce una TM nota detta *macchina di Turing universale* denotata U, sarà quella che riconoscerà A_{TM}. Avrà 2 nastri, nel primo conterrà la codifica dell'input $< M, w >$, nel secondo conterrà la configurazione corrente $< (a, q, b) >$ dell'esecuzione simulata di M.

È necessario definire la codifica di M, w. Essendo

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$$

Siano

$$n = |\Sigma| \quad m = |\Gamma| \quad s = |Q|$$

la codifica sarà

$$(n)_2, (m)_2, (s)_2, < \delta >$$

dove , è un carattere speciale, $(k)_2$ è la codifica binaria di $k \in \mathbb{Z}$ e $< \delta >$ è la codifica binaria delle transizioni

$$< \delta >= R_1; R_2 \ldots; R_j$$

Le regole sono del tipo

$$R = ((q, a), (r, b, z)) \qquad z \in \{L, R\}$$

I passi di esecuzione sono i seguenti

1. Sul nastro 2 viene scritta la configurazione iniziale
2. al passo t-esimo, la configurazione sarà del tipo

   $$< (a, q, b) >$$

   (a) si estrae q dalla configurazione
   (b) si scansiona il nastro 1 in cerca di una regola del tipo

   $$< (q, x), (r, y, z) >$$

   se $x \neq b[1]$[1], si controlla la regola successiva, se $x = b[1]$ si aggiorna la configurazione sul nastro 2 in accordo con la regola letta.
   (c) se il nastro 2 contiene una configurazione con uno stato fra q_{acc}, q_{rej}, accetta o rifiuta di conseguenza.
3. Se $w \in L(M)$, U accetta, altrimenti rifiuta. Se M va in loop, U andrà in loop di conseguenza.

U è quindi un riconoscitore per A_{TM}. ■

[1] primo carattere di b

2.3.2 Diagonalizzazione

La diagonalizzazione è una tecnica utilizzata per dimostrare alcune semplici proprietà degli insiemi. Sappiamo un insieme generico A è numerabile se esiste una biezione da $\mathbb{N}$ a A. Ad esempio, l'insieme dei numeri pari E è numerabile perché esiste

$$f : E \to \mathbb{N} \quad | \quad f(n) = 2n$$

che è una biezione.

Teorema : $\mathbb{R}$ non è numerabile.

Dimostrazione : è necessario dimostrare che un sottoinsieme di $\mathbb{R}$ non sia numerabile. Si pone per assurdo che esiste una biezione da $\mathbb{N}$ a $[0, 1]$. Sia f tale biezione, ad ogni valore di $\mathbb{N}$ associa un numero reale fra zero ed uno.

$$\begin{aligned} f(1) &= 0.52423\ldots \\ f(2) &= 0.08362\ldots \\ f(3) &= 0.92841\ldots \\ &\vdots \end{aligned}$$

Si definisce un numero d tale che, la parte intera di d è 0, e l'i-esima cifra decimale di d è diversa dall'i-esima cifra decimale di $f(i)$.

$$\begin{aligned} f(1) &= 0.\,5\,2\,4\,2\,3\ldots \\ f(2) &= 0.\,0\,8\,3\,6\,2\ldots \\ f(3) &= 0.\,9\,2\,8\,4\,1\ldots \\ &\vdots \end{aligned} \qquad \begin{aligned} d &= 0.\,a\,b\,c\,d\,e\ldots \\ a &\neq 5 \\ b &\neq 8 \\ c &\neq 8 \end{aligned}$$

Essendo f biettiva per ipotesi, $\exists k \in \mathbb{N} | f(k) = d$, ma in questo modo la k-esima cifra di $f(k)$ è identica alla k-esima cifra di d, dato che $d = f(k)$. Ma allora d non è contenuto nell'immagine di f, quindi f non è biettiva. ■

Proposizione : L'insieme delle stringhe binarie (o non) di lunghezza infinita non è numerabile.

La dimostrazione sarà omessa, ma si può dimostrare con la diagonalizzazione. Si presenteranno ora due proposizioni che saranno utili a dimostrare un risultato fondamentale.

Proposizione 1 : L'insieme TM di tutte le macchine di turing è numerabile.

Dimostrazione 1 : Si consideri Σ^*, ossia l'insieme di tutte le stringhe finite (su un generico alfabeto Σ). Ogni macchina di turing M può essere codificata con una stringa finita. Quindi

$$\{< M > \ | \ M \in TM\} \subseteq \Sigma^*$$

Si definisce una relazione d'ordine $<_l$ che rappresenta l'ordinamento lessicografico per tutte le stringhe di l caratteri. Si estende poi a tutte le stringhe con la relazione $<^*$ definita

$$x <^* y \iff |x| < |y| \vee (|x| = |y| \wedge x <_l y)$$

$<^*$ è una relazione d'ordine totale. Si definisce $f : \mathbb{N} \to \Sigma^*$

$$f(i) = i\text{-esimo elemento di } \Sigma^* \text{ secondo l'ordinamento } <^*$$

f è una biezione dato che per ogni $i \in \mathbb{N}$ si può associare un elemento di Σ^*. Quindi Σ^* è numerabile, e di conseguenza $\{< M > \mid M \in TM\}$ è numerabile. ■

Proposizione 2 : L'insieme dei linguaggi non è numerabile.

Dimostrazione 2 : Sia $\mathcal{L}$ l'insieme di tutti i linguaggi le cui stringhe sono binarie. Un generico linguaggio L è in Σ^* dove $\Sigma = \{0, 1\}$. Definisco per ogni linguaggio L una stringa binaria $\mathcal{X}_L$ di lunghezza infinita tale che

l'i-esimo carattere di $\mathcal{X}_L$ è 1 se l'i-esimo elemento di Σ^* (secondo un arbitrario ordinamento) è in L.
Altrimenti è 0.

Questa applicazione $f : \mathcal{L} \to \mathcal{B}$ è biettiva, $\mathcal{B}$ è l'insieme delle stringhe di lunghezza infinita, che sappiamo non essere numerabile, ma allora anche $\mathcal{L}$, ossia un sotto-insieme del totale dei linguaggi non è numerabile, implica che l'insieme dei linguaggi non è numerabile. ■

Teorema fondamentale : Esistono linguaggi che non sono turing riconoscibili.

Dimostrazione : Ogni macchina di turing ha ad esso associato un solo linguaggio. Essendo l'insieme dei linguaggi non numerabile, ed essendo l'insieme delle TM numerabile, esiste almeno un linguaggio che non si può associare a nessuna TM $\implies$ esistono linguaggi non riconoscibili. ■

Teorema : A_{TM} **non** è turing-decidibile.

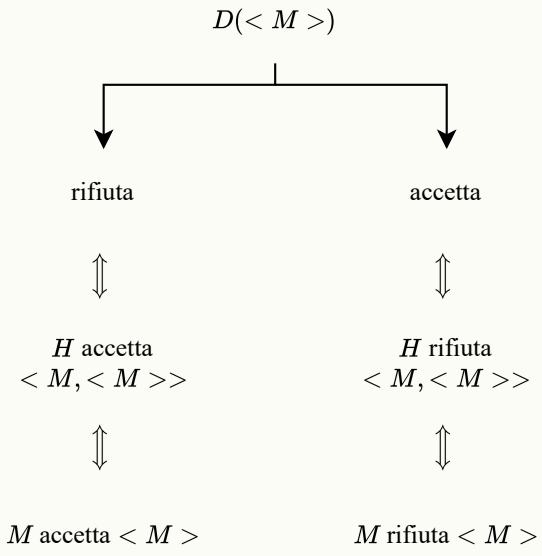
Dimostrazione : Si utilizzerà la diagonalizzazione. Si pone per assurdo che esiste una TM H che è un decisore per A_{TM}, ovvero

$$H(< M, w >) = \begin{cases} \text{accetta se } M \text{ accetta } w \\ \text{rifiuta se } M \text{ rifiuta } w \end{cases}$$

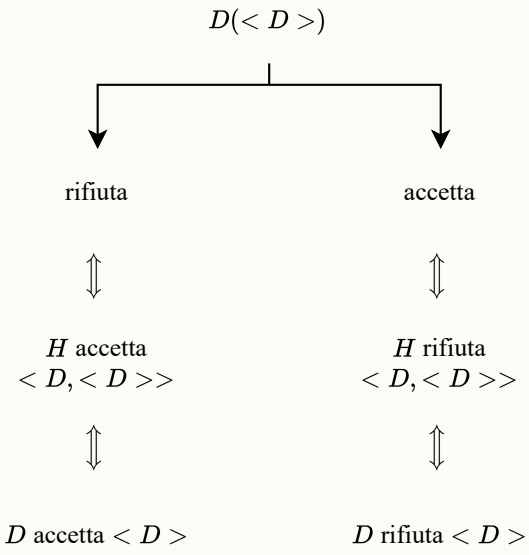
Si definisce una nuova TM D che prende come input una TM e si comporta nel seguente modo

1. Su input $< M >$, esegue $H(< M, < M >>)$
2. Se H accetta, D rifiuta. Se H rifiuta, D accetta.

Ricordando il comportamento di H, si ha la seguente situazione "transitiva"



Se alla TM D si da in input $< D >$ si ricade nella seguente situazione.



Ma allora D accetta $< D >$ solo e soltanto se D rifiuta $< D >$. È una palese contraddizione, è quindi impossibile che H sia un decisore per A_{TM}. ■

Definizione (co-turing Riconoscibilità) : L è co-turing riconoscibile (o più semplicemente, co-riconoscibile) se $\overline{L}$ è turing riconoscibile.

Teorema : L è decidibile se e solo se L è sia riconoscibile che co-riconoscibile.

Dimostrazione : Si dimostrano separatamente le due implicazioni.
$\boxed{\implies}$: Per ipotesi L è decidibile, allora anche $\overline{L}$ lo è, quindi sia L che $\overline{L}$ sono riconoscibili.
$\boxed{\impliedby}$: Per ipotesi L ed $\overline{L}$ sono riconoscibili. Siano M_1, M_2 le TM che li riconoscono. Si costruisce una nuova TM M tale che

1. su input w esegue parallelamente w su M_1 ed M_2
2. se M_1 accetta, M accetta. se M_2 accetta, M rifiuta.

$\forall w$ si ha che $w \in L \vee w \in \overline{L}$, solamente una fra M_1, M_2 può accettare w, quindi M non può mai andare in loop $\implies$ M è un decisore per L. ■

Corollario : $\overline{A_{TM}}$ non è turing riconoscibile.

2.4 Riducibilità

La riducibilità è una tecnica che permette di dimostrare la non decidibilità/non riconoscibilità di molti linguaggi. Informalmente, dati due linguaggi A e B, diremo che A **si riduce a** B, e denoteremo

$$A \leq B$$

Se esiste un algoritmo (una TM) per B che risolve (decide) anche A.

trovare una soluzione per A **non può essere** più difficile di trovare una soluzione per B

Si consideri il seguente linguaggio

$$HALT_{TM} = \{< M, w > \mid M \in TM \wedge M(w) \text{ termina}\}$$

Data una TM si deve stabilire se termina o continua all'infinito su un certo input.

Teorema (Halting Problem) : $HALT_{TM}$ non è decidibile.

Dimostrazione : Per assurdo, sia R un decisore per $HALT_{TM}$. Si definisce una TM S tale che

1. prende come input una TM ed una stringa $< M, w > \in A_{TM}$
2. esegue R su $< M, w >$
3. se R accetta (quindi si ha la certezza che M termina), S simula M su w ed accetta solamente se M accetta w.

Quindi S è un decisore per tutte le coppie $< M, w >$ ossia A_{TM}. Ma A_{TM} non è decidibile, è quindi impossibile che esista un decisore per $HALT_{TM}$. ■

Si consideri

$$E_{TM} = \{< M > \mid M \in DFA \wedge L(M) = \emptyset\}$$

Proposizione : E_{TM} non è decidibile.

Dimostrazione : Si pone per assurdo che R decide E_{TM}. Si definisce una nuova TM S che prende come input gli elementi di A_{TM} e si comporta come segue

1. costruisce una nuova TM M', il cui comportamento è il seguente
   (a) prende come input una stringa x
   (b) se $x \neq w$ rifiuta
   (c) se $x = w$ esegue M su input w
   (d) M' accetta se M accetta
2. esegue R su input $< M' >$
3. se R accetta, S rifiuta. Se R rifiuta, S accetta.

Essendo R un decisore, allora S è un decisore di A_{TM}, ma A_{TM} non è decidibile, è quindi impossibile che esista un decisore per E_{TM}. ■

Si vuole formalizzare il concetto di riduzione, intuitivamente si può dire che la riduzione è definita da una funzione, che negli obiettivi di questo corso sarà calcolata da una TM.

Definizione : Una funzione $f : \Sigma^* \to \Sigma^*$ si dice **calcolabile** se esiste una TM che su input w termina scrivendo sul nastro $f(w)$. Spesso tale TM ha un nastro dedicato per l'output della funzione.

Nell'halting problem è stata definita una TM a tal scopo. La definizione di riduzione che verrà data non rappresenta il concetto in senso lato, bensì è la sua formulazione più semplice detta *mapping reduction*, sufficiente per gli obiettivi di questo corso.

Definizione (Riduzione) : Siano A e B due generici linguaggi. Si dice che A è **riducibile** a B, e si denota

$$A \leq_m B$$

Se *esiste* una funzione calcolabile $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall w \in \Sigma^*, \quad w \in A \iff f(w) \in B$$

Il seguente teorema è di fondamentale importanza quando si vuole dimostrare la non decidibilità di un linguaggio.

Teorema : Se $A \leq_m B$ e B è decidibile, allora anche A è decidibile.

Dimostrazione : Sia M_B un decisore per B, si definisce una TM A_M tale che

1. su input $w \in \Sigma^*$, calcola $f(w)$
2. esegue M_B su input $f(w)$
3. Se $M_B(f(w))$ accetta, M_A accetta, altrimenti rifiuta.

Essendo che, per ogni w si ha che $w \in A$ solo se $f(w) \in B$, M_A potrà sempre o accettare o rifiutare (data l'ipotesi che M_B è un decisore), quindi M_A è un decisore per A. ■

Corollario : Se $A \leq_m B$ e A non è decidibile, allora anche B non è decidibile.

2.4.1 Applicazioni della Riducibilità

In questa sezione verranno trattati differenti esempi di come la *riducibilità* può essere usata per dimostrare la non decidibilità di molti linguaggi, e verranno presentati *alcuni risultati* che estendono le applicazioni, in particolare, quando si trattano linguaggi non riconoscibili. È consigliato allo studente che deve sostenere l'esame di porre particolare attenzione a questa sezione.

Esempio 1

Si vuole dimostrare in maniera formale l'indecidibilità di $HALT_{TM}$. La notazione $M(w) \neq \infty$ indicia che la TM M su input w non va in loop, e termina.

$$HALT_{TM} = \{< M, w > \mid M \in TM \wedge M(w) \neq \infty\}$$

Basta dimostrare che

$$A_{TM} \leq_m HALT_{TM}$$

Essendo A_{TM} indecidibile, ciò comporterebbe anche l'indecidibilità di $HALT_{TM}$. Si definisce una funzione $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall < M, w > \in \Sigma^*, \quad < M, w > \in A_{TM} \iff f(< M, w >) \in HALT_{TM}$$

f deve essere calcolabile, quindi si definisce una TM che calcola f, ed esegue i seguenti passi

1. Su input $< M, w >$ costruisce una TM ausiliaria M'
2. Tale M' su un generico input x, esegue i seguenti passi
   (a) Esegue M su input x
   (b) Se $M(x)$ accetta, allora M' accetta, se $M(x)$ rifiuta, entra volontariamente in uno stato di loop.
3. L'output della funzione sarà $< M', w >$

$\boxed{\implies}$: se $< M, w > \in A_{TM}$, allora M accetta w, pertanto M' a sua volta accetterà w, terminando. Si ha quindi che $< M', w > \in HALT_{TM}$. La prima implicazione della riducibilità è dimostrata.

$\boxed{\impliedby}$: se $< M, w > \notin A_{TM}$, allora $M(w)$ o rifiuta, o va in loop.

- Se $M(w)$ rifiuta, per definizione, M' entra in uno stato di loop.
- Se $M(w)$ va in loop, anche M' (che la esegue) va in loop.

Si ha quindi che, se $< M, w > \notin A_{TM}$, la TM M' va in loop su w, quindi non termina, allora

$$f(< M, w >) =< M', w > \notin HALT_{TM}$$

La seconda implicazione della riducibilità è dimostrata. Quindi $A_{TM} \leq_m HALT_{TM}$. Ne consegue che $HALT_{TM}$ è indecidibile.

Esempio 2

Si vuole dimostrare l'indecidibilità di EQ_{TM}.

$$EQ_{TM} = \{< M_1, M_2 > \mid M_1, M_2 \in TM \wedge L(M_1) = L(M_2)\}$$

Basta dimostrare che

$$E_{TM} \leq_m EQ_{TM}$$

Si definisce una funzione $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall < M > \in \Sigma^*, \quad < M > \in E_{TM} \iff f(< M >) \in EQ_{TM}$$

f deve essere calcolabile, quindi si definisce una TM che calcola f, ed esegue i seguenti passi

1. Su input $< M >$, definisce una TM ausiliaria M', la cui computazione è descritta come segue
   (a) M' su un qualsiasi input x rifiuta sempre.
2. L'output della funzione sarà $< M, M' >$

$\boxed{\implies}$: Se $M \in E_{TM}$, allora $L(M) = \emptyset$. Essendo che M' rifiuta su ogni input, anche il suo linguaggio è vuoto, quindi $L(M) = L(M')$, ne consegue che $< M, M' > \in EQ_{TM}$.

$\boxed{\impliedby}$: Se $M \notin E_{TM}$, allora $L(M) \neq \emptyset$, Essendo che M' rifiuta su ogni input, il suo linguaggio è vuoto, quindi $L(M) \neq L(M') = \emptyset$, ne consegue che $< M, M' > \notin EQ_{TM}$.

Risulta naturale chiedersi: È possibile utilizzare la riducibilità per dimostrare anche la non riconoscibilità di un linguaggio? Il seguente teorema risponde a tale domanda.

Teorema : Se $A \leq_m B$ e B è riconoscibile, allora anche A è riconoscibile.

Corollario : Se $A \leq_m B$ e A non è riconoscibile, allora anche B non è riconoscibile.

Essendo che A_{TM} è riconoscibile ma non decidibile, per il teorema sulla co-riconoscibilità è noto che il linguaggio $\overline{A_{TM}}$ non è riconoscibile. È possibile utilizzare $\overline{A_{TM}}$ e la riducibilità per dimostrare la non riconoscibilità di molti linguaggi. $\overline{A_{TM}}$ è però un linguaggio ambiguo con la quale lavorare, risulta più funzionale considerare direttamente A_{TM} grazie al seguente teorema.

Teorema : Se $A \leq_m B$, allora $\overline{A} \leq_m \overline{B}$.

Dimostrazione : La funzione f rimane invariata dato che $\forall w \in \Sigma^*$ si ha che

$$w \in A \iff f(w) \in B$$

$$w \in \overline{A} \iff f(w) \in \overline{B}$$

■

Le conseguenze sono ovvie, se $A \leq_m B$ e $\overline{A}$ non è riconoscibile, allora anche $\overline{B}$ non è riconoscibile. Per dimostrare la non riconoscibilità di un generico linguaggio A, basterà dimostrare A_{TM} si riduce a $\overline{A}$.

Esempio 3

Si vuole dimostrare la non riconoscibilità di EQ_{TM}. Basta dimostrare che

$$A_{TM} \leq_m \overline{EQ_{TM}}$$

Si definisce una funzione $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall < M, w > \in \Sigma^*, \quad < M, w > \in A_{TM} \iff f(< M, w >) \in \overline{EQ_{TM}}$$

In pratica, se M accetta w, allora $f(< M, w >)$ deve restituire due TM che hanno linguaggi differenti. f deve essere calcolabile, quindi si definisce una TM che calcola f, ed esegue i seguenti passi

1. Su input $< M, w >$, definisce due TM ausiliarie M_1, M_2, tali che
   - M_1 rifiuta sempre
   - M_2 accetta solo se $M(w)$ accetta, altrimenti rifiuta.
2. l'output della funzione è $< M_1, M_2 >$

$\boxed{\implies}$: Se $< M, w > \in A_{TM}$, essendo che $M(w)$ accetta si ha che

- M_1 rifiuta sempre
- M_2 accetta sempre

Quindi $L(M_1) \neq L(M_2) \implies < M_1, M_2 > \in \overline{EQ_{TM}}$.

$\boxed{\impliedby}$: Se $< M, w > \notin A_{TM}$, essendo che $M(w)$ rifiuta si ha che

- M_1 rifiuta sempre
- M_2 rifiuta sempre

Quindi $L(M_1) = L(M_2) = \emptyset \implies < M_1, M_2 > \in EQ_{TM} \implies < M_1, M_2 > \notin \overline{EQ_{TM}}$. Ne consegue che EQ_{TM} non è riconoscibile.

Esempio 4

Si vuole dimostrare la non decidibilità di L definito come segue

$$L = \{< M > \mid M \in TM \wedge w \in L(M) \implies w \in \{0,1\}^* \wedge |w|\%2 = 1\}$$

Ossia il linguaggio di tutte le TM che accettano esclusivamente stringhe binarie di lunghezza dispari. Basta dimostrare che

$$A_{TM} \leq_m L$$

Si definisce una funzione $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall < M, w > \in \Sigma^*, \quad < M, w > \in A_{TM} \iff f(< M, w >) \in L$$

f deve essere calcolabile, quindi si definisce una TM che calcola f, ed esegue i seguenti passi

1. Su input $< M, w >$, definisce una TM ausiliaria M', tale che:
2. Su input x, se $|x|\%2 = 0$ rifiuta.
3. Altrimenti, accetta se e solo se M accetta w.
4. L'output sarà M'.

$\boxed{\implies}$: Se $< M, w > \in A_{TM}$, essendo che $M(w)$ accetta si ha che M' accetterà tutte e sole le stringhe binarie di lunghezza dispari, quindi $f(< M, w >) = M' \in L$.

$\boxed{\impliedby}$: Se $< M, w > \notin A_{TM}$, M' rifiuterà a prescindere, si avrà che $L(M') = \emptyset \implies M' \notin L$. Ne consegue che L è indecidibile.

Esempio 5

Si vuole dimostrare la non decidibilità di L definito come segue

$$L = \{< M > \mid M \in TM \wedge L(M) = \{0^n 1^n 0^n,\ n \geq 0\}\}$$

Basta dimostrare che

$$A_{TM} \leq_m L$$

Si definisce una funzione $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall < M, w > \in \Sigma^*, \quad < M, w > \in A_{TM} \iff f(< M, w >) \in L$$

f deve essere calcolabile, quindi si definisce una TM che calcola f, ed esegue i seguenti passi

1. Su input $< M, w >$, definisce una TM ausiliaria M', tale che:
2. Su input x, esegue $M(w)$.
3. Se $M(w)$ accetta, M' accetta solo se $x \in \{0^n 1^n 0^n,\ n \geq 0\}$, altrimenti rifiuta.
4. Se $M(w)$ rifiuta, M' rifiuta a prescindere.
5. L'output sarà M'.

$\boxed{\implies}$: Se $< M, w > \in A_{TM}$, essendo che $M(w)$ accetta si ha che M' accetterà solo se

$$x \in \{0^n 1^n 0^n,\ n \geq 0\}$$

quindi $L(M') = \{0^n 1^n 0^n,\ n \geq 0\} \implies f(< M, w >) = M' \in L$.

$\boxed{\impliedby}$: Se $< M, w > \notin A_{TM}$, M' rifiuterà a prescindere (o andrà in loop nel caso $M(w)$ vada in loop), si avrà che $L(M') = \emptyset \neq \{0^n 1^n 0^n,\ n \geq 0\} \implies M' \notin L$. Ne consegue che L è indecidibile.

Esempio 6

Si vuole dimostrare la non decidibilità di L definito come segue

$$L = \{< M > \mid M \in TM \wedge L(M) \supseteq \{w \mid w \text{ inizia con uno } 0\}\}$$

Ossia il linguaggio di tutte le TM il cui linguaggio contiene ogni stringa che inizia con 0. Basta dimostrare che

$$A_{TM} \leq_m L$$

Si definisce una funzione $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall < M, w > \in \Sigma^*, \quad < M, w > \in A_{TM} \iff f(< M, w >) \in L$$

f deve essere calcolabile, quindi si definisce una TM che calcola f, ed esegue i seguenti passi

1. Su input $< M, w >$, definisce una TM ausiliaria M', tale che:
2. su ogni input x esegue $M(w)$, e se questa accetta, M' accetta, altrimenti rifiuta.
3. L'output sarà M'.

$\boxed{\implies}$: Se $< M, w > \in A_{TM}$, essendo che $M(w)$ accetta si ha che M' accetterà ogni input, quindi $L(M') = \Sigma^*$ ed essendo l'insieme di ogni stringa, conterrà anche ogni stringa che inizia con zero, allora $< M' > \in L$.

$\boxed{\impliedby}$: Se $< M, w > \notin A_{TM}$, M' rifiuterà (o andrà in loop nel caso $M(w)$ vada in loop), si avrà che $L(M') = \emptyset$, quindi non conterrà nemmeno una stringa che inizia con 0, allora $< M' > \notin L$. Ne consegue che L è indecidibile.

Esempio 7

Si vuole dimostrare la non decidibilità di U definito come segue

$$U = \{< T, T' > \parallel T, T' \in TM \wedge L(T) \cup L(T') = \Sigma^*\}$$

Ossia il linguaggio di tutte le coppie di TM la cui unione dei linguaggi equivale all'insieme di tutte le stringhe. Basta dimostrare che

$$A_{TM} \leq_m U$$

Si definisce una funzione $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall < M, w > \in \Sigma^*, \quad < M, w > \in A_{TM} \iff f(< M, w >) \in U$$

f deve essere calcolabile, quindi si definisce una TM che calcola f, ed esegue i seguenti passi

1. Su input $< M, w >$, definisce una TM ausiliaria T che rifiuta per ogni input x.
2. Definisce poi una TM T' che accetta solo se $M(w)$ accetta.
3. L'output è $< T, T' >$

$\boxed{\implies}$: Se $< M, w > \in A_{TM}$, essendo che $M(w)$ accetta si ha che T' accetterà ogni stringa ed il suo linguaggio sarà uguale a tutte le stringhe. T invece avrà linguaggio vuoto. $L(T) \cup L(T') = \emptyset \cup \Sigma^* = \Sigma^* \implies < T, T' > \in U$.

$\boxed{\impliedby}$: Se $< M, w > \notin A_{TM}$, M' rifiuterà (o andrà in loop nel caso $M(w)$ vada in loop), si avrà che $L(T') = \emptyset$, inoltre anche T ha linguaggio vuoto, quindi $L(T) \cup L(T') = \emptyset \neq \Sigma^* \implies < T, T' > \notin U$. Ne consegue che U è indecidibile.

2.5 I Teoremi di Incompletezza

I risultati riguardanti i teoremi di Gödel possono essere ottenuti utilizzando le TM. Negli anni è sempre stata presente l'esigenza di poter dimostrare le proposizioni matematiche, già nel 300 a.c. Euclide stipulò gli assiomi della geometria piana.

Nel XIX secolo i matematici si sono interrogati riguardo la formalità degli assiomi di Euclide, in particolare, è iniziata la ricerca verso una formulazione più rigorosa della matematica che ha anche sancito la nascita della logica, che fornisce una serie di connettivi, con i quali si possono derivare gli assiomi ed ottenere nuove proposizioni.

Si definisce **sistema di prova** un insieme di assiomi uniti alla logica del primo ordine, con la quale è possibile dimostrare ogni proposizione derivante da essi, anche in maniera automatica (calcolabile da un algoritmo).

Denoteremo Π un generico sistema di prova, in particolare si può definire in funzione della calcolabilità. Riguardo Π si ha che

- Per ogni affermazione x, esiste una codifica $< x >$ sottoforma di stringa.
- Per ogni dimostrazione w, esiste una codifica $< w >$ sottoforma di stringa.
- Esiste una TM V che è un decisore, per cui $V(< x, w >)$ accetta se e solo se w è una dimostrazione di x.

Definizione : un'affermazione x è **dimostrabile** se $\exists w$ tale che $V(< x, w >)$ accetta.

Definizione : un'affermazione x è **indipendente** se, ne x ne $\overline{x}$ sono dimostrabili.

Un sistema di prova Π può godere di differenti proprietà

- Π è **consistente** se, per ogni affermazione x, al massimo 1 fra x e $\overline{x}$ è vera, ossia, non esiste un affermazione vera, la cui negata è a sua volta vera.
- Π è **valido** se ogni affermazione dimostrabile è vera.
- Π è **completo** se, per ogni affermazione x, almeno 1 fra x e $\overline{x}$ è vera.
- Π è **incompleto** se esiste un affermazione x indipendente.

Osservazione : Se Π è valido, allora è anche consistente. Se Π è valido e completo, per ogni affermazione x, esattamente una fra x e $\overline{x}$ è vera, e ciò che è dimostrabile è anche vero.

I teoremi di incompletezza riguardano tali proprietà, il primo, afferma che un sistema di prova consistente è necessariamente incompleto, ossia che esiste almeno un'affermazione indipendente, che non si può dimostrare. Il secondo, afferma, per un sistema di prova Π, che l'affermazione

$x =$ "Π è consistente"

Non è dimostrabile. Prima di dimostrare i due teoremi, è necessario introdurre due TM relative ad un generico sistema di prova Π.

Definizione : Dato un sistema Π, si definisce una TM P, il cui comportamento è descritto come segue

1. Data in input un affermazione $< x >$
2. Controlla per ogni $k \in 1, 2, 3 \dots$
   - Per ogni possibile stringa w tale che $|w| = k$
   - Se $V(< x, w >)$ accetta, P scrive sul nastro di output $< w >$.

si definisce poi una TM R, il cui comportamento è descritto come segue

1. Data in input un affermazione $< x >$
2. Controlla per ogni $k \in 1, 2, 3 \dots$
   - Per ogni possibile stringa w tale che $|w| = k$
   - Se $V(< x, w >)$ accetta, R accetta, altrimenti rifiuta.

Si definisce il linguaggio

$$L_{provable} = \{< x > \mid x \text{ è dimostrabile } \}$$

Alcune osservazioni

- Se Π è valido, l'output di P è vero, anche se non è assicurato che R termini
- Se Π è completo, allora R è un decisore per $L_{provable}$. In particolare, per ogni $< x >$, se x è vera, $R(< x >)$ accetta, altrimenti $R(< x >)$ rifiuta

Verrà considerato adesso un caso speciale del primo teorema di incompletezza, abbiamo visto come un sistema valido è anche consistente. Il primo teorema di Gödel riguarda i sistemi consistenti, il caso speciale che verrà trattato riguarda i sistemi validi.

Teorema (caso speciale) : Sia Π un generico sistema di prova. Se Π è valido, allora non è completo.

Dimostrazione : Si pone per assurdo che Π è valido e completo, si definisce una TM D per il linguaggio

$$HALT_{TM} = \{< M, w > \mid M \in TM \wedge M(w) \neq \infty\}$$

Essendo Π completo, la TM R è un decisore per $L_{provable}$, quindi, data una generica TM M ed una generica stringa w, l'affermazione

$$< x >= "M(w) \neq \infty"$$

È decisa da R. La TM D è definita come segue

$$D(< M, w >) = R(< "M(w) \neq \infty" >)$$

Ma allora D è chiaramente un decisore per $HALT_{TM}$, ma questo linguaggio è indecidibile, è quindi impossibile che R sia un decisore per $L_{provable}$, allora Π non può essere completo. ■

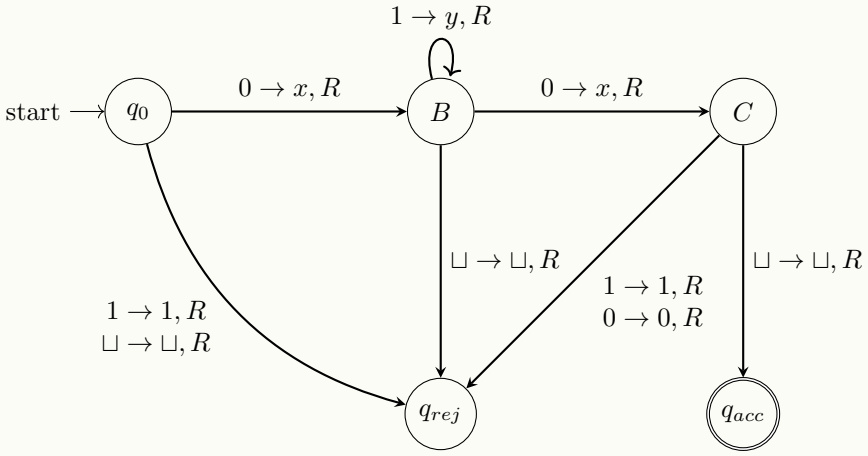
È necessario sapere che ZFC è il sistema di prova per la teoria degli insiemi, comprende gli assiomi standard della teoria assiomatica degli insiemi su cui, insieme con l'assioma di scelta, si basa tutta la matematica ordinaria secondo formulazioni moderne. In passato, si pensava che ogni affermazione in ogni branca della matematica potesse essere dimostrata a partire da tali assiomi.

In particolare, se M è una generica TM, e w è una stringa, se $M(w)$ termina, la proposizione

"M termina con input w"

È dimostrabile *canonicamente* tramite gli assiomi del sistema ZFC, dato che basta controllare la traccia di esecuzione di M su w, e constatare che ad un certo punto la TM si ferma. Ovviamente ciò è possibile se si da come ipotesi che M non vada in loop su w.

Si consideri la TM mostrata in figura



È chiaro che su input $w = 0\ 0\ \sqcup$ la TM termina, per dimostrarlo è necessario mostrare la traccia di esecuzione della macchina sulla stringa data.

Lemma : Sia M una generica TM e w una stringa, se M termina su w, l'affermazione $< M(w) \neq \infty >$ è dimostrabile.

Teorema di Incompletezza I : Sia Π un generico sistema di prova. Se Π è consistente, allora non è completo.

Dimostrazione : Si definisce un'apposita TM D il cui comportamento è descritto come segue

1. su input $< M >$, dove M è una macchina di Turing
2. si controlla per ogni $k \in 1, 2, 3, \ldots$
   - per ogni stringa w tale che $|w| = k$
   - Se w è una prova per l'affermazione "$M(< M >) \neq \infty$", allora D si sposta volontariamente in uno stato q_L di loop
   - Se w è una prova per l'affermazione "$M(< M >) = \infty$", allora D termina.

Si applica la diagonalizzazione e si considera il caso particolare $D(< D >)$. Essendo Π per ipotesi consistente, si ha che al massimo un'affermazione fra

$$\begin{cases} D(< D >) = \infty \\ D(< D >) \neq \infty \end{cases}$$

è dimostrabile. Si considerano i due casi separatamente

- Se "$D(< D >) = \infty$" è dimostrabile, allora, per costruzione di D, essa stessa con input se stessa, ossia $D(< D >)$ termina, ossia
  $$D(< D >) \neq \infty$$
  allora per il *Lemma* esiste una dimostrazione per $D(< D >) \neq \infty$, quindi $D(< D >) \neq \infty$ è dimostrabile. Si ha che sia
  $$\begin{cases} D(< D >) = \infty \\ D(< D >) \neq \infty \end{cases}$$
  sono entrambe dimostrabili, quindi Π non è consistente.

- Se "$D(< D >) \neq \infty$" è dimostrabile, allora per costruzione $D(< D >)$ durante l'esecuzione troverà tale dimostrazione w ed entrerà nello stato di loop q_L, ma allora, tale traccia di computazione è proprio una dimostrazione dell'affermazione

  $$D(< D >) = \infty$$

  Si ha che sia

  $$\begin{cases} D(< D >) = \infty \\ D(< D >) \neq \infty \end{cases}$$

  sono entrambe dimostrabili, quindi Π non è consistente.

In conclusione, si è dimostrato che il generico sistema di prova Π non può essere sia completo che consistente. ■

Teorema di Incompletezza II : Sia Π un sistema di prova *consistente*, l'affermazione

"Π è consistente"

Non è dimostrabile.

Dimostrazione : Si consideri la TM D definita nella dimostrazione precedente, si è visto come questa, non riesca a trovare una dimostrazione per nessuna delle due affermazioni

$$\begin{cases} D(< D >) = \infty \\ D(< D >) \neq \infty \end{cases}$$

Ciò, signigica che $D(< D >)$ non trovando una dimostrazione, non termina e va in loop, ma allora l'affermazione $D(< D >) = \infty$ si può codificare ed è vera. Ne consegue che

$$\Pi \text{ consistente} \implies D(< D >) = \infty$$

Ma come si è mostrato in precedenza, $D(< D >) = \infty$ è un affermazione non dimostrabile, quindi è necessariamente vero che anche l'affermazione che la implica, ossia "Π consistente " sia indimostrabile.■

Esistono affermazioni vere e non dimostrabili

CAPITOLO

3

COMPLESSITÀ TEMPORALE

Questo capitolo si occuperà di trattare le risorse necessarie alle TM per risolvere i problemi, con risorse, si definiscono

- tempo
- spazio
- numero di processori
- randomicità
- altre

Ci si occuperà della trattazione di tempo e spazio, la domanda scatenante la teoria della complessità è: "*quante risorse occorrono per decidere i problemi?*".

I problemi che si affrontano sono identificabili in delle *classi* che li contengono, e sono classificati in base al tempo (o spazio) necessario per risolverli. La domanda fondamentale della teoria della complessità (che verrà formalizzata in seguito) è

> trovare una soluzione ad un problema, è tanto difficile quanto verificare che una soluzione data sia corretta?

Un altra domanda cardine è

> se è possibile risolvere un problema utilizzando poco spazio, è possibile risolverlo anche utilizzando poco tempo?

Definizione (costo computazionale) : : Sia M un decisore, il **tempo di esecuzione** di M è una funzione $t : \mathbb{N} \to \mathbb{N}$ definita come segue

$$t(n) = \max_{\substack{x \in \Sigma^* \wedge \\ |x| = n}} (\text{step di computazione di } M)$$

Sia x la stringa in input di lunghezza n che richiede il maggior numero di passi di computazione per essere decisa. Tale numero di passi di computazione, è proprio il costo computazionale di M su un input di lunghezza n. È centrale nella teoria della complessità lo studio di t al variare di n.

Definizione (O-grande) : : Siano f e g due funzioni definite su un generico sottoinsieme di $\mathbb{R}$, si dice che

$$f(x) \in O(g(x))$$

se e solo se

$$\exists x_0 \in \mathbb{R},\ c > 0 \ \text{ tale che } \ |f(x)| \leq c|g(x)|\ \forall x > x_0$$

Notazione : Se $f(x) \in O(g(x))$, verrà scritto $f(x) = O(g(x))$.

Nei capitoli precedenti sono state considerate delle varianti della TM, che possono essere simulato e rese equivalenti ad una TM normale. Tali varianti vanno rivalutate da un punto di vista della risorsa del tempo.

3.1 Classi di Complessità

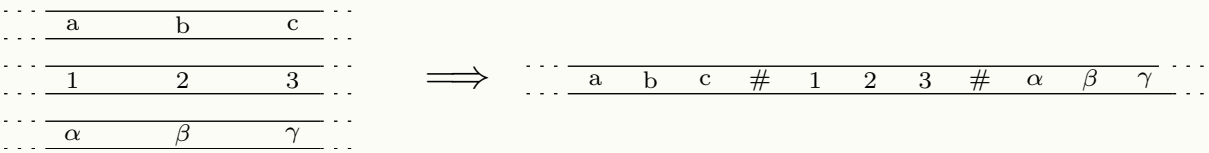
Esempio : Si consideri la TM che ad ogni passo di computazione può rimanere ferma sul nastro, la cui funzione di transizione è definita tale che

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

Questa poteva essere resa equivalente ad una TM classica, aggiungendo un semplice stato in cui la TM simula l'azione S spostandosi a destra e poi a sinistra. È chiaro che una TM classica che simula la TM che può stare ferma, dovrà compiere al più il doppio dei passi computazionali.

Ne consegue che, se M è una TM che può stare ferma, il cui costo computazionale è $t(n) = O(t(n))$, allora una TM classica che la simula avrà costo computazionale $2t(n) = O(t(n))$.

Esempio : Si consideri la TM multi nastro, si vuole calcolare l'overhead quando si considera una sua simulazione con TM classica.



Si può simulare con una TM classica come esposto nella sezione 2.1.2. Se una TM multi nastro M ha costo computazionale $t(n)$, allora la lunghezza di ogni nastro sarà al massimo n. Sia M' la TM classica che simula M, l'inizializzazione costa sicuramente $t(n)$ in quanto è necessario considerare tutto il nastro, in oltre, per ogni step di computazione di M, M' dovrà considerare tutto il nastro per leggere i caratteri marcati ed eventualmente spostarne il contenuto. Ogni step di computazione su M' ha costo $t(n)$. Ne consegue che la complessità di M' sarà

$$t(n) + t(n)^2 = O(t(n)^2)$$

Il numero dei nastri quindi influisce (in maniera non trascurabile) sulla complessità.

Esempio : Il linguaggio delle stringhe palindrome è decidibile in $O(n)$ su una macchina con due nastri, su una TM a singolo nastro può essere deciso in $O(n^2)$.

Definizione : : Sia $t : \mathbb{N} \to \mathbb{N}$ una funzione, si definisce

$$Dtime(t(n)) = \left\{ L \mid \begin{array}{c} \exists M \in TM(\text{singolo nastro}) \text{ tale che} \\ L(M) = L \wedge M \text{ ha complessità } O(t(n)) \end{array} \right\}$$

È l'insieme dei linguaggi decidibili da TM che hanno complessità $O(t(n))$. Il linguaggio delle stringhe palindrome appartiene all'insieme $Dtime(n^2)$.

Definizione (Classe Polinomiale) : Si definisce P l'insieme

$$P = \bigcup_{k \in \mathbb{N}} Dtime(t^k)$$

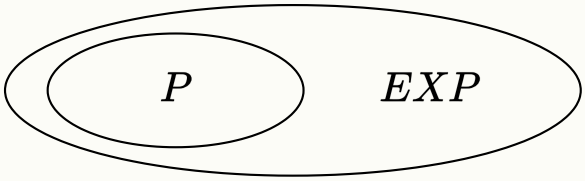
Ovvero, l'insieme di tutti i linguaggi che possono essere decisi in tempo polinomiale. L'insieme delle stringhe palindrome è in P.

Osservazione : La classe P è robusta rispetto le varianti di TM. Se un linguaggio è deciso in tempo polinomiale da una TM non classica, allora è deciso in tempo polinomiale anche da una TM classica.

Definizione (Classe Esponenziale) : Si definisce EXP l'insieme

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

Il teorema di *gerarchia dei tempi*, che verrà formalizzato e dimostrato in seguito, afferma che la classe P è un sottoinsieme della classe EXP.



Esistono linguaggi che sono in EXP, ma che, tramite algoritmi intelligenti, si può dimostrare sono anche in P. Verranno considerati linguaggi riguardanti i grafi, il cui input delle TM sarà la codifica di un grafo $x =< G = (V, E) >$, la complessità sarà misurata in funzione delle cardinalità di V ed E, se è polinomiale in queste, lo sarà anche in funzione della lunghezza della stringa di codifica.

Esempio : Si consideri il linguaggio

$$PATH = \{< G, s, t > \mid s, t \in V(G) \wedge \text{ esiste un cammino } s \to t \text{ in } G\}$$

Un cammino non è altro che una sequenza di vertici

$$v_0 \to V_1 \cdots \to v_l$$

il numero di vertici in ogni cammino è limitato da $n = |V(G)|$, è possibile controllare ogni cammino del grafo (sono n^n) e verificare che ce ne sia uno che inizia in s e termina in t. La complessità è $O(n^n) = O(2^{n \log(n)})$, è chiaro che $PATH \in EXP$.

Si può però considerare una DFS, in particolare una TM che esegue i seguenti passi

- marca lo stato s, ed inizia a fare una passeggiata sul grafo partendo da s
- marca ogni stato raggiungibile
- se alla fine t è marcato, la TM accetta, altrimenti rifiuta

Tale TM decide $PATH$ in tempo polinomiale, è chiaro che $PATH \in P$.

Esempio : Si consideri il linguaggio

$$2COL = \{< G > \mid G \text{ è 2-colorabile } \}$$

Un grafo è 2-colorabile se è possibile assegnare un colore ad ogni nodo (da un insieme di due colori possibili) in modo tale che ogni arco connetta sempre e solo nodi di colori diversi. Esistono al più 2^n possibili modi di colorare il grafo (ad ognuno degli n nodi si assegna uno fra i due colori), quindi, per ogni possibile colorazione, si può verificare che rispetti la condizione di 2-colorabilità in $O(n)$, ne consegue che esiste una TM che decide $2COL$ in $O(n2^n)$.

Si consideri il seguente algoritmo

- si seleziona un nodo e si colora

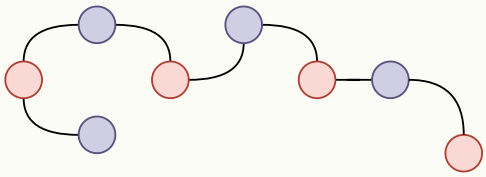


Figura 3.1: grafo 2-colorato

- si selezionano tutti i nodi adiacenti, e si colorano del colore opposto
- si prosegue per tutti i nodi
- alla fine, quando ogni nodo è colorato, è possibile verificare la condizione di 2-colorabilità in $O(n)$

L'algoritmo descritto decide $2COL$ in tempo polinomiale, è chiaro che $2COL \in P$.

Negli esempi mostrati, dei linguaggi in EXP si sono poi rivelati anche in P, ovviamente, non è assicurato che ciò accada. Esistono linguaggi che sono in EXP ma che sicuramente non sono in P, ed esistono linguaggi che sono in EXP, ma per cui non è stato dimostrato se sono o non sono in P.

Esempio : Si consideri il linguaggio

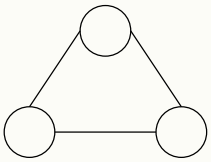
$$3COL = \{< G > \mid G \text{ è 3-colorabile } \}$$

Per un ragionamento analogo al precedente, $3COL \in EXP$, ma non è chiaro se è, oppure non è, in P. L'unico risultato dimostrato è il seguente.

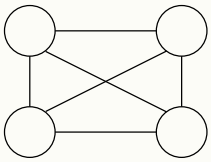
Teorema : $3COL \in Dtime(1.3289^n)$

Esempio : Un clique in un grafo, è un sottoinsieme totalmente connesso. Si consideri

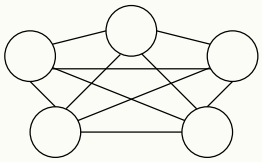
$$3CLIQUE = \{< G > \mid G \text{ contiene un triangolo (clique da 3 nodi)}\}$$



3-clique



4-clique



5-clique

È stato dimostrato che $3CLIQUE$ appartiene a P, in particolare

$$3CLIQUE \in Dtime(n^{2.39})$$

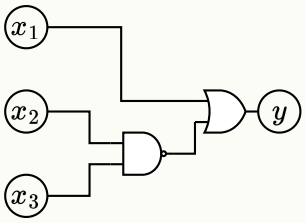
Non è ancora stato dimostrato che $3CLIQUE \in Dtime(n^2)$. In generale, non è noto se il linguaggio

$$kCLIQUE = \{< G > \mid G \text{ contiene un clique da } k \text{ nodi}\}$$

Sia o no in P, è almeno in $Dtime(n^k)$. La sua appartenenza o no alla classe polinomiale, risulterà nota quando verrà formalizzata la dipendenza di k da n. Se k non dovesse dipendere in nessun modo da n, allora $kCLIQUE$ sarebbe un elemento di P.

3.2 Soddisfacibilità

Un circuito logico, viene rappresentato con un grafo diretto ed aciclico, in cui sono presenti dei nodi input ($x_1, x_2 \dots x_n$ variabili booleane), delle porte logiche (AND, OR, NOT, ed altre) e dei nodi output ($y_1, y_2 \dots y_n$ variabili booleane che dipendono dagli input).



$$y = x_1 \vee \overline{(x_2 \wedge x_3)}$$

Un circuito logico può anche essere codificato con una stringa, si definisce *formula* un circuito che presenta un solo output. Una formula si identifica quindi con una funzione $C : \{0,1\}^n \to \{0,1\}$, e denotiamo $< C >$ la sua codifica sotto forma di stringa. Data una formula C ed un'assegnamento delle variabili $x_1, x_2 \dots x_n$, una TM può *valutare* il risultato dell'output, si definisce il linguaggio

$$CIRC - EVAL = \{< C, x > \mid C(x) = 1\}$$

Dove x è l'assegnamento delel variabili e $C(x)$ è il valore dell'output dato tale assegnamento. Tale linguaggio è in P, tipicamente, se m è il numero delle porte logiche, il costo computazionale della TM che decide $CIRC - EVAL$ è in $O(m \log(m))$.

Definizione (soddisfacibilità) : Un circuito logico C si dice **soddisfacibile** se esiste almeno un assegnamento delle variabili in input che renda uguale ad 1 l'output (o tutte le variabili in output). Di conseguenza, si definisce il linguaggio

$$SAT = \{< C > \mid \exists x \in \{0,1\}^n \text{ tale che } C(x) = 1\}$$

Se un circuito è soddisfacibile, piuttosto che dire che $< C >$ si trova nell'insieme SAT diremo che "C è SAT". Verrà trattato un sotto-insieme di tale linguaggio che considera esclusivamente i circuiti con 1 output

$$FORMULA - SAT = \{< C >\in SAT \mid C \text{ è una formula}\}$$

Questi due linguaggi sono chiaramente in EXP, dato che è possibile controllare ogni singola assegnazione di input (sono 2^n), e per ognuna di queste è possibile valutarne l'output (in tempo $poly(n)$)[1]. È quindi possibile definire una TM che decida $FORMULA - SAT$ in tempo $O(2^n poly(n))$.

Non è noto tutt'oggi se sia possibile decidere $FORMULA - SAT$ in tempo polinomiale, è però noto il seguente risultato

$$SAT \in P \iff P = NP$$

Questo teorema (fondamentale) verrà trattato e dimostrato formalmente in seguito.

Definiamo *clausola* una formula logica consistente in una o più variabili in OR fra loro

$$\begin{cases} (x_1 \vee x_2 \vee \overline{x_3}) & \text{è una clausola} \\ (x_1 \wedge x_2 \wedge \overline{x_3}) & \text{non è una clausola} \\ (x_1 \wedge x_2 \vee \overline{x_3}) & \text{non è una clausola} \\ (x_2) & \text{è una clausola} \end{cases}$$

Definizione (CNF) : una formula logica è in *CNF* (forma normale congiuntiva) se consiste in una serie di clausole in AND fra loro. Ad esempio

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_4 \vee x_2 \vee x_1) \wedge (\overline{x_5} \vee x_2)$$

Proposizione : ogni formula logica può essere scritto in forma normale congiuntiva. Si può quindi assumere (senza perdita di generalità) che una formula sia in CNF.

Si definisce l'insieme

$$CNF - SAT = \{< C >\in FORMULA - SAT \mid C \text{ è in forma normale congiuntiva}\}$$

[1] con $poly(n)$ si denota un costo polinomiale in funzione di n

è vero che

$$CNF - SAT \in EXP \qquad CNF - SAT \in P \iff P = NP$$

Le formule in CNF godono di una proprietà, la complessità computazionale di una TM che ne valuta il risultato è in funzione del numero delle variabili e del numero di clausole.

Si definisce anche il linguaggio

$$K - SAT = \{< C > \in CNF - SAT \mid \text{ ogni clausola ha al più } K \text{ variabili } \}$$

è vero che

$$K - SAT \in EXP \qquad K - SAT \in P \iff P = NP$$

Il migliore algoritmo che è stato definito per decidere $3 - SAT$ opera in tempo (circa) $O(1.34^n)$. In generale il costo per decidere $K - SAT$ dipende da K.

3.2.1 Complessità di $2 - SAT$

Si consideri il seguente sotto-insieme di $K - SAT$

$$2 - SAT = \{< C > \in CNF - SAT \mid \text{ ogni clausola ha al più 2 variabili } \}$$

Si vuole costruire un *grafo* (diretto) associato ad ogni formula. Una generica clausola di $2-SAT$ coinvolge due variabili ed è del tipo

$$(x_i \vee x_j)$$

Quest'ultima implica che

$$\overline{x_i} \to x_j$$
$$\overline{x_j} \to x_i$$

Definizione : Data una generica formula $\phi(x_1, x_2 \ldots, x_n)$ in $2 - SAT$, si definisce il *grafo associato* G come segue

- Vi è un nodo associato ad ogni variabile x_i e al negato $\overline{x_i}$
- Per ogni clausola $(x_i \vee x_j) \equiv \overline{x_i} \to x_j$, si aggiunge un arco $(\overline{x_i}, x_j)$

Ad esempio, per la formula

$$(\bar{x} \vee y) \wedge (\bar{y} \vee z) \wedge (x \vee \bar{z}) \wedge (y \vee z)$$

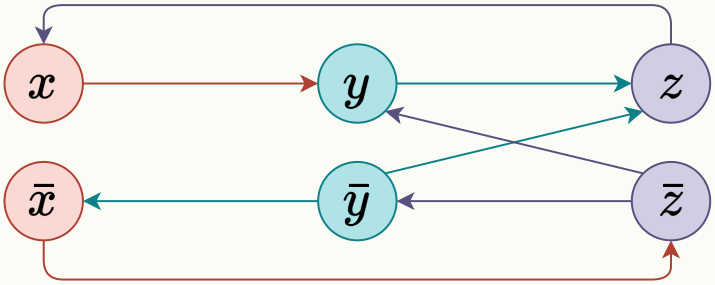
che è equivalente a

$$(x \to y) \wedge (y \to z) \wedge (\bar{x} \to \bar{z}) \wedge (\bar{y} \to z)$$

a sua volta equivalente a

$$(\bar{y} \to \bar{x}) \wedge (\bar{z} \to \bar{y}) \wedge (z \to x) \wedge (\bar{z} \to y)$$

si definisce il grafo associato



La seguente proposizione sarà fondamentale per dimostrare che una formula $2-SAT$ può essere decisa in tempo polinomiale.

Proposizione : Sia ϕ una formula in CNF in cui ogni clausola ha al più 2 variabili, e sia G il grafo associato, ϕ è soddisfacibile ($\phi \in 2-SAT$) *se e solo se*, nessuna componente fortemente connessa di G contiene sia una variabile che la sua negata.

Dimostrazione : Verranno dimostrate entrambe le direzioni del se e solo se.

$\boxed{\implies}$: Sia ϕ una formula in $2-SAT$, e sia G il grafo associato. Sia (a,b) un arco del grafo, ne consegue in ϕ c'è la clausola

$$a \to b \equiv \bar{a} \vee b$$

quindi se $a=1$, anche $b=1$. Tale fatto si può estendere ad un cammino nel grafo, se esiste un cammino

$$x_1, x_2 \ldots, x_k$$

ne consegue che

$$x_1 \to x_2 \cdots \to x_k$$

quindi

$$x_1 \to x_k$$

Si consideri una generica variabile x. Se x ed $\bar{x}$ sono in una stessa componente fortemente connessa, allora esiste un cammino da x ad $\bar{x}$ ed un cammino da $\bar{x}$ ad x

Si ha però che

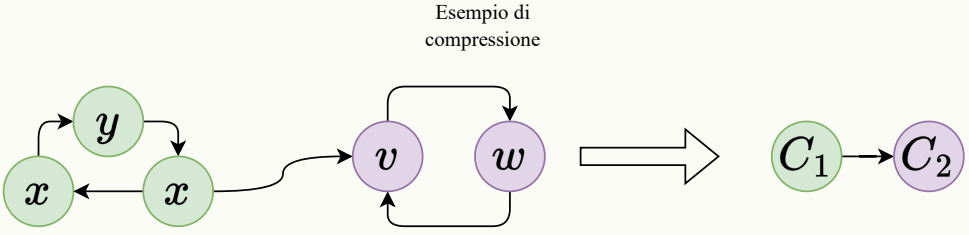
$$\begin{cases} x \to \bar{x} \equiv \bar{x} \vee \bar{x} \equiv \bar{x} \\ \bar{x} \to x \equiv x \vee x \equiv x \end{cases}$$

Ma allora in ϕ esiste sia la clausola x che la clausola $\bar{x}$, essendo ϕ in CNF essa vedrà queste due clausole in AND

$$x \wedge \bar{x}$$

Ma tale formula per qualunque assegnazione di x sarà sempre 0, quindi ϕ è insoddisfacibile, ma ciò contraddice l'ipotesi, è quindi impossibile che una componente fortemente connessa contenga sia x che $\bar{x}$.

$\boxed{\impliedby}$: Sia ϕ una formula in CNF con ogni clausola contenente al più 2 variabili, il cui grafo associato G ha una componente fortemente connessa in cui sono contenute sia x che $\bar{x}$. Sia G' il grafo in cui ogni nodo C_i rappresenta una componente fortemente connessa di G (si applica la compressione dei nodi).



Si consideri un ordinamento topologico di G'

$$(C_1, C_2 \ldots C_n)$$

dove $i<j$ indica che C_i viene prima di C_j nell'ordinamento. Si consideri il seguente *assegnamento* di variabili in input per ϕ

- Per ogni variabile x si ha che
  - se x è contenuta in una componente C_i, e $\bar{x}$ è contenuta in una componente C_j tale che $i \leq j$ (C_i non appare dopo C_j nell'ordinamento), allora la variabile x si assegna uguale a 0.
  - nel caso opposto, se x è contenuta in una componente C_i, e $\bar{x}$ è contenuta in una componente C_j tale che $i \geq j$ (C_i appare dopo C_j nell'ordinamento), allora la variabile x si assegna uguale a 1.

Dato tale assegnamento, è vero il seguente **fatto** :

> per nessun arco (a, b) in G, può succedere che a sia vero[2] e b sia falso. La dimostrazione è la seguente, si pone per assurdo che il *fatto* sia falso, che esiste quindi una situazione in cui
>
> - a è vero
> - b è falso
>
> supponiamo che C_i sia la componente in cui è contenuto a. Essendoci l'arco (a, b) esiste la clausola $\bar{a} \vee b$ ed esiste quindi anche l'arco $(\bar{b}, \bar{a})$, essendo a vero, si ha che $\bar{a}$ è falso. Ma allora, il nodo $\bar{a}$ compare in una componente C_j tale che $j < i$
>
> la componente in cui è contenuta $\bar{a}$ precede la componente in cui è contenuta a nell'ordinamento topologico
>
> Tale affermazione è vera dato l'assengamento particolare scelto per le variabili. Essendo b falso, ne consegue che $\bar{b}$ è contenuto in una componente C_k che viene dopo C_i, quindi
>
> $$k > i > j$$
>
> nell'ordinamento topologico
>
> $$C_k > C_i > C_j$$
>
> Ma allora la presenza dell'arco $(\bar{b}, \bar{a})$ contraddice l'ordinamento topologico, quindi è impossibile che il fatto sia falso.

Tale fatto implica che ϕ è SAT. Quindi entrambe le direzioni della proposizione sono dimostrate. ■

Data tale proposizione, per decidere se una formula è o non è in $2 - SAT$ basterà controllare il grafo associato trovando le componenti fortemente connesse, ciò si può fare (ad esempio, con l'algoritmo di Tarjan) in tempo polinomiale. Ciò porta al seguente risultato:

Teorema : $2 - SAT \in P$

3.3 La classe NP

I problemi presentati, ad esempio

- $3 - COL$
- $3 - SAT$
- ecc...

condividono una proprietà: data una *soluzione*, questa può essere verificata valida o meno in tempo polinomiale. Si vuole definire in questa sezione la classe NP tramite il *non determinismo*, esiste però, anche un'altra definizione altrettanto naturale.

Definizione : Dato un linguaggio L, una TM V è detta **verificatore** se, dato un input $< x, y >$ si ha che

[2] con a vero si intende che a è 1. con a falso si intende che a è 0

- $x \in L \iff \exists y$ tale che $V(< x, y >)$ accetta

V data una soluzione può verificare che questa sia contenuta o no nel linguaggio L. Si dice che V ha *tempo polinomiale* se

- L'esecuzione di V è in $O(|x|^k)$ per qualche $k \in \mathbb{Z}^+$
- $|y| = poly(|x|)$, ossia la cardinalità di y è polinomiale rispetto la cardinalità di x.

Definizione (NP) : La classe dei linguaggi NP contiene tutti i linguaggi L che ammettono un verificatore polinomiale.

Esempio : Si consideri il linguaggio $3 - COL$, una soluzione è data sotto-forma di grafo colorato, se n è il numero di nodi, è possibile controllare ogni coppia di nodi in $O(n^2)$, e verificare che, se questi sono adiacenti, sono anche di colori diversi. La verifica avviene in tempo polinomiale. Nonostante questo, la ricerca di una soluzione dato un grafo non colorato è un problema in EXP. Quindi $3 - COL$ è un problema NP. Si vuole dimostrare formalmente.

Proposizione : $3 - COL \in NP$

Dimostrazione : Sia V un verificatore per $3-COL$ che prende in input un grafo G ed una colorazione $y = (c_1, c_2 \dots c_n)$, con c_i colore dell'i-esimo nodo di G. La procedura di V è definita come segue

- controlla ogni coppia (c_i, c_j) se $(i, j) \in E(G)$
- rifiuta se e solo se $c_i \neq c_j$ per ogni coppia

Chiaramente V ha una complessità polinomiale in quanto al più controlla ogni arco del grafo, e verifica che y sia una soluzione. Se $V(< G, y >)$ accetta, allora y è una colorazione valida per G e G è 3-colorabile. ■

Teorema : $P \subseteq NP \subseteq EXP$

Dimostrazione : Per il teorema delle gerarchie dei tempo è noto che $P \neq EXP$, si ha che

$$P \neq NP \vee NP \neq EXP$$

Si consideri un linguaggio $L \in P$, ossia $\exists M$ che è una TM e $x \in L \iff M(x)$ accetta ed ha complessità polinomiale in funzione di $|x|$. Si definisce un verificatore $V(< x, y >)$ la cui procedura è definita come segue

- ignora l'input y
- esegue $M(x)$
- se M accetta su x, V accetta, altrimenti rifiuta.

è chiaro che V soddisfi le seguenti proprietà

- V ha la stessa complessità di M, è quindi polinomiale
- V è un verificatore per L

Per definizione $L \in NP$. Sapendo che esiste un verificatore polinomiale per L, si considera una TM che considera tutti i possibili certificati y tali che $|y| = poly(|x|)$, e accetta solo se $V(< x, y >)$ accetta, quindi tale TM è un decisore per L che opera in tempo esponenziale, quindi $L \in EXP$. ■

Si considererà ora la TM non deterministica per dare una definizione alternativa di NP. Una TM non deterministica ha la funzione di transizione che, piuttosto che portare da uno stato ad un altro, porta da uno stato ad un insieme di stati.

Definiamo la complessità computazionale di una TM non deterministica come il massimo numero di passi impiegati in funzione di n per ogni stringa x di lunghezza n, considerando il ramo di computazione più lungo.

Definizione : : Sia $t : \mathbb{N} \to \mathbb{N}$ una funzione, si definisce

$$Ntime(t(n)) = \left\{ L \mid \begin{array}{c} \exists M \in TM(\text{non deterministica}) \text{ tale che} \\ L(M) = L \wedge M \text{ ha complessità } O(t(n)) \end{array} \right\}$$

Per semplicità, denoteremo NTM una TM non deterministica.

Definizione : $NP = \bigcup_{k \in \mathbb{Z}^+} Ntime(n^k)$

Definizione : $NEXP = \bigcup_{k \in \mathbb{N}} Ntime(2^{n^k})$ Si vuole dare un esempio di linguaggio NP usando tale definizione, si consideri SAT, si definisce una NTM N che decide SAT, ossia $N(< \phi >)$ accetta se ϕ è soddisfacibile. La procedura di tale TM è definita come segue

- sia n il numero di variabili, si definisce un array x lungo n non inizializzato, e si definisce un contatore $i = 0$.
- "TOP" : si incrementa i di 1, se $i > n$, esegui la linea "CHECK", altrimenti esegui entrambe le seguenti linee in modo parallelo (diramazione della computazione)
  - $x[i] = 0$, ed esegui la linea "TOP"
  - $x[i] = 1$, ed esegui la linea "TOP"
- "CHECK" : accetta se e solo se $\phi(x) = 1$

Teorema : Le due definizioni di NP sono equivalenti.

Dimostrazione : Verranno dimostrate le due implicazioni separatamente.
$\boxed{\implies}$: Se L ha un verificatore polinomiale V, è possibile definire una NTM N la cui procedura è descritta come segue:

- su input x
- N utilizzando il non determinismo determina ogni possibile soluzione y tale che $|y| = poly(|x|)$
- N accetta solo se $V(< x, y >)$ accetta

è chiaro che N è un decisore non deterministico per L che opera in tempo polinomiale.

$\boxed{\impliedby}$: Se L è decidibile in tempo polinomiale da una NTM N, allora su input x, $poly(|x|)$ è un upper bound sui passi di computazione di ogni ramo parallelo di N. Si definisce un verificatore V, che opera su input y, definito come l'insieme dei passi deterministici di un ramo di N, in particolare

- $V(x, y)$ intepreta y come un insieme di (al più) lunghezza $poly(|x|)$ che simula $N(x)$ in modo deterministico seguendo le scelte di "fork" determinate da y.

$V(< x, y >)$ ha complessità $poly(|x|)$, ed inoltre $x \in L \iff N(x)$ accetta $\iff \exists y \mid V(< x, y >)$ accetta. ■

3.4 Riduzioni di Linguaggi NP

Sono stati presentati molti linguaggi in NP che sono anche in P, tramite la riduzione, si può dimostrare che un vasto insieme di linguaggi è in P se e solo se un singolo (e determinato) linguaggio è in P.

Teorema : $SAT \in P \implies 4 - COL \in P$

Dimostrazione : Per ipotesi, sia M_{SAT} la TM che decide SAT in tempo polinomiale, si definisce una TM M_{4COL} per decidere $4 - COL$. La riduzione deve "associare" l'affermazione

G è 4-colorabile

a

ϕ_G è soddisfacibile

Si definisce quindi una formula associata ad un grafo, che è soddisfacibile solo se il grafo è 4-colorabile.

Dato $G = (V, E)$, si definisce una formula ϕ_G con $2n$ variabili, dove $n = |V|$ è il numero di nodi.

$$\phi_G(x_1, x'_1, x_2, x'_2, \dots x_n, x'_n)$$

per ogni nodo i esiste la coppia di variabili x_i, x'_i. Il colore di un nodo è codificato da queste due variabili come segue

x_i	x'_i	colore di i
0	0	rosso
0	1	verde
1	0	giallo
1	1	blu

è necessario codificare il vincolo che, per ogni arco $(i, j) \in E$, si ha che i nodi i e j hanno colori diversi, ossia che

$$(x_i, x'_i) \neq (x_j, x'_j)$$

deve essere necessariamente vera per far si che ϕ_G sia soddisfacibile, in logica:

$$\phi_{i,j} = \overline{x_i \leftrightarrow x_j \wedge x'_i \leftrightarrow x'_j}$$

che è equivalente a

$$\phi_{i,j} = ((\bar{x}_i \vee x_j) \wedge (\bar{x}_j \vee x_i)) \wedge ((\bar{x}'_i \vee x'_j) \wedge (\bar{x}'_j \vee x'_i))$$

alla fine, la formula finale ϕ_G sarà l'and logico fra tutte queste clausole $\phi_{i,j}$ al variare di $(i, j) \in E$

$$\phi_G = \bigwedge_{(i,j)\in E} \phi_{i,j}$$

chiaramente la funzione di riduzione è polinomiale, inoltre, se $G \in 4 - COL$, esiste una 4-colorazione $C = (c_1, c_2 \dots, c_n)$ che ha codifica binaria $x = (x_1, x'_1, \dots x_n, x'_n)$ che soddisfa ϕ_G, ossia $\phi_G(x) = 1$. ■

Formalizziamo il concetto di riduzione polinomiale

Definizione : Siano A, B due linguaggi diremo che A *è riducibile in tempo polinomiale* a B, e denoteremo

$$A \leq_m^P B$$

se esiste una TM $R : \Sigma^* \to \Sigma^*$ tale che

$$\forall x \in \Sigma^*, \quad x \in A \iff R(x) \in B$$

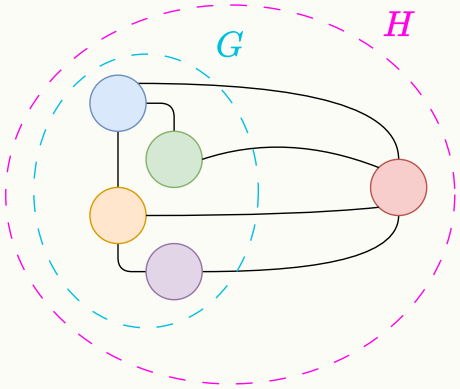
e R ha complessità di tempo polinomiale.

Teorema : se $A \leq_m^P B$ e $B \in P$ allora $A \in P$.

Dimostrazione : data una TM M_B che decide B in tempo polinomiale, è vero che $M_B(R(x))$ decide ogni $x \in A$, in tempo polinomiale in $|x|$, dove R è la TM della riduzione, per definizione anch'essa polinomiale. ■

Teorema : $3 - COL \leq_m^P 4 - COL$

Dimostrazione : si definisce la TM della riduzione R tale che $R(< G >) =< H >$, dove G è un grafo 3-colorabile ed H è un grafo 4-colorabile. H si ottiene aggiungendo a G un nodo connesso a tutti gli altri nodi, con un quarto colore nuovo non presente in G.



Se G è 3-colorabile, H sarà 4-colorabile, e se H è 4-colorabile, G deve essere 3-colorabile. ■

Teorema : $A \leq_m^P B \wedge B \leq_m^P C \implies A \leq_m^P C$

Dimostrazione : sia R_1 la riduzione da A a B

$$x \in A \iff R_1(x) \in B$$

sia R_2 la riduzione da B a C

$$x \in B \iff R_2(x) \in C$$

allora, per ogni x

$$x \in A \implies R_1(x) \in B \implies R_2(R_1(x)) \in C$$

ma allora $R_1 \circ R_2$ è una riduzione da A a C perché

$$x \in A \iff R_1(R_2(x)) \in C$$

■

Non tutte le riduzioni hanno complessità polinomiale, si consideri il seguente linguaggio

$$4 - CROMA = \{< G > \mid \mathcal{X}(G) = 4\}$$

$\mathcal{X}(G)$ è il *numero cromatico* di G, $\mathcal{X}(G) = k$ se G è $k-$colorabile, ma per ogni $k' < k$ non è k'-colorabile. È vero che

$$4 - CROMA \leq_T^P SAT$$

Dove $\leq_T^P$ è una riduzione ma non di tipo map-reduction, la T sta per turing, la riduzione avviene mediante una procedura/algoritmo.

Dato G, si considera una TM (descritta da un algoritmo) la cui procedura è definita come segue

1. si considera una riduzione R che mappa $4 - COL$ in SAT
2. si stabilisce se G è o non è 4-colorabile tramite la risoluzione della formula logica associata.
3. si considera una riduzione R' che mappa $3 - COL$ in SAT
4. si stabilisce se G è o non è 3-colorabile tramite la risoluzione della formula logica associata

La procedura mostra che, se è possibile decidere SAT, è possibile decidere $4-CROMA$, ma tale riduzione non avviene mediante una funzione.

Teorema : se $A \leq_m^P B$ e $B \in NP$ allora $A \in NP$.

Dimostrazione : la riduzione R è una TM (di tempo polinomiale) tale che

$$x \in A \iff R(x) \in B$$

sia N_B la NTM che decide (in tempo polinomiale) B, si definisce $N_A(x) = N_B(R(x))$, è chiaro che N_A è una NTM che decide A in tempo polinomiale. ■

La seguente definizione è fondamentale.

Definizione (NP completezza) : Un linguaggio A si dice NP *difficile* se, per ogni singolo linguaggio $L \in NP$, è vero che $L \leq_m^P A$. Se A è sia NP difficile che in NP, allora si dice che A è **NP-completo**.

Se si dimostra che un linguaggio NP completo è in P, allora ogni singolo linguaggio in NP sarebbe in P. Ad oggi non è mai stato dimostrato, infatti:

non è noto se $P = NP$ sia vero o falso.

il seguente risultato è uno dei passi più importanti nella teoria della complessità.

Teorema (Cook-Levin) : SAT è NP completo.

Dimostrazione : si è già dimostrato che $SAT \in NP$, bisogna dimostrare solo che SAT sia NP difficile, ovvero che ogni linguaggio NP si riduce polinomialmente a SAT.

Sia A un generico linguaggio in NP, e sia N la NTM che lo decide in tempo polinomiale, in particolare, in $O(n^k)$.

$$L(N) = A$$

Per definizione, per ogni input $x \in A$, $N(x)$ ha almeno un ramo accettante.

Per la dimostrazione si introduce la definizione di **tabella di computazione**, per ogni ramo di computazione di N su input $w = w_1 w_2 \dots w_n$, è associata una tabella di computazione, della forma

#	q_0	w_1	w_2	w_3	$\cdots$	w_n	$\sqcup$	$\sqcup$	$\cdots$	$\sqcup$	#
#	$\cdots$				$\vdots$	$\vdots$	$\vdots$			$\cdots$	#
#	$\cdots$			$\cdots$	a	q_1	b	$\cdots$		$\cdots$	#
#	$\cdots$			$\cdots$	q_2	a	c	$\cdots$		$\cdots$	#
$\vdots$					$\vdots$	$\vdots$	$\vdots$				
#	$\cdots$									$\cdots$	#

- sia n la dimensione dell'input w
- ogni riga della tabella inizia e finisce con il simbolo #, o un qualsiasi altro arbitrario simbolo che non sia nell'alfabeto della NTM.
- ogni elemento (i, j) della tabella è detto *cella*
- ogni riga ha n^k+3 (ma scriveremo per semplicità n^k) celle, gli spazi che non sono contenuti dall'input o dallo stato hanno il carattere $\sqcup$.
- una computazione può essere lunga al più n^k passi. La tabella avrà ordine $n^k \times n^k$
- ogni cella può contenere un solo simbolo, questo può essere uno stato, un carattere dell'alfabeto del nastro, il simbolo # oppure $\sqcup$.
- ogni riga contiene una configurazione della computazione della TM.

Una *finestra* nella tabella, è una sotto tabella di 2 righe e 3 colonne, rappresentata in verde nell'immagine.

Le righe della tabella rappresentano una successione di configurazioni nella computazione (andando dall'alto verso il basso), una tabella è valida se la successione è coerente con l'evoluzione della computazione descritta dalla funzione di transizione δ della NTM.

Una tabella è *accettante* se contiene in una riga lo stato q_{acc} ed è valida. Una NTM accetta un input w se esiste almeno una tabella accettante su w.

Definiremo una riduzione che data N produrrà una formula booleana ϕ che simula l'esecuzione di un input w su N. Siano

- Q l'insieme degli stati di N
- Γ l'alfabeto del nastro di N

ogni variabile della formula ϕ sarà del tipo

$$x_{i,j,s}$$

dove

$$i, j \in [1, 2 \dots n^k] \qquad s \in \mathbb{S} = Q \cup \Gamma \cup \{\#\}$$

esiste quindi una variabile per ogni cella (i, j) della tabella, ed ogni possibile valore che può assumere. Vogliamo definire ϕ in modo tale che

$$\phi(x) = 1 \iff \text{la tabella è accettante}$$

Osservazione : Il numero di variabili è $n^{2k} \cdot |\mathbb{S}|$, è quindi polinomiale in n, dato che $|\mathbb{S}|$ non dipende da n.

Le variabili saranno definite in tal modo

$$x_{i,j,s} = \begin{cases} 1 \text{ se nella cella } (i,j) \text{ c'è il valore } s \\ 0 \text{ altrimenti} \end{cases}$$

Si progetta la formula ϕ in modo tale che, un assegnamento che la soddisfa, corrisponde ad una tabella accettante per N. Definiamo diverse sottoformule di *phi*.

prima formula : ϕ_{start}
questa formula serve a descrivere il fatto che la prima regola di una tabella valida deve necessariamente contenere la configurazione iniziale, che sappiamo essere

$$\#\ q_0\ w_1\ w_2\ w_3\ \dots w_n\ \sqcup\ \dots\ \sqcup\ \#$$

la formula sarà quindi

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \dots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \dots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$$

è chiaro che questa formula deve essere necessariamente vera per permettere che una tabella sia valida, dato che lo stato iniziale è sempre determinato in tal modo per ogni ramo di computazione.

seconda formula : ϕ_{acc}
questa formula descrive il fatto che, una tabella per essere accettante deve contenere *almeno* una cella con lo stato accettante, è quindi (intuitivamente) definita come segue

$$\phi_{acc} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{acc}}$$

terza formula : ϕ_{cell}
questa formula deve descrivere che, per ogni variabile $x_{i,j,s} = 1$ (la cella (i, j) contiene s), deve essere vero che $x_{i,j,t} = 0$ per ogni $t \neq s$, ossia, una cella può avere un solo valore. Inoltre deve essere vero che una cella ha esattamente un valore, quindi deve esistere necessariamente s tale che $x_{i,j,s} = 1$ per ogni cella (i, j).

$$\bigvee_{s \in \mathbb{S}} x_{i,j,s} \qquad \text{almeno un valore}$$

$$\bigwedge_{\substack{s,t \in \mathbb{S} \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \qquad \text{al massimo un valore}$$

la terza formula completa è

$$\phi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \Big(\Big(\bigvee_{s \in \mathbb{S}} x_{i,j,s} \Big) \wedge \Big(\bigwedge_{\substack{s,t \in \mathbb{S} \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \Big) \Big)$$

quarta formula : ϕ_{move}
questa formula assicura che la riga $l+1$ è coerente con la riga l, ossia che le due configurazioni che si susseguono sono coerenti e in accordo con la funzione di transizione δ della NTM N. Per fare ciò, basta controllare ogni possibile finestra su due righe, questa deve essere *lecita* (non violare le regole di δ).

- in una finestra, tutti i simboli che non sono adiacenti ad uno stato q devono rimanere invariati
- i simboli adiacenti ad uno stato q devono evolvere nella riga successiva seguento uno dei possibili risultati della δ

$\delta(q_1, b) = (q_2, c, L)$

a	q_1	b
q_2	a	c

lecita

a	b	b
a	b	c

non lecita

nella terza colonna il valore del nastro è cambiato da una configurazione all'altra, ma ciò è impossibile in quanto la testina (data dalla posizione dello stato) non era adiacente a quella posizione

denotiamo la finestra $[i,j]$ quella composta dalle celle

$$\begin{matrix} (i,j) & (i+1,j) & (i+2,j) \\ (i,j+1) & (i+1,j+1) & (i+2,j+1) \end{matrix}$$

$$\phi_{move} = \bigwedge 1 \leq i,j \leq n^k (\text{ la finestra } [i,j] \text{ è lecita})$$

Senza entrare troppo nel formalismo, una finestra $[i,j]$ è lecita se la seguente è vera

$$\bigvee_{\substack{a_1 \ldots, a_6 \\ \text{rendono la finestra} \\ \text{lecita}}} (x_{i,j,a_1} \wedge x_{i+1,j,a_2} \wedge \cdots \wedge x_{i+2,j+1,a_6})$$

La formula ϕ tale che, un assegnamento che la soddisfa, corrisponde ad una tabella accettante per N, è la seguente

$$\phi = \phi_{start} \wedge \phi_{acc} \wedge \phi_{cell} \wedge \phi_{move}$$

la riduzione ha costo

$$n^k n^k |\mathbb{S}| = poly(n)$$

quindi un qualsiasi linguaggio A si riduce polinomialmente a SAT. ■

Utilizzando la riduzione è possibile dimostrare che molti linguaggi incontrati sono NP completi, ad esempio, essendo che

$$SAT \leq_m^P CIRCUIT - SAT$$

ne consegue che anche $CIRCUIT - SAT$ è NP completo. Ed essendo che

$$CIRCUIT - SAT \leq_m^P 3 - SAT$$

anche $3 - SAT$ è NP completo. Essendo vere le seguenti relazioni

$$3 - COL \leq_m^P 4 - COL \leq_m^P SAT \leq_m^P CIRCUIT - SAT \leq_m^P 3 - SAT \leq_m^P CLIQUE$$

tutti i linguaggi sopra elencati sono NP completi.

Una conseguenza della NP completezza è il seguente risultato.

Teorema : Se S è un linguaggio NP completo, allora $S \in P \iff P = NP$

Dimostrazione : $\forall L \in NP$, $L \leq_m^P S$. Se $S \in P$, allora L, riducendosi ad S in tempo polinomiale, risulta anch'esso in P. D'altra parte, se $P = NP$ ed S è NP completo, allora $S \in P$. ■

Si è mostrato che la decidibilità di SAT è un problema difficile, e ad oggi non sembra essere decidibile in tempo polinomiale, si vuole ora analizzare la sua risolvibilità, ossia la ricerca di una soluzione per una formula ϕ.

SAT è **self-riducibile**, si assuma che $P = NP$, allora per definizione una TM deterministica M può decidere SAT in tempo polinomiale.

> $M(\phi)$ accetta se ϕ è soddisfacibile $M(\phi)$ rifiuta se ϕ non è soddisfacibile

Si può trovare un assegnamento per ϕ considerando la seguente procedura

1. data $\phi(x_1, x_2 \dots x_n)$
2. si considera $\phi' = \phi(0, x_2 \dots x_n)$
3. si controlla $M(\phi')$, questa per ipotesi avrà tempo polinomiale
4. se $M(\phi')$ accetta si pone nella soluzione $x_1 = 0$, altrimenti si pone $x_1 = 1$
5. si controllano nello stesso modo tutti gli altri letterali $x_2, x_3 \dots, x_n$
6. al termine si avrà un assegnamento valido per ϕ, trovato in tempo polinomiale.

Essendo che ogni problema NP si riduce a SAT, si ha che se $P = NP$, una soluzione valida per ogni problema NP può essere trovata in tempo polinomiale.

Teorema : $P = NP \implies EXP = NEXP$

Dimostrazione : Si utilizza una tecnica chiamata *padding*, l'assunzione è che $P = NP$, si vuole dimostrare che $NEXP \subseteq EXP$ (sappiamo già che $EXP \subseteq NEXP$), sia $L \in NEXP$ e sia N la NTM che decide L in tempo $O(2^{n^k})$ per qualche k.

Si considera un linguaggio derivato

$$L' = \{< x, 1^{2^{|x|^k}} > \mid x \in L\}$$

Nota bene : $1^{2^{|x|^k}}$ è la stringa composta da $2^{|x|^k}$ caratteri (che sono appunto, 1). Il carattere 1 non deve essere presente nell'alfabeto della NTM.

> **Fatto** : $L' \in NP$, la dimostrazione è semplice, si consideri una NTM N', la cui procedura è descritta come segue
>
> 1. controlla che l'input è della forma $x' =< x, 1^{2^{|x|^k}} >$. Tale controllo avviene in tempo $O(|x'|)$
> 2. se il controllo ha successo, esegue $N(x)$
>
> chiaramente N' decide L' in tempo polinomiale in $|x'|$.

Visto che per ipotesi $P = NP$, si ha che $L' \in P$, allora esiste una TM deterministica M' che decide L' in tempo polinomiale. Allora è possibile decidere L in tempo esponenziale

- dato x, si crea la stringa $x' =< x, 1^{2^{|x|^k}} >$, la creazione di tale stringa avviene in tempo $O(2^{|x|^k})$
- avvio $L'(x')$, questa opera in tempo polinomiale, se L' accetta, allora $x \in L$, altrimenti rifiuta

è quindi chiaro che L viene deciso da tale procedura in tempo esponenziale $\implies L \in EXP$. ■

Teorema (di Ladner) : : Se $P \neq NP$, allora esiste un linguaggio $L \in NP$ tale che N non è NP completo. Se $P = NP$, ogni linguaggio in NP è NP completo.

3.5 La Classe *coNP*

Definiamo ora una nuova classe di complessità. La definizione di NP riguarda la decidibilità, ossia la certificazione che un certo elemento sia in un linguaggio, la classe $coNP$ riguarda la certificazione che un elemento *non sia* in un linguaggio.

Ad e sempio, esiste il linguaggio

$$UNSAT = \overline{SAT}$$

e ci si chiede: sapendo che $SAT \in P$, $UNSAT \in NP$?

Definizione ($coNP$) : $coNP = \{L \mid \overline{L} \in NP\}$

Osservazione : $coNP \neq \overline{NP}$

Teorema : $SAT \in P \iff UNSAT \in P$

Dimostrazione : dato un decisore per SAT, si decide $UNSAT$ negando la risposta di tale decisore. ■

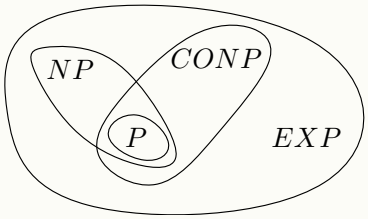
Teorema : $P = coP$, ovvero, P è chiuso per il complemento.

Dimostrazione : Dato un decisore per un linguaggio in P, è necessario invertire il risultato della TM. ■

Teorema : $coNP \subseteq EXP$

Dimostrazione : $L \in coNP \implies \overline{L} \in NP \subseteq EXP \implies L \in COEXP$ ma $COEXP = EXP$. ■

Se $P \neq NP$ è valida la seguente relazione fra insiemi



Teorema : $P \subseteq coNP$

Dimostrazione : $L \in P \implies \overline{L} \in P \subseteq NP \implies \overline{L} \in NP \implies L \in coNP$. ■

Teorema : $P = NP \implies P = coNP = NP$

Dimostrazione : $L \in coNP \implies \overline{L} \in NP = P \implies \overline{L} \in P$ ma essendo che $P = coP$ si ha $L \in P$. ■

Corollario : $coNP \neq NP \implies P \neq NP$.

Definizione : L è $coNP$ completo se

- $L \in coNP$
- $\forall A \in coNP, \quad A \leq_m^P L$

Teorema : $UNSAT$ è $coNP$ completo.

Dimostrazione : Ssappiamo che $UNSAT \in coNP$, è vero che

$$A \leq_m^P UNSAT \iff \overline{A} \leq_m^P \overline{UNSAT} = SAT$$

siccome $\overline{A}$ è in NP e SAT è NP completo, è vero che $\overline{A}$ si riduce a SAT, quindi $UNSAT$ è $coNP$ completo. ■

In passato, diversi problemi che sono in $NP \cap coNP$, sono stati dimostrati essere anche in P, si consideri l'insieme

$$PRIMES = \{< x > \mid x \text{ è un numero primo}\}$$

nel 1975 è stato dimostrato che $PRIMES \in NP$, e successivamente, nel 2001, è stato dimostrato che $PRIMES \in P$. Altri problemi sono in $NP \cap coNP$, ma non è stato ancora verificato se questi siano o non siano in P.

CAPITOLO

4

COMPLESSITÀ SPAZIALE

Data una TM M, si definisce *complessità spaziale* la funzione $S : \mathbb{N} \to \mathbb{N}$ tale che

$$S(n) = \max_{x \text{ t.c. } |x|=n} \Big(\text{numero di celle distinte utilizzate da } M \text{ su input } x\Big)$$

C'è una differenza fondamentale fra complessità spaziale e temporale, lo spazio può essere riutilizzato, il tempo no.

La complessità spaziale non considera le dimensioni dell'input, in quanto non permetterebbe a nessuna TM di avere una complessità spaziale inferiore a $O(n)$, dato che l'input occupa delle celle di memoria. Si considerano sempre quindi 2 nastri, uno riservato all'input, ed uno riservato al calcolo. Le celle del nastro di input non saranno considerate nella misura di complessità spaziale.

Definizione : $SPACE(S(n)) = \{L \mid \exists\ TM\ M \text{ tale che } L(M) = L \wedge M \text{ ha complessità } O(S(n))\}$
Da ciò, ne derivano alcune classi importanti

$$PSPACE = \bigcup_{k \in \mathbb{Z}^+} SPACE(n^k)$$

$$EXPSPACE = \bigcup_{k \in \mathbb{Z}^+} SPACE(2^{n^k})$$

$$LOG = \bigcup_{k \in \mathbb{Z}^+} SPACE(\log(n))$$

Vediamo alcuni esempi

$$A = \{0^n 1^n \mid n \in \mathbb{N}\}$$

Si vuole mostrare che $A \in LOG$, ossia che esiste un algoritmo che, se la dimensione dell'input è n, utilizzera un numero in $O(\log(n))$ di celle distinte per completare la computazione e fornire un risultato.

In precedenza, è stata già presentata una TM che decidesse questo linguaggio, la seguente procedura è differente, ed è descritta dai seguenti punti

- Sul nastro di lavoro, viene memorizzato un contatore per memorizzare il numero di 0 ed 1 dell'input
- il numero di 0 ed 1 è al più $2n$, per rappresentare tale numero in binario servono $O(\log_2(n))$ celle
- la TM, inizierà a contare il numero di 0, incrementando il contatore

- quando troverà il primo 1, inizierà a contare gli 1 decrementando il contatore
- se trova uno 0 che segue un 1, rifiuta
- al termine, se il contatore è uguale a 0 (il numero di 1 è identico al numero di 0) accetta

è chiaro che tale TM decide A in complessità di spazio $O(\log(n))$.

Osservazione : In termini di complessità di spazio, la TM multinastro è equivalente alla TM classica.

Si consideri ora il linguaggio PAL contenente tutte le stringhe palindrome, ad alto livello, la TM che lo decide dovrebbe

1. su input x, considerare $n = |x|$
2. per ogni $i \in [1, 2 \ldots, n]$
   (a) rifiuta se $x_i \neq x_{n+1-i}$
3. accetta

La TM userà un numero costante di nastri, ad alto livello la procedura è descritta dai seguenti punti

- sul nastro 1, si realizza l'indice i, che occupa $O(\log_2(n))$
- si memorizza l'indicie sul nastro 2 e si incrementa ad ogni passo, tale operazione richiede sempre complessità spaziale logaritmica
- sul nastro 3 ad ogni passo si calcola $n + 1 - i$
- ad ogni iterazione si controlla che $x_i \neq x_{n+1-i}$

è chiaro che tale TM decide A in complessità di spazio $O(\log(n))$.

Gli esempi visti hanno in comune lo stesso modello per decidere i linguaggi in complessità spaziale logaritmica, in particolare, su input x, si può valutare la lunghezza dell'input $x = |n|$ in spazio logaritmico.

In tale spazio, è anche possibile gestire un contatore/indice con il quale eseguire un numero di iterazioni che è polinomiale rispetto ad n, si possono inoltre considerare simboli presi dai nastri ed eseguire su di essi semplici operazioni aritmetiche.

Esempio : è possibile eseguire il prodotto fra due numero $a \cdot b = c$ in complessità di spazio logaritmica seguendo tale procedura:

1. si inizializza $c = 0$
2. si inizializza un indice $i = 0$
3. per i che va da 0 a b:
   - si incrementa c di a

Non è efficiente in termini di tempo, ma lo è in termini di spazio. Per molti problemi, non sono state trovate delle TM che li decidessero in complessità di spazio logaritmica, non si sa quindi se siano o non siano in LOG, un seguente esempio è il linguaggio

$$PATH = \{< G, s, t > \mid s, t \in V(G) \wedge \text{ esiste un cammino } s \to t \text{ in } G\}$$

Per cui si è considerato un algoritmo nella sezione 3.1, questo, opera in tempo polinomiale, ma la complessità di spazio non è logaritmica, dato che per marcare i nodi, usa al più un bit per nodo, è quindi in $O(n)$ (si ricordi che nelle TM che operano sui grafi, le dimensioni dell'input identificano il numero dei nodi del grafo).

Esempio : $3SAT \in PSPACE$, questo è certo perché è possibile provare ogni singolo assegnamento della formula, essendo n il numero dei letterali, ad ogni assegnamento provato si usano al più n celle.

Essendo che $3SAT$ è NP completo, ne consegue che

$$NP \subseteq PSPACE$$

4.1 Relazione fra Spazio e Tempo

In questa sezione verrà resa chiara la relazione che intercorre fra spazio e tempo, in particolare, le relazioni di inclusione fra gli insiemi che definiscono le varie classi.

Teorema : $DTIME(f(n)) \subseteq SPACE(f(n))$

Dimostrazione : Supponiamo che una TM ha complessità temporale $f(n)$, per assurdo, ha complessità spaziale $g(n)$, con $g(n) = \Omega(f(n))$ (è asintoticamente maggiore), ad esempio, $f(n)$ potrebbe essere polinomiale e $g(n)$ esponenziale.

Ma allora, essendo che ogni cella per essere riempita richiede almeno 1 passo di computazione, la TM necessariamente esegue $g(n)$ passi di computazione, ma per ipotesi ha complessità temporale $f(n)$, allora non può avere complessità spaziale maggiore di quella temporale. ■

Il teorema ha un significato molto chiaro:

il tempo limita lo spazio

Vediamo ora in che modo lo spazio può limitare il tempo.

Teorema : Per ogni $f(n) \geq \log(n)$[1], si ha che $SPACE(f(n)) \subseteq DTIME(2^{O(f(n))})$

Dimostrazione : Nella valutazione della complessità spaziale, una TM ha un nastro di lavoro ed un nastro per l'input, una qualsiasi configurazione (snapshot durante la computazione) di una TM M, può essere rappresentato come segue

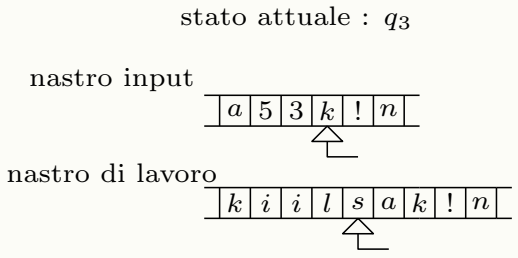
$$a_1\ a_2\ a_3\ q_i\ a_4\ a_5\ a_6\ ;\ k$$

dove

- a_j sono caratteri sul nastro di lavoro
- q_i è lo stato attuale, e la sua posizione indica la posizione della testina sul nastro di lavoro
- k è il numero della cella su cui è presenta la testina sul nastro dell'input

Essendo M deterministica, ed essendo un decisore, nessuna configurazione può ripetersi due volte durante una computazione. L'implicazione è chiaria :

> Il numero MASSIMO di passi di computazione che può fare M su un qualsiasi input, è limitato dal numero totale di configurazioni, in quanto al più, una computazione può includere nella sequenza ogni singola configurazione, ma queste non possono ripetersi



configurazione
$i\ l\ s\ q_3\ a\ k\ !\ ;\ 4$

Il numero totale di configurazioni è dato da

- ogni possibile contenuto del nastro di lavoro, ossia $|\Gamma|^{f(n)}$
- un possibile stato $|Q|$
- una posizione del nastro di input n

[1] con $\geq$ si intende, che è asintoticamente maggiore, ossia $f(n) \in \Omega(\log(n))$

$$\#\text{configurazioni} = |\Gamma|^{f(n)}| \cdot |Q| \cdot n$$

essendo per ipotesi $f(n) \geq \log(n)$ si ha che

$$|\Gamma|^{f(n)} \cdot |Q| \cdot n \leq |\Gamma|^{f(n)} \cdot |Q| \cdot 2^{f(n)}$$

quindi il numero di configurazioni è in $2^{O(f(n))}$, dato che $|\Gamma|$ e $|Q|$ sono costanti. Per le osservazioni precedenti, anche il numero di passi di computazione è limitato da $2^{O(f(n))}$. ■

Corollario : $PSPACE \subseteq EXP$, inoltre

$$LOG \subseteq P \subseteq PSPACE \subseteq EXP$$

ma $P \neq EXP$ quindi una delle inclusioni è stretta, in particolare, almeno una delle due seguenti affermazioni è vera

- $P \neq PSPACE$
- $PSPACE \neq EXP$

4.2 Non Determinismo

In termini di complessità spaziale, il non determinismo non ha lo stesso impatto che ha nella complessità temporale. Il seguente risultato è *cruciale* ed è alla base di molti altri risultati importanti nell'ambito della complessità spaziale.

Teorema : $PATH \in SPACE(\log^2(n))$

Dimostrazione : Sia M la TM che deve decidere $PATH$, l'input è la codifica del grafo $G = (V, E)$, e la coppia di nodi s, t per cui si vuole stabilire se esiste un cammino fra di essi. La TM può calcolare $n = |V|$ in spazio logaritmico.

Osservazione : Se esiste un cammino fra s e t, questo è composto da al più n nodi.

Verrà adoperata una procedura ricorsiva, che ricercherà un nodo u per cui

- esiste un cammino da s ad u
- esiste un cammino da u a t

se tale nodo u esiste, allora esiste un cammino da s a t. L'algoritmo opererà ricorsivamente "spezzando" in 2 il grafo ad ogni passo.

Si definisce la seguente procedure `FIND_PATH(x,y,k)` definita come segue

```
FIND_PATH(x,y,k){
    if(k==0){
        if(x=y OR (x,y)=arco in E){ ACCEPT }
    }
    per ogni nodo w{
        if(FIND_PATH(x,w,k-1) AND FIND_PATH(w,y,k-1)){
            ACCEPT
        }
    }
    REJECT
}
```

Al termine, si avrà che la procedura accetterà solo se esiste un cammino da x ad y con al più k nodi. Sarà necessario fra si che la TM M esegua la procedura

`FIND_PATH(s,t,`$\lceil \log(n) \rceil$`)`

Osservazioni

- ad ogni passo ricorsivo è necessario memorizzare un numero costante di variabili, ciò richiede spazio $O(\log(n))$
- la ricorsione si divide in 2 ad ogni passo, l'altezza di ogni ramo ricorsivo è quindi al più $O(\log(n))$

Ne consegue che lo spazio totale necessario sarà in $O(\log^2(n))$. ■

Occupiamoci adesso di definire le classi di spazio per le NTM (TM non deterministiche).

Definizione : $NSPACE(f(n)) = \{L \mid \exists \text{ NTM } N \text{ t.c. } L(N) = L \wedge N \text{ ha complessità di spazio } f(n)\}$

la complessità di spazio di una NTM si valuta sul ramo di computazione che ha occupato più celle distinte. Ne conseguno naturalmente le seguenti classi

$$NPSPACE = \bigcup_{k \in \mathbb{Z}^+} NSPACE(n^k)$$

$$NEXPSPACE = \bigcup_{k \in \mathbb{N}} NSPACE(2^{n^k})$$

$$NLOG = NSPACE(\log(n))$$

Vedremo che $PSPACE = NPSPACE$, ossia, che il non determinismo non ha alcuna influenza sulla complessità di spazio.

Lemma : $PATH \in NLOG$

Dimostrazione : Si considera una NTM N per $PATH$ che esegue la seguente procedura

- su input G, s, t accetta se $s = t$
- calcola deterministicamente in spazio logaritmico $n = |V|$
- definisce $curNode = s$
- per ogni i da 1 ad n
  - *non deterministicamente* (fork) considera ogni possibile nodo $u \in V$
  - se $(curNode, u) \in E$, allora $curNode = u$
  - se $curNode = t$, accetta
  - se $(curNode, u) \notin E$, allora rifiuta
- rifiuta

Su ogni ramo il nuemro di variabili utilizzate è costante, l'algritmo opera in complessità spaziale $O(\log(n))$ e accetta solo se esiste un cammino da s a t. ■

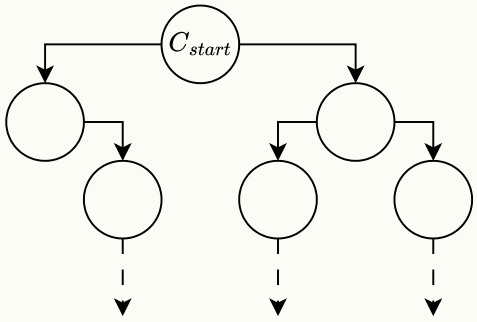
Vedremo che $PATH$ è $NLOG$ completo.

Teorema (di Savitch) : $NLOG \subseteq P \cap SPACE(\log^2(n))$

Dimostrazione : Sia $A \in NLOG$, ossia $\exists\ NTM\ N$ tale che $L(N) = A$ e N ha complessità spaziale $O(\log(n))$, si ricordi che una configurazione è della forma

$$C = afjej\ q_i\ sfj\ ;\ r$$

dove q_i è lo stato, che indica anche la posizione della testina sul nastro di lavoro, ed r è la posizione della testina sul nastro di input. Esistono $2^{O(\log(n))}$ possibili configurazioni, si associa, per ogni coppia N, x (x input di N) un grafo $G_{N,x}$, i cui nodi sono le possibili configurazioni C, ed esiste un arco (C, C') se C' è una configurazione che può seguire da C in accordo con le regole stabilite dalla funzione di transizione δ. Inoltre esisterà una configurazione iniziale C_{start} (naturalmente definita).



Definiamo configurazioni *accettanti*, quelle in cui lo stato corrente è lo stato accettante q_{acc}, è possibile (senza perdita di generalità) trasformare ogni possibile grafo $G_{N,x}$ in modo che abbia una sola configurazione accettante, ossia

$$C_{acc} = q_{acc}\sqcup \ ; \ 1$$

Basta assumere che $N(x)$, prima di accettare, si occupi di cancellare ogni elemento sul nastro di lavoro, e spostare le testine (di entrambi i nastri) nell'estrema sinistra. Tale trasformazione preserva la complessità di spaziale di N.

Naturalmente, $N(x)$ accetta se e solo se esiste un cammino da C_{start} a C_{acc} nel grafo $G_{N,x}$. Pertanto, $A \in NLOG \implies A \in P$, dato che esiste una TM M (deterministica) che decide in tempo polinomiale se $x \in A$ tramite la seguente procedura

- si scrive $G_{N,x}$ enumerando vertici ed archi, ciò richiede complessità temporale $poly(n)$.
- M utilizza l'algoritmo di marcatura (che ha complessità polinomiale) per verificare che esiste un cammino fra i due nodi del grafo.

$$NLOG \subseteq P$$

In maniera simile, è possibile utilizzare l'algoritmo ricorsivo visto ad inizio della sezione 4.2 per trovare un cammino fra i due nodi in complessità spaziale $O(\log^2(n))$.

Non è però possibile scrivere il grafo $G_{N,x}$ nel nastro di lavoro in quanto violerebbe lo spazio logaritmico, ma ciò non è necessario in quanto si può valutare di volta in volta se esiste un arco (C_i, C_j) valutando l'input x. ■

Il teorema di Savitch si può generalizzare:

Teorema : $NSPACE(f(n)) \subseteq DTIME(2^{O(f(n))}) \cap SPACE(f(n)^2)$

Corollario : $NPSPACE \subseteq EXP \cap PSPACE$, ma allora, sapendo già che $PSPACE \subseteq NPSPACE$ ne consegue che

$$NPSPACE = PSPACE$$

4.3 $NLOG$ completezza

Definizione ($NLOG$ completezza) : B è $NLOG$ completo se

- $B \in NLOG$
- se $A \in NLOG$, allora $A \leq_m^L B$

$\leq_m^L$ è un nuovo tipo di riduzione che deve garantire la seguente proprietà

$$A \leq_m^L B \implies A \in LOG \wedge B \in LOG$$

Definizione : si definisce la riducibilità **log space reduction** $\leq_m^L$ definita come segue:
$A \leq_m^L B$ se $\exists R : 0,1^* \to 0,1^*$ computabile in spazio $O(\log(n))$ tale che

$$x \in A \iff R(x) = B$$

Bisogna definire il termine "computabile in spazio $O(\log(n))$", il termine $R(x)$ potrebbe avere dimensione $poly(n)$, in tal caso scrivendo $R(x)$ sul nastro di lavoro, si violerebbe lo spazio logaritmico, si considera nel modello della TM un *nastro di output* che può essere scritto una sola volta, senza mai modificare le celle utilizzate, tale nastro verrà usato per scrivere l'output $R(x)$.

Teorema : $PATH$ è $NLOG$ completo.

Dimostrazione : Sappiamo già che $PATH$ è un linguaggio in $NLOG$, bisogna dimostrare che tutti i linguaggi in tale insieme si riducono a $PATH$. Sia A un qualsiasi linguaggio in $NLOG$, e sia N la NTM (di complessità logaritmica) tale che $L(A) = N$

- si definisce una TM R che, dato un input x per N, costruisce il grafo $G_{N,x}$ delle cofnigurazioni visto nella dimostrazione del teroema di Savitch.
- L'output di R (ossia $R(x)$) sarà $(G_{N,x}, C_{start}, C_{acc})$
- Ne consegue naturalmente che $x \in A \iff$ esiste un cammino $C_{start} \to C_{acc}$ in $G_{N,x} \iff R(x) = (G_{N,x}, C_{start}, C_{acc}) \in PATH$

In particolare, è possibile enumerare tutte le possibili configurazioni (che sono polinomiali in n) con un numero $O(\log(n))$ celle, e con la medesima complessità di spazio è possibile verificare se, date due configurazioni C, C', esiste o no l'arco (C, C'). ■

Si vuole mostrare che la definizione di *log space reduction* soddisfi le seguenti proprietà

- $A \leq_m^L B \implies \begin{cases} B \in LOG \implies A \in LOG \\ B \in NLOG \implies A \in NLOG \end{cases}$
- $A \leq_m^L B \wedge B \leq_m^L C \implies A \leq_m^L C$

Teorema : Se P e Q sono computabili in spazio $O(\log(n))$, allora lo è anche $R = P \circ Q$

Dimostrazione : Essendo che P e Q sono computabili in spazio $O(\log(n))$, e che *"lo spazio limita il tempo"*, si ha che P e Q hanno complessità di tempo polinomiale

- P ha complessità temporale $O(n^p)$ per qualche p
- Q ha complessità temporale $O(n^q)$ per qualche q

Si vuole definire una TM M capace di scrivere l'output $R(x) = Q(P(x))$ in complessità spaziale $O(\log(n))$.

Il problema, è che $y := P(x)$ potrebbe essere lungo $poly(n)$, quindi bisogna calcolare $R(x) = Q(P(x)) = Q(y)$ senza calcolare esplicitamente y.

la seguente parte la copio parola per parola da ciò che ha dettato il professore ma non l'ho capita

La TM M tiene traccia della posizione corrispondente a quello che sarebbe il nastro di input di Q, ciò

richiede spazio $O(\log(n))$, M deve determinare $y[i]$ (i-esimo carattere di y) e simulare un passo della computazione di Q, ciò può essere fatto in spazio $O(\log(n))$.

Si ricalcola $P(x)$ fino ad ottenere $y[i]$, scartando il resto, questo permette di simulare un passo di computazione di Q. fine parte ambigua ■

Corollario : $A \leq_m^L B \wedge B \in LOG \implies A \in LOG$

Fino ad ora abbiamo visto esempi di linguaggi

- NP completi
- $NLOG$ completi
- $coNP$ completi

Vediamo adesso alcuni esempi di linguaggi P completi e $PSPACE$ completi.

Definizione (P completezza) : C è P completo se $C \in P$ e $\forall A \in P$ si ha che $A \leq_m^L C$

Osservazione : La definizione riguarda la classe P (complessità temporale) ma utilizza la *log space reduction* (inerente alla complessità spaziale).

Problema aperto : La seguente proposizione non è stata dimostrata, e tutt'ora non si sa se sia vera o false

$$C \in LOG \implies P \subseteq LOG$$

Vediamo un esempio di problema P completo:

$$CIRCUIT - EVAL = \{< C, x > \mid C \text{ è un circuito booleano } \wedge C(x) = 1\}$$

Teorema : $CIRCUIT - EVAL$ è P completo.

Dimostrazione : TODO

Vediamo adesso un particolare linguaggio, conosciamo il linguaggio

$$SAT = \{< \phi > \mid \exists x,\ \phi(x) = 1\}$$

Ed il linguaggio delle tautologie

$$TAUT = \{< \phi > \mid \forall x,\ \phi(x) = 1\}$$

Definizione : Una formula booleana è **totalmente quantificata** se ogni variabile è coinvolta in un quantificatore (universale o esistenziale). Un esempio:

$$\exists z \forall x \exists y (x \to y) \vee z$$

Una formula di questo tipo è una *sentenza*, o è vera o falsa, non ci sono assegnamenti di variabili in quanto sono già considerate dai quantificatori. Si definisce il seguente linguaggio delle **Totally Quantified Boolean Formula**

$$TQBF = \{< \phi > \mid \phi \text{ è una TQBF}\}$$

Teorema : $TQBF \in PSPACE$

Dimostrazione : si considera il seguente algoritmo ricorsivo

`IsTrue`$(Q_1 x_1, Q_2 x_2 \ldots, Q_n x_n, \phi(x))$
dove $x = x_1, x_2 \ldots, x_n$

La procedura è descritta dai seguenti passi

- Se $n = 0 \implies \phi$ ha solo costanti, ritorna la valutazione

- Altrimenti, se $Q_1 = \exists$, ritorna

$$\texttt{IsTrue}(Q_2x_2, Q_3x_3 \ldots, Q_nx_n, \phi(0, x_2, x_3 \ldots, x_n)) \\ \vee \\ \texttt{IsTrue}(Q_2x_2, Q_3x_3 \ldots, Q_nx_n, \phi(1, x_2, x_3 \ldots, x_n))$$

- Altrimenti, se $Q_1 = \forall$, ritorna

$$\texttt{IsTrue}(Q_2x_2, Q_3x_3 \ldots, Q_nx_n, \phi(0, x_2, x_3 \ldots, x_n)) \\ \wedge \\ \texttt{IsTrue}(Q_2x_2, Q_3x_3 \ldots, Q_nx_n, \phi(1, x_2, x_3 \ldots, x_n))$$

L'algoritmo è corretto, la profondità di ricorsione è n e ad ogni chiamata ricorsiva lo spazio utilizzato è $O(n)$. Lo spazio totale utilizzato dall'algoritmo è $O(n^2) \implies TQBF \in PSPACE$. ■

Teorema : $TQBF$ è $PSPACE$-difficile. Ossia, ogni problema in $PSPACE$ si riduce in tempo plinomiale (quindi anche in spazio polinomiale) a $TQBF$.

Dimostrazione : Sia A un generico linguaggio in $PSPACE$, esiste una TM che decide A in spazio polinomiale, sia questo $O(n^a)$ per qualche $a \in \mathbb{N}$. Vogliamo definire una riduzione $R : \{0,1\}^* \to \{0,1\}^*$, che resituisca in output una Totally Quantified Boolean Formula ϕ_M tale che

$$x \in A \iff M(x) \text{ accetta} \iff R(x) = \phi_M \text{ è soddisfatta}$$

Osservazione : Una stessa TQBF può avere più formulazioni equivalenti, ad esempio, si può assumere che tutti i quntificatori siano all'inizioe della formula

$$\exists x_1 \forall x_2 [x_1 \to x_2] \wedge \ldots$$

Sappiamo che $M(x)$ opera in spazio polinomiale, ha $2^{O(n^a)}$ possibili configurazioni, una di queste sarà C_{start} definita

$$q_0 \sqcup \; ; \; 1$$

Come già visto, si può assumere (senza perdita di generalità) che esista un un'unica configurazione accettante C_{acc}.

Se dovessimo considerare il grafo $G_{M,x}$ considerato nella dimostrazione del teorema di Savitch, sappiamo che $M(x)$ accetta se e solo se esiste un cammino $C_{start} \longrightarrow C_{acc}$ in $G_{M,x}$.

La riduzione R dato x, dovrà calcolare una formula ϕ_M tale che, questa è soddisfacibile solo se esiste un cammino nel $C_{start} \longrightarrow C_{acc}$ in $G_{M,x}$. La formula ϕ_M deve avere dimensione polinomiale rispetto alle dimensioni dell'input n, e deve essere computabile in spazio polinomiale.

Le variabili di ϕ_M sono le possibili configurazioni C_i dell'esecuzione di M su x.

Idea 1) : Consideriamo una sotto formula ϕ_{yields} tale che $\phi_{yields}(C_i, C_j) = 1 \iff C_j$ segue la configurazione C_i in accordo con le regole stabilite dalla funzione di transazione δ. Potremmo definire ϕ_M come segue

$$\phi_M = \exists C_1 \exists C_2 \exists C_3 \ldots \exists C_l \\ \phi_{yields}(C_1, C_2) \wedge \phi_{yields}(C_2, C_3) \ldots \phi_{yields}(C_{l-1}, C_l)$$

Con $C_1 = C_{start}$ e $C_l = C_{acc}$. Seppure l'idea è logicamente corrette, il numero l di possibili configurazioni è in $2^{O(n^a)}$, è quindi esponenziale, allora la riduzione R in questione non è computabile in spazio polinomiale, e quindi non va bene.

Idea 2) : Un'altra idea è quella di utilizzare l'algoritmo ricorsivo visto nel teorema di Savitch, definiamo la procedura $\phi_k(C_0, C_1)$, che è vera se e solo se esiste un cammino da C_0 a C_1 di lunghezza 2^k passi. La riduzione $R(x)$ dovrà solamente restituire $\phi_{O(n^a)}(C_{start}, C_{acc})$.

- caso base ($k = 0$) : allora $\phi_0(C_0, C_1)$ è vera se e solo se $\phi_{yields}(C_0, C_1)$ è vera, oppure $C_0 = C_1$.
- caso induttivo : $\phi_k(C_0, C_1)$ è definita come:

– $\exists\ C_{mid}$ tale che

$$\phi_{k-1}(C_0, C_{mid}) \wedge \phi_{k-1}(C_{mid}, C_1)$$

Anche in questo caso la logica è corretta ma le dimensioni di ϕ_k sono esponenziali dato che raddoppiano ad ogni passo

$$|\phi_k| = O(n^a) + 2|\phi_{k-1}|$$

Idea Finale) : Si considera l'idea 2 ma scritta in maniera diversa,

$$\phi_k(C_0, C_1)$$

è definita come segue : $\exists C_{mid} \forall D \forall D'$ tale che

$$\begin{pmatrix} (D, D') = (C_0, C_{mid}) \\ \vee \\ (D, D') = (C_{mid}, C_1) \end{pmatrix} \implies \phi_{k-1}(D, D')$$

In questo caso le dimensioni di ϕ_k sono $O(n^a) + |\phi_{k-1}| = O(n^{2a})$, che è polinomiale. ■

Teorema (Immerman-Szelepcsényi) : : La classe $NLOG$ è chiusa rispetto al complemento

$$NLOG = coNLOG$$

Dimostrazione : Essendo che $PATH$ è $NLOG$ completo, è necessario dimostrare che $PATH$ è in $coNLOG$, ossia che $\overline{PATH}$ è in LOG.

Bisogna quindi trovare un certificato in spazio logaritmico che verifichi che, dato un grafo, (G, s, t), non esiste il cammino $s \longrightarrow t$.

Sia V il **verificatore**, questo ha 3 nastri

- Nastro di input, contiene $< G, s, t >$
- Nastro di lavoro, il cui numero di celle usate deve essere logaritmico
- Nastro *read once*, può essere letto una sola volta e conterrà il certificato per dimostrare che il cammino non esiste

Definiamo R_l l'insieme dei nodi raggiungibili da s in (al più) l passi, e denotiamo $r_l = |R_l|$ il numero di questi nodi. Chiaramente $R_0 = \{s\} \implies r_0 = 1$. È ovvio che $R_l \subseteq R_{l+1}$.

Il certificato contenuto nel nastro read once, sarà diviso in diverse parti (qui sotto organizzate in righe) del tipo

certificato r_0
certificato r_1
certificato r_2
$\vdots$
certificato r_n
certificato : non esiste il cammino $s \longrightarrow t$

per ora si omette la struttura interna dei certificati.

Il verificatore, per definire il certificato r_{l+1}, deve avere in possesso r_l e l, e non tutti i certificati precedenti. finire/riscrivere la dimostrazione

4.4 Teoremi di Gerarchia

I risultati presentati in questa sezione permettono di separare le classi di complessità utilizzando la tecnica della *diagonalizzazione*.

Il teorema di *gerarchia dei tempi* afferma che se $t_1(n)$ è asintoticamente, molto più piccola di $t_2(n)$, allora esiste un linguaggio $L \in Dtime(t_2(n))$ tale che $L \notin Dtime(t_1(n))$

Vediamo un **idea di dimostrazione**, si definisce una codifica

$$[.]_{TM} : \sigma^* \to TM$$

che associa ad ogni stringa una TM, se la stringa non corrisponde ad alcuna TM sensata, allora se ne attribuisce una di default. In tal modo

$$\forall TM \ M \ \exists x \text{ tale che } [x]_{TM} = M$$

Definisco una TM D in modo tale che termini sicuramente in $O(t_2(n))$ passi, il comportamento è descritto dalla seguente procedura

-