Automi Calcolabilità e Complessità

Concetti fondamentali per la preparazione dell'esame

Lorenzo Melotto

Dicembre 2022

Le seguenti dispense sono state realizzate prendendo in aula gli appunti del prof. Daniele Venturi ed in secondo momento in parte ampliati seguendo il libro *Sipser, Introduction to the Theory of Computation terza edizione*. La maggior parte delle immagini presenti nelle dispense sono state prese dal libro. Il materiale è utilizzabile solo per scopo didattico e personale e ne è assolutamente vietata la vendita.

Contents

1 Automi a stati finiti (DFA)

1.1 Definizione DFA

Un **DFA** è una quintupla

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

Dove:

- Q=insieme **finito** degli stati
- Σ=alfabeto di **input**
- δ=funzione di **transizione** $Q \times \Sigma \to Q$
- q_0=stato **iniziale**
- F=stati di **accettazione** ($F \subseteq Q$)

1.2 Definizione - Linguaggio di un DFA

Siano M un **DFA**, A l'insieme di tutte le stringhe che M accetta. A è il **linguaggio** di M e si scrive

$$L(M) = A$$

1.3 Funzione di transizione estesa

Definiamo la seguente funzione di transizione estesa con l'obiettivo di valutare δ su una stringa

$$\delta^* : Q \times \Sigma^* \to Q$$

definita nel seguente modo:

$$\begin{cases} \delta^*(q, \epsilon) = \delta(q, \epsilon) \\ \delta^*(q, ax) = \delta^*(\delta(q, a), x)) \end{cases}$$

1.4 Linguaggi accettati da un DFA

Una stringa $x \in \Sigma^*$ è accettata da un DFA $M = (Q, \Sigma, \delta, q_0, F)$ se

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

ovvero

$$L(M) = \{x \in \Sigma^* \mid \delta^*(q_0, x) \in F\}$$

Dove Σ^* è l'insieme di stringhe composte dai caratteri in Σ e δ^* è la funzione di transizione estesa.

1.5 Linguaggi regolari

I **linguaggi regolari** sono tutti i linguaggi riconosciuti da un DFA e vengono definiti nel seguente modo

$$\text{REG} = \{L \subseteq \Sigma^* \mid \exists \text{DFA } M : L(M) = L\}$$

1.6 Proprietà dei linguaggi regolari

In quanto i linguaggi sono **insiemi di stringhe**, si possono definire delle operazioni.
Sia Σ l'alfabeto e $L_1, L_2 \subseteq \Sigma^*$ due linguaggi:

- **Unione**: $L_1 \cup L_2 = \{x \in \Sigma^* : x \in L_1 \vee x \in L_2\}$
- **Intersezione**: $L_1 \cap L_2 = \{x \in \Sigma^* : x \in L_1 \wedge x \in L_2\}$
- **Complemento**: $\neg L_1 = \{x \in \Sigma^* : x \notin L_1\}$
- **Concatenazione di stringhe**: Siano $x = a_1, ..., a_n,\ b = b_1, ..., b_m$ con $x, y \in \Sigma^*$ e $n, m > 0$, si ha che

  $$xy = a_1...a_n b_1...b_m$$

- **Concatenazione di linguaggi**: $L_1 \circ L_2 = \{xy : x \in L_1 \wedge y \in L_2\}$
- **Potenza**: è un caso speciale della concatenazione.
  Sia $x \in \Sigma^*$ e $n > 0$

  $$x^n = \underbrace{xxxx...x}_{n \text{ volte}}$$

  Definita nel seguente modo:

  $$\begin{cases} x^0 = \epsilon \\ x^{n+1} = x^n x \quad n \geq 0 \end{cases}$$

  E analogamente per i linguaggi:
  Sia $L \subseteq \Sigma^*$

  $$\begin{cases} L^0 = \{\epsilon\} \\ L^{n+1} = L^n \circ L \quad n \geq 0 \end{cases}$$

- **Operazione stella**: sia $L \subseteq \Sigma^*$

  $$L^* = \{x_1 x_2 ... x_k : k \geq 0, x_i \in L\}$$

  Alternativamente:

  $$\bigcup_{n \geq 0} L^n = \{\epsilon\} \cup L_1 \cup L_2 \cup \dots$$

2 NFA

Introdotti per dimostrare la chiusura di REG rispetto a $\circ$ e $*$.

2.1 Definizione NFA

Dato Σ sia $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$, dove ϵ indica la **parola vuota**.
Un automa **non deterministico** a stati finiti è una quintupla

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

Dove:

- Q, Σ, δ, q_0 sono definiti come per DFA
- $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$

Ora la δ manda da uno stato ad un insieme di stati.

2.2 Differenze tra NFA e DFA

Le differenze principali sono:

- **DFA**: ogni passo di computazione segue univocamente dal precedente
- **NFA**: diverse scelte per lo stesso passo di computazione

2.3 Computazione di un NFA

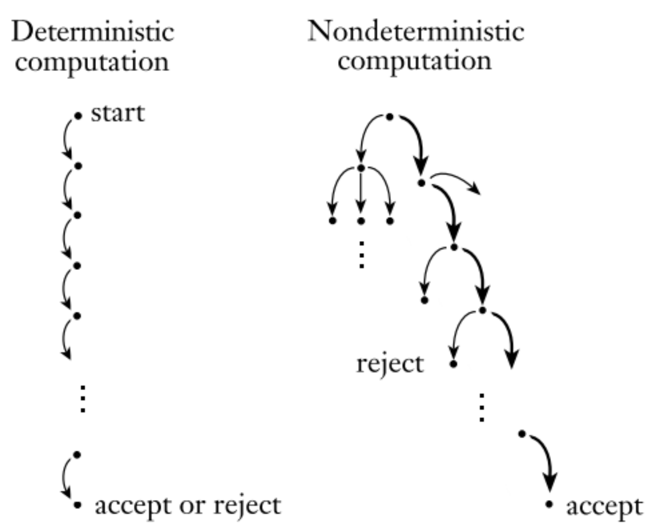


Figure 1: Computazione di un NFA

Un NFA accetta se esiste **almeno un ramo di computazione accettante**.

2.4 Esempio di computazione di NFA

Si consideri il seguente NFA N_1:

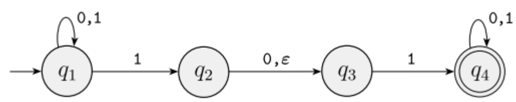


Figure 2: NFA N_1

Ecco una possibile esecuzione di N_1 con input 010110:

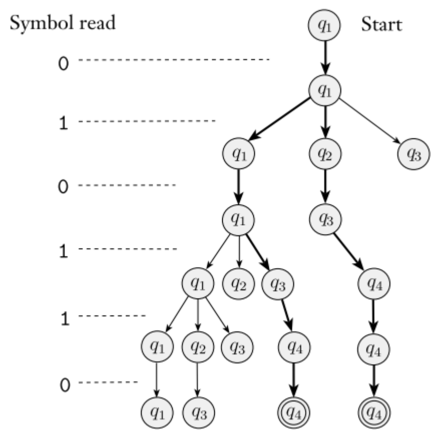


Figure 3: Albero di computazione di N_1 con input 010110

Da cui si evince che la stringa viene accettata in quanto almeno un ramo di computazione accetta, in questo caso, i rami che accettano sono due.

2.5 Configurazione per NFA

Una configurazione è una coppia $(q, x) \in Q \times \Sigma_\epsilon$

$$(p, ax) \vdash (q, x) \Leftrightarrow q \in \delta(p, a) \quad a \in \Sigma_\epsilon, x \in \Sigma_\epsilon^*$$

Un NFA $N = (Q, \Sigma, \delta, q_0, F)$ accetta $w \in \Sigma_\epsilon^*$ se e solo se w può essere scritto come $w = y_1 y_2 ... y_m,\ y_i \in \Sigma_\epsilon$ ed esiste una sequenza di stati $r_0, ..., r_k,\ r_i \in Q$ tale che

$$\begin{aligned} &r_0 = q_0 \\ &r_{i+1} \in \delta(r_i, y_{i+1}) \\ &r_m \in F \end{aligned}$$

2.6 Equivalenza tra NFA e DFA

Gli automi a stati finiti deterministici e non deterministici riconoscono la stessa classe di linguaggi, ovvero

$$L(\text{NFA}) = L(\text{DFA})$$

Dimostrazione: diciamo che due automi sono **equivalenti** se riconoscono lo stesso linguaggio. Dobbiamo dimostrare la doppia inclusione, ovvero

$$L(\text{NFA}) \subseteq L(\text{DFA}) \wedge L(\text{DFA}) \subseteq L(\text{NFA})$$

Chiaramente se $L \in L(\text{DFA})$, $\exists$DFA M che riconosce L, ma M è un **caso speciale di NFA** e quindi $L \in L(\text{NFA})$.
Resta da dimostrare l'altro lato dell'inclusione. Supponiamo per ipotesi che $\exists$NFA $N = (Q_N, \Sigma, \delta_N, q_0^N, F_N)$ che riconosce L. Dobbiamo costruire un DFA $D = (Q_D, \Sigma, \delta_D, q_0^D, F_D)$ che riconosce L.
L'idea è quella di creare uno stato in Q_D per ogni possibile insieme di stati in Q_N. Dopodiché viene **simulato deterministicamente** ogni livello dell'albero di computazione di N.
Costruzione di D: Per semplicità, iniziamo dal caso senza ϵ-archi:

1. $Q_D = \mathcal{P}(Q_N)$, quindi $|Q_D| = 2^{|Q_N|}$
2. Sia $R \in Q_D,\ a \in \Sigma$ allora
$$\delta_D(R, a) = \bigcup_{r \in R} \delta_N(r, a) = \{q \in Q_N : q \in \delta_N(r, a) \text{ per qualche } r \in R\}$$
3. $q_0^D = \{q_0^N\}$
4. $F_D = \{R \in Q_D : R \cap F_N \neq \emptyset\}$. In altre parole, D accetta nello stato R se almeno uno stato di Q_N in R è finale.

Presenza di ϵ-archi: per ogni stato R di D viene definito con $E(R)$ la collezione di stati che possono essere raggiunti da elementi di R attraverso zero o più ϵ-archi. Rispetto al caso senza ϵ-archi si aggiungono le seguenti accortezze:

1. $q_0^D = E(\{q_0^N\})$
2. δ_D diventa:
$$\delta_D(R, a) = \bigcup_{r \in R} E(\delta_N(r, a))$$

Ad ogni passo di computazione di D su un input, D entra in uno stato che corrisponde al sottoinsieme di stati in cui N potrebbe essere. $\square$

2.6.1 Esempio di equivalenza

Si consideri il seguente NFA $N_4 = (\{1, 2, 3\}, \{a, b\}, \delta, 1, \{1\})$

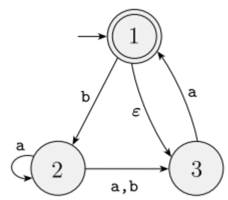


Figure 4: NFA N_4

Il suo DFA D **equivalente** è:

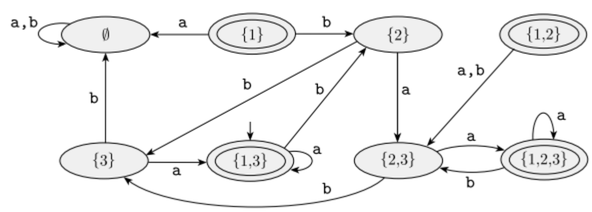


Figure 5: DFA D equivalente a NFA N_4

<u>Nota</u>: L'automa equivalente può essere ulteriormente semplificato in quanto non ci sono archi entranti per gli stati $\{1\}$ e $\{1, 2\}$, quindi possono essere eliminati senza alterare il comportamento dell'automa.

2.7 Dimostrazioni di chiusura rispetto alle operazioni sui linguaggi regolari

2.7.1 REG è chiusa rispetto a $\cup$

<u>Idea</u>: si hanno due linguaggi regolari A_1 e A_2 e si vuole dimostrare che $A_1 \cup A_2$ è regolare. L'idea è quella di prendere due NFA N_1 e N_2 rispettivamente per il linguaggi A_1 e A_2 e combinarli insieme per formare un nuovo NFA N, che accetta se almeno uno tra N_1 e N_2 accetta. Siano

$$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$
$$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$

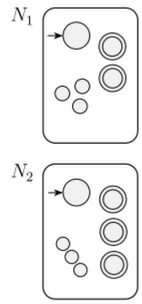


Figure 6: NFA N_1 e N_2

Con il **non determinismo** si può fare la seguente cosa:

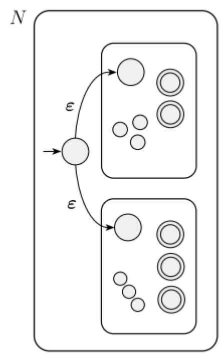


Figure 7: NFA N che rappresenta l'unione tra N_1 e N_2

N viene definito nel seguente modo: $N = (Q, \Sigma, \delta, q_0, F)$ dove

- $Q = \{q_0\} \cup Q_1 \cup Q_2$
- q_0 è il nuovo stato iniziale di N
- $F = F_1 \cup F_2$
- Per $q \in Q,\ a \in \Sigma_\epsilon$

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

2.7.2 REG è chiusa rispetto a $\circ$

Idea: si hanno due linguaggi regolari A_1 e A_2 e si vuole dimostrare che $A_1 \circ A_2$ è regolare. L'idea è quella di prendere due NFA N_1 e N_2 rispettivamente per il linguaggi A_1 e A_2 e combinarli insieme per formare un nuovo NFA N in modo tale che ogni volta che N_1 si trova in uno stato accettante, degli ϵ-archi partono da questi stati finali collegandosi allo stato iniziale di N_2, stando a significare che è stato trovato un pezzo iniziale di stringa che si trova nel linguaggio A_1. Siano

$$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$
$$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$

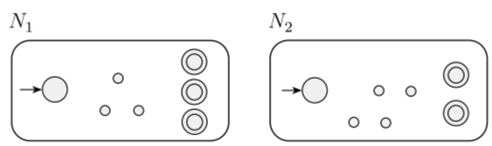


Figure 8: NFA N_1 e N_2

I due automi vengono collegati nel seguente modo usando il **non determinismo**:

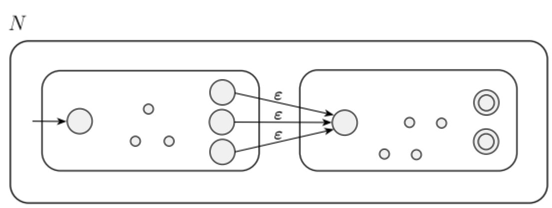


Figure 9: NFA N che rappresenta la concatenazione tra N_1 e N_2

N viene definito nel seguente modo: $N = (Q, \Sigma, \delta, q_1, F_2)$ dove

- $Q = Q_1 \cup Q_2$
- Lo stato iniziale di N è quello di N_1, quindi q_1
- Gli stati di accettazione di N sono quelli di N_2, quindi F_2
- Per $q \in Q,\ a \in \Sigma_\epsilon$

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1,\ q \notin F_1 \\ \delta_1(q, a) & q \in F_1,\ a \neq \epsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1,\ a = \epsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

2.7.3 REG è chiusa rispetto a *

Idea: si ha un linguaggio regolare A_1 e si vuole dimostrare che A_1^* è regolare. L'idea è quella di prendere un NFA N_1 per il linguaggio A_1 e modificarlo per riconoscere A_1^* aggiungendo degli ϵ-archi che tornano allo stato iniziale ogni volta che si raggiunge uno stato accettante. Inoltre bisogna fare in modo che N accetti anche ϵ in quanto fa sempre parte di A_1^*. Questo si fa aggiungendo un nuovo stato iniziale che è a sua volta anche uno stato accettante che viene collegato al vecchio stato iniziale con un ϵ-arco. Sia

$$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$

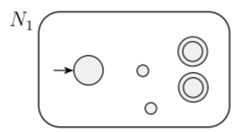


Figure 10: NFA N_1

L'automa viene modificato nel seguente modo usando il **non determinismo**:

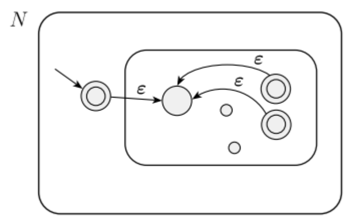


Figure 11: NFA N che rappresenta l'operazione * su N_1

N viene definito nel seguente modo: $N = (Q, \Sigma, \delta, q_0, F)$ dove

- $Q = \{q_0\} \cup Q_1$
- Lo stato q_0 è il nuovo stato iniziale
- $F = \{q_0\} \cup F_1$
- Per $q \in Q,\ a \in \Sigma_\epsilon$

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1,\ q \notin F_1 \\ \delta_1(q,a) & q \in F_1,\ a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1,\ a = \epsilon \\ \{q_1\} & q = q_0,\ a = \epsilon \\ \emptyset & q = q_0,\ a \neq \epsilon \end{cases}$$

3 Espressioni regolari

3.1 Introduzione

Le **espressioni regolari** sono l'equivalente delle espressioni algebriche per i linguaggi. Ad esempio

- Nell'aritmetica: $(5+3) \times 4$
- Nei linguaggi: $(0 \cup 1)0^*$

Dove $(0 \cup 1)0^*$ sta a significare:

1. $(0 \cup 1) \equiv \{0\} \cup \{1\} = \{0, 1\}$
2. $0^* \equiv \{0\}^*$
3. Infine: $(0 \cup 1)0^* \equiv \{0, 1\} \circ \{0\}^*$

Questa **espressione regolare** rappresenta il linguaggio formato dalle stringhe che **iniziano** per 0 o per 1 **seguite** da un qualsiasi numero di 0. L'ordine delle operazioni è il seguente: $*$, $\circ$, $\cup$ a meno di riordinamenti dati dalle parentesi.

3.2 Definizione Espressione Regolare

Sia Σ l'alfabeto. Un'**espressione regolare** su Σ, la cui notazione è la seguente $re(\Sigma)$, è definita per ricorsione:

- **Caso base**:
$$\begin{cases} \emptyset \in re(\Sigma) \\ \epsilon \in re(\Sigma) \\ a \in re(\Sigma),\ a \in \Sigma \end{cases}$$
- **Caso induttivo**:
$$\begin{cases} R_1 \cup R_2 & R_1, R_2 \in re(\Sigma) \\ R_1 \circ R_2 & R_1, R_2 \in re(\Sigma) \\ R_1^* & R_1 \in re(\Sigma) \end{cases}$$

Linguaggio $L(r)$ associato all'espressione regolare $r \in re(\Sigma)$:

- **Caso base**:
$$\begin{cases} r = \emptyset,\ L(r) = \emptyset \\ r = \epsilon,\ L(r) = \{\epsilon\} \\ r = a \in \Sigma,\ L(r) = \{a\} \end{cases}$$
- **Passo induttivo**:
$$\begin{cases} r = R_1 \cup R_2,\ L(r) = L(R_1) \cup L(R_2) \\ r = R_1 \circ R_2,\ L(r) = L(R_1) \circ L(R_2) \\ r = R_1^*,\ L(r) = L(R_1)^* \end{cases}$$

Nota: R indica un'**espressione regolare**, $L(R)$ indica il **linguaggio associato** a tale espressione regolare.

3.2.1 Esempi di espressioni regolari

Sia $\Sigma = \{0, 1\}$ l'alfabeto:

- $0^*10^* = \{w : w$ contiene esattamente un $1\}$
- $\Sigma^*1\Sigma^* = \{w : w$ contiene almeno un $1\}$
- $\Sigma^*001\Sigma^* = \{w : w$ contiene la stringa 001 come sottostringa$\}$
- $(0 \cup 1000)^*$ ogni occorrenza di 1 è seguita da 3 zeri.
- $(0 \cup \epsilon)(1 \cup \epsilon) = \{\epsilon, 0, 1, 01\}$

Convenzioni:

- $1^*\emptyset = \emptyset$. Concatenando l'insieme vuoto a qualsiasi insieme genera l'insieme vuoto.
- $\emptyset^* = \{\epsilon\}$. L'operazione * sull'insieme vuoto genera la stringa vuota.

3.3 Teorema - Un linguaggio è regolare se e solo se un'espressione regolare lo descrive

Un linguaggio è **regolare** se e solo se un'**espressione regolare lo descrive**, ovvero $L(re) = L(\text{DFA})$.

3.3.1 Lemma - $L(re) \subseteq L(\textbf{DFA})$

Se un linguaggio è descritto da un'**espressione regolare** allora è **regolare**.

Dimostrazione: data r **espressione regolare**, costruiamo un DFA M_r tale che $L(M_r) = L(r)$. Tale DFA viene creato per induzione usando i casi base ed induttivi della definizione di espressione regolare.

- **Caso base**:
  - $r = a \in \Sigma$, quindi $L(r) = \{a\}$. Il seguente DFA riconosce $L(r)$

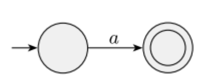


Figure 12: DFA che riconosce $L(r)$

La descrizione formale di M_r è la seguente:

$$M_r = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$$

dove la δ è la seguente:

$$\begin{cases} \delta(q_1, a) = \{q_2\} \\ \delta(q, b) = \emptyset & q \neq q_1, b \neq a \end{cases}$$

- $r = \epsilon$, quindi $L(r) = \{\epsilon\}$

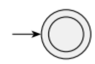


Figure 13: DFA che riconosce $L(r)$

La descrizione formale di M_r è la seguente:

$$M_r = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$$

e $\delta(q, b) = \emptyset\ \forall q, b$.

- $r = \emptyset$, quindi $L(r) = \emptyset$

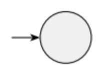


Figure 14: DFA che riconosce $L(r)$

La descrizione formale di M_r è la seguente:

$$M_r = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$$

e $\delta(q, b) = \emptyset\ \forall q, b$.

- **Caso induttivo**: siano R_1, R_2 due espressioni regolari
  - $r = R_1 \cup R_2$ allora esistono due DFA M_1, M_2 che riconoscono $L(R_1) = L(M_1),\ L(R_2) = L(M_2)$
  - $r = R_1 \circ R_2$
  - $r = R_1^*$

  Per questi tre casi si usano le costruzioni fornite per la dimostrazione della chiusura dei linguaggi regolari rispetto alle operazioni regolari.

3.3.2 Esempio

Sia $r = (ab \cup a)^*$. Costruire NFA N che accetta $L(r)$.

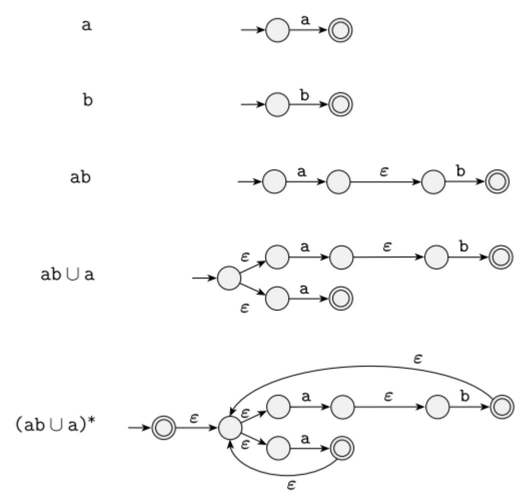


Figure 15: Conversione di r in un NFA che lo riconosce.

3.3.3 Lemma - $L(\mathbf{DFA}) \subseteq L(re)$

Se un linguaggio è **regolare** allora è descritto da un'**espressione regolare**.

Dimostrazione: se L è **regolare** allora è generato da un'**espressione regolare**. Siccome L è regolare esiste un DFA M tale che $L(M) = L$. Vogliamo convertire M in un'espressione regolare che genera lo stesso linguaggio e questo verrà fatto attraverso il concetto di **automa generalizzato**.

GNFA(Generalized Nondeterministic Finite Automaton): a differenza degli NFA, i GNFA hanno archi etichettati da **espressioni regolari** che gli permettono di leggere blocchi di input alla volta.

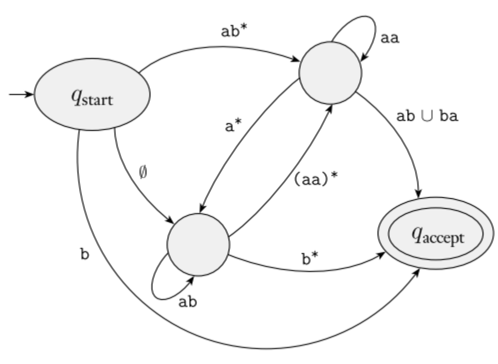


Figure 16: Un esempio di GNFA

Per convenienza, viene richiesto che il GNFA sia in **forma canonica**, ovvero rispetti le seguenti proprietà:

- Lo stato iniziale ha degli archi **uscenti** collegati ad ogni altro stato ma non ha **nessun arco entrante**
- C'è un solo stato accettante che ha archi **entranti** per tutti gli stati e **nessun arco uscente**. Inoltre è **diverso dallo stato iniziale**
- Ad eccezione per lo stato iniziale e quello accettante, ogni stato è collegato con **tutti gli altri stati** attraverso un singolo arco (incluso se stesso)

<u>Conversione da DFA in GNFA</u>:

- Aggiungo il nuovo **stato iniziale** connesso attraverso ϵ-archi ai vecchi stati iniziali
- Aggiungo il nuovo **stato finale** usando ϵ-archi
- Gli archi con **etichette multiple** vengono rimpiazzati da un arco avente per etichetta l'unione delle etichette precedenti
- Aggiungo archi con etichetta $\emptyset$ per gli stati che **non avevano collegamenti** (in realtà questi archi possono essere omessi in quanto non vanno a modificare il linguaggio riconosciuto)

<u>Conversione da GNFA a espressione regolare</u>: si parte da un GNFA con k stati. In quanto sappiamo che il GNFA deve avere almeno **uno stato iniziale** ed **uno finale**, sappiamo che $k \geq 2$.

- Se $k > 2$, costruiamo il GNFA **equivalente** con $k - 1$ stati
- Se $k = 2$, il GNFA ha un **unico arco** che va dallo stato iniziale a quello finale con **etichetta l'espressione regolare equivalente**.

<u>Rimozione di uno stato in un GNFA</u>: per passare da un GNFA con k stati in uno con $k - 1$ stati bisogna **levare uno stato**. Scegliamo uno stato da rimuovere (ne va bene uno qualsiasi, a patto che non sia quello iniziale o quello di accettazione) e chiamiamolo q_{rip}. Dopo averlo rimosso, compensiamo alla rimozione di q_{rip} andando a sistemare tutte le etichette degli archi rimanenti in modo da poter riconoscere sempre lo stesso linguaggio.
Siano q_i e q_j due stati collegati a q_{rip} prima della sua eliminazione. La nuova etichetta sull'arco che collegherà q_i e q_j dopo l'eliminazione di q_{rip} sarà un'**espressione regolare** che descrive tutte le stringhe che porterebbero la macchina dallo stato q_i allo stato q_j sia **direttamente** che **indirettamente**, ovvero passando prima per q_{rip}. Questo nuovo arco avrà la seguente etichetta:

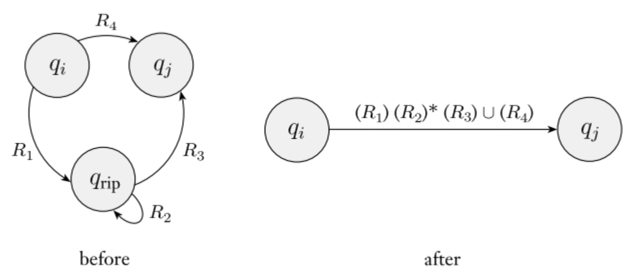


Figure 17: Etichette prima e dopo dell'eliminazione di q_{rip}

3.3.3.1 Definzione - GNFA Un GNFA è una quintupla definita nel seguente modo:

$$(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{acc}})$$

dove:

- Q è l'insieme finito di stati
- Σ è l'alfabeto di input
- $\delta : Q \setminus \{q_{\text{acc}}\} \times Q \setminus \{q_{\text{start}}\} \to \mathcal{R}$, dove $\mathcal{R}$ rappresenta l'insieme delle **espressioni regolari** su Σ
- q_{start} è lo stato iniziale
- q_{acc} è lo stato di accettazione

Inoltre diciamo che un GNFA **accetta** $w \in \Sigma^*$ se $w = w_1 w_2 ... w_k,\ w_i \in \Sigma^*$ ed esiste una sequenza di stati $q_0, q_1, ..., q_k$ tali che

1. $q_0 = q_{\text{start}}$ è lo stato iniziale
2. $q_k = q_{\text{accept}}$ è lo stato di accettazione
3. $\forall i,\ w_i \in L(R_i)$ dove $R_i = \delta(q_{i-1}, q_i)$ o in altre parole, R_i è l'espressione che si trova sull'etichetta dell'arco che va da q_{i-1} a q_i.

3.3.3.2 Algoritmo di conversione

Algoritmo di conversione:

Convert(G):

- Se $k = 2$, restituisci **espressione regolare** che collega q_{start} a q_{acc}
- Se $k > 2$, scelgo $q_{\text{rip}} \in Q \setminus \{q_{\text{start}}, q_{\text{acc}}\}$ e definisco G' che sarà

$$G' = (Q', \Sigma, \delta', q_{\text{start}}, q_{\text{acc}})$$
$$Q' = Q \setminus \{q_{\text{rip}}\}$$

Dove δ' è definita nel seguente modo:
$\forall q_i \in Q' \setminus \{q_{\text{acc}}\},\ q_j \in Q' \setminus \{q_{\text{start}}\}$

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup R_4$$

dove

- $R_1 = \delta(q_i, q_{\text{rip}})$
- $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$
- $R_3 = \delta(q_{\text{rip}}, q_j)$
- $R_4 = \delta(q_i, q_j)$

Infine richiamo **Convert(G')**.

3.3.3.3 Conclusione delle dimostrazione

Dimostriamo ora che **Convert(G)** restituisce il valore corretto.

Per ogni GNFA G, **Convert(G)** è equivalente a G.
Dimostrazione: per induzione sul numero k di stati:

- $k = 2$. Banale in quanto se G ha solo due stati, l'etichetta sull'arco che li collega è l'espressione regolare che descrive tutte le stringhe che permettono a G di raggiungere lo stato di accettazione
- Assumiamo sia vero per $k-1$ stati. Basta mostrare che G e G' riconoscono lo stesso linguaggio.

  Diciamo che se G accetta w, allora anche G' lo accetta e viceversa.

  $\Rightarrow$ G accetta w, allora anche G' lo accetta.
  In G c'è un ramo accettante $q_{\text{start}}, q_1, ..., q_{\text{acc}}$.
    - Se in questi stati q_{rip} non c'è, allora sicuramente G' accetta w.
    - Se q_{rip} c'è allora **Convert(G)** lo rimuove aggiornando correttamente le espressioni regolari su ciascuna etichetta, facendo in modo che G' accetti w.

  $\Leftarrow$ Se G' accetta w, gli archi in G' tengono conto di tutte le possibili transizioni di stati **dirette** o **indirette** attraverso q_{rip}. Quindi G accetta w. Ma G' ha $k-1$ stati e il teorema segue dall'ipotesi induttiva.
  $\square$

3.3.4 Esempio espressione regolare associata a DFA 1

Trovare l'espressione regolare equivalente al seguente DFA:

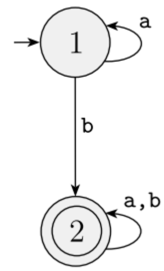


Figure 18: DFA di cui si vuole ricavare l'espressione regolare associata

I seguenti passaggi mostrano come passare da un DFA ad un GNFA da cui verrà ricavata l'espressione regolare tramite l'algoritmo **Convert(G)**:

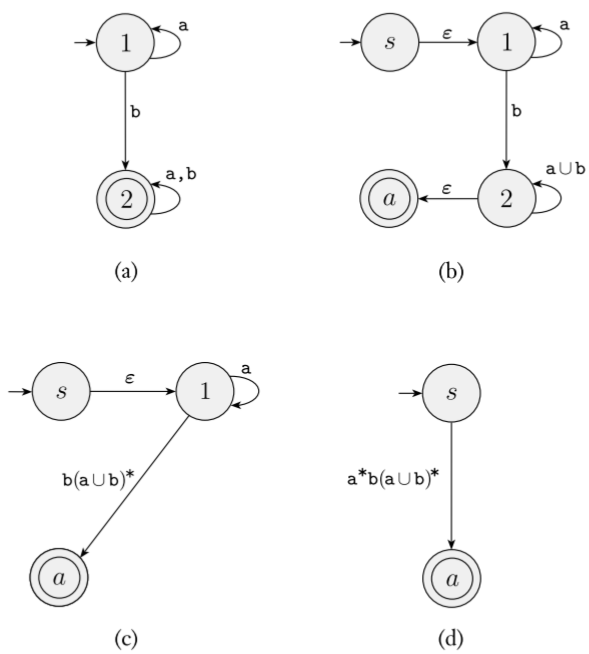


Figure 19: Da DFA a GNFA a espressione regolare

Nella figura 19(b) viene creato il GNFA a partire dal DFA raffigurato nella figura 18 a cui è stato aggiunto il nuovo stato iniziale e quello di accettazione.
Notare come nell'arco che da 2 va in se stesso l'etichetta a, b è stata rimpiazzata dall'espressione regolare $a \cup b$.
Nella figura 19(c) è stato rimosso lo stato 2 e sono state aggiornate tutte le etichette degli archi rimanenti in modo da non alterare il linguaggio riconosciuto.

Questo risultato viene ottenuto dall'applicazione dell'algoritmo **Convert(G)** prendendo $q_i = 1, q_j = a$ e $q_{\text{rip}} = 2$ e ottenendo $R_1 = b, R_2 = (a \cup b), R_3 = \epsilon$ e $R_4 = \emptyset$. Quindi l'espressione sul nuovo arco sarà

$$(R_1)(R_2)^*(R_3) \cup R_4 \equiv (b)(a \cup b)^*(\epsilon) \cup \emptyset$$

Che dopo essere semplificato diventa $b(a \cup b)^*$.
Nella figura 19(d) viene rimosso lo stato 1 da 19(c) seguendo la stessa procedura. Siccome rimangono solo lo stato iniziale e quello di accettazione, l'etichetta associata all'arco che va tra questi due stati è l'**espressione regolare** equivalente al DFA nella figura 18.

3.3.5 Esempio espressione regolare associata a DFA 2

Vediamo un esempio più complesso. Trovare l'espressione regolare equivalente al seguente DFA:

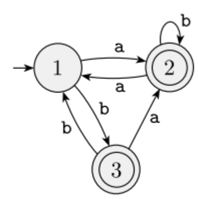


Figure 20: DFA di cui si vuole ricavare l'espressione regolare associata

Gli step per la conversione sono i seguenti:

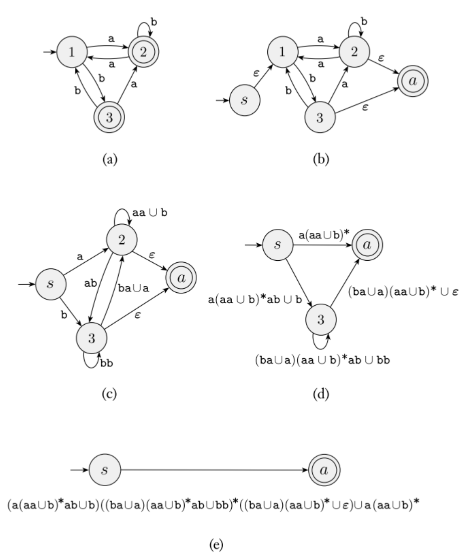


Figure 21: Da DFA a GNFA a espressione regolare

4 Linguaggi non regolari

Non tutti i linguaggi sono regolari. Analizziamo il linguaggio

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

Se si cerca di realizzare un DFA che riconosca L, si scopre presto che si dovrebbe fare in modo che l'automa ricordi il numero di 0 che incontra. Siccome il numero di 0 non è limitato poiché n potrebbe essere un numero infinitamente grande, l'automa dovrebbe tener conto di un numero illimitato di possibilità, cosa che non può essere fatta con un numero **finito** di stati.
Se il numero di stati di un DFA M è finito, ma l'input ha una dimensione maggiore del numero di stati, deve succedere la seguente cosa:

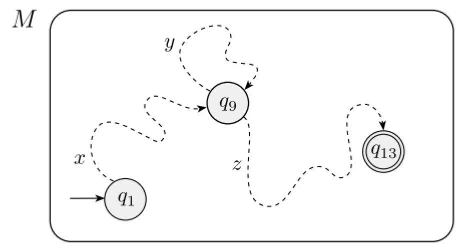


Figure 22: Ripetizione di uno stato più di una volta

4.1 Pumping lemma

Tecnica utilizzata per mostrare la **non regolarità** di un linguaggio. Questa tecnica utilizza il fatto che tutti i linguaggi regolari hanno una certa proprietà, che se si dimostra non essere vera per un certo linguaggio, si può **affermare con certezza che quest'ultimo non sia regolare**. Questa proprietà dice che tutte le stringhe del linguaggio possono essere "pompate" (pumped) se sono lunghe almeno quanto un certo valore speciale chiamato **lunghezza di pumping**. "Pompare" una stringa vuol dire che quest'ultima contiene una sezione che può essere ripetuta un qualsiasi numero di volte e rimanere comunque all'interno del linguaggio.

4.1.1 Teorema - Pumping lemma

Se L è un linguaggio riconosciuto da DFA M allora esiste un certo valore p (la lunghezza di **pumping**) dove se $w \in L(M)$ con $|w| \geq p$, allora w può essere spezzata in tre parti $w = xyz$ tali che

1. $\forall i \geq 0,\ xy^i z \in L(M)$
2. $|y| > 0$
3. $|xy| \leq p$ (condizione d'aiuto)

Quando w viene spezzata in xyz, queste condizioni impongono che solo x o z possono essere ϵ mentre $y \neq \epsilon$ sempre.
<u>Dimostrazione</u>: siano

- $M = (Q, \Sigma, \delta, q_1, F)$ il DFA per L
- $p =$ numero di stati
- $w = w_1 w_2 ... w_n \in L$ con $n \geq p$

Sia $r_1, r_2, ..., r_{n+1}$ la **sequenza di stati attraversati** da M con input w, ovvero

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

Per il **pigeonhole principle** tra i primi $p + 1$ elementi della sequenza, due devono essere lo stesso stato. Sia il primo stato r_j e il secondo r_l. Siccome r_l viene attraversato nei primi $p+1$ stati della sequenza a partire da r_1, si ha che $l \leq p + 1$. Sia $w = xyz$ dove

$$\begin{aligned} x &= w_1 \dots w_{j-1} \\ y &= w_j \dots w_{l-1} \\ z &= w_l \dots w_n \end{aligned}$$

x porta M dallo stato r_1 allo stato r_j, y porta M dallo stato r_j allo stato r_j e z porta M dallo stato r_j allo stato r_{n+1} che è uno **stato di accettazione**, come mostrato nella figura 22. Quindi M accetta xy^iz per $i \geq 0$ (condizione 1). Inoltre sappiamo che $j \neq l$, quindi $|y| > 0$ (condizione 2) e che $l \leq p + 1$, quindi $|xy| \leq p$ (condizione 3). Sono quindi soddisfatte tutte le condizioni del **pumping lemma**. □

Per utilizzare il pumping lemma per mostrare che un linguaggio B non è regolare si seguono i seguenti passi:

- Iniziamo **assumendo** che B **sia regolare** per ottenere una **contraddizione**.
- Si utilizza il **pumping lemma** per garantire l'esistenza della lunghezza di pumping p tale che tutte le stringhe di lunghezza **maggiore o uguale** a p in B possano essere "pompate".
- Infine si dimostra che w **non può essere "pompata"** andando a considerare **tutti i modi** in cui w può essere spezzata nelle tre parti x, y e z e, per ogni suddivisione, trovare il valore i tale che $xy^iz \notin B$.

L'esistenza di questa stringa w contraddice il pumping lemma se B fosse regolare, il che significa che B non può esserlo.

4.1.1.1 Esercizio pumping lemma 1

Mostre che $L = \{0^n 1^n : n \geq 0\}$ non è regolare.
Soluzione: Assumiamo per assurdo che L sia regolare. Questo vuol dire che $\exists p$ (lunghezza di pumping) tale che per ogni $w \in L$ si può scomporre come nell'enunciato del pumping lemma. Costruisco w tale che per ogni scomposizione legale $w = xyz$ (con $|y| > 0$ e $|xy| \leq p$) $\exists i$ tale che $xy^i z \in L$.
Contraddizione: prendo $w = 0^p 1^p$ con $|w| = n = 2p > p$. Siccome $|xy| \leq p$, y contiene tutti 0.

$$w = \underbrace{0 \dots 0}_{x} \underbrace{0 \dots 0}_{y} \underbrace{1 \dots 1}_{z}$$

Per $i \geq 2$, sia $\hat{w} = xy^i z \in L$. Infatti per $i = 2$ è sufficiente:

$$\hat{w} = 0^q 1^p \quad p < q$$

Quindi la stringa $\hat{w}$ con $i \geq 2$ ha più 0 che 1.
Altri due modi che avremmo potuto scegliere di dividere w che creano una contraddizione:

- La stringa y contiene solamente 1

  $$w = \underbrace{0 \dots 0}_{x} \underbrace{1 \dots 1}_{y} \underbrace{1 \dots 1}_{z}$$

  Come per prima, $\hat{w} = xy^i z \notin L$ per $i \geq 2$ in quanto conterrebbe più 1 che 0.

- La stringa y contiene sia 0 che 1. In questo caso la stringa $\hat{w} = xy^i z$ con $i \geq 2$ potrebbe avere lo stesso numero di 0 e 1, ma l'ordine non sarebbe rispettato in quanto ci sarebbero degli 1 prima degli 0.

Importante: questi ultimi 2 casi però sono stati **esclusi immediatamente** in quanto con la condizione 3 del pumping lemma abbiamo preso $|xy| \leq p$. Essendo la stringa presa in considerazione $0^p 1^p$, la y deve apparire per forza prima del primo 1.

4.1.1.2 Esercizio pumping lemma 2

Mostrare che

$$L = \{w \in \{0,1\}^* : |w|_0 = |w|_1\}$$

dove $|w|_0 =$ "numero di 0 nella stringa" e $|w|_1 =$ numero di 1 nella stringa non è regolare.
Soluzione: Assumiamo per assurdo che sia regolare. Questo vuol dire che $\exists p$ tale che vale il pumping lemma.

- **Esempio di scomposizione che non porta ad una contraddizione**
  Prendo $w = (01)^p \in L$ con $|w| = 2p$. Scompongo $w = xyz$ nel seguente modo:

- $x = \epsilon$
- $y = 01$
- $z = (01)^{p-1}$

Ottenendo la seguente stringa:

$$w = \underbrace{01}_{y}\underbrace{01010101\ldots01}_{z}$$

Utilizzando questa scomposizione si ha che $\forall i \geq 0,\ xy^iz \in L$ e quindi L risulterebbe essere regolare anche se non lo è.

- **Esempio di scomposizione funzionante**
  Prendo $w = 0^p1^p$ con $|w| = 2p > p$. Prendo qualsiasi scomposizione $w = xyz$ t.c $|y| > 0,\ |xy| \leq p$. Questa scomposizione mi garantisce che y è **composta da soli** 0.

  Si possono presentare due casi:

  1. $w = \underbrace{00000}_{x}\underbrace{\ldots0000}_{y}\underbrace{1\ldots1}_{z}$
  2. $w = \underbrace{000}_{x}\underbrace{00\ldots0}_{y}\underbrace{\underbrace{000}_{l}\underbrace{1\ldots1}_{p}}_{z}$

     Dove l indica il numero di 0 dopo la fine di y ed il primo 1 in z, mentre p indica il numero di 1 in z.

  In entrambi i casi $\exists i : xy^iz \notin L$.

  Vediamo in dettaglio entrambi i casi:

  1. Siano: $|y| = k > 0,\ k \leq p;\ |x| = p - k;\ |z| = p$. Si ha:

     $$|xy^2z| = p - k + 2k + p = 2p + k$$
     $$\#0 = p - k + 2k = p + k$$

     Ma #0 dovrebbe essere pari a p, quindi **non è regolare**.
  2. Siano: $|y| = k > 0;\ |x| = p - k - l;\ |z| = p + l$. Si ha:

     $$|xy^2z| = p - k - l + 2k + p + l = 2p + k$$
     $$\#0 = p - k - l + 2k + l = p + k$$

     Ma, come prima, #0 dovrebbe essere pari a p, quindi **non è regolare**.

5 Linguaggi acontestuali (context-free languages - CFL)

Le **grammatiche acontestuali** (context-free grammars - CFG) sono un metodo più potente di descrivere i linguaggi. Queste grammatiche possono descrivere certe caratteristiche che hanno uno struttura **ricorsiva**, molto utili per molte applicazioni.
La collezione di linguaggi associati a grammatiche acontestuali prende il nome di **linguaggi acontestuali** (context-free languages). Questi linguaggi vengono introdotti in quanto non tutti i linguaggi sono regolari e serve un modo per poterli descrivere in modo formale.
Il prossimo è un esempio di CFG, chiamiamola G_1:

$$\begin{aligned} &A \to 0A1 \\ &A \to B \qquad \text{(regole)} \\ &B \to \# \end{aligned}$$

Una grammatica consiste di una collezione di **regole di sostituzione**, chiamate anche **produzioni**. Le regole contengono **variabili** e **terminali**:

- $0, 1, \#$ sono **terminali**, spesso rappresentate tramite l'utilizzo di lettere minuscole, numeri o simboli speciali
- A, B sono **variabili**, spesso rappresentate tramite l'utilizzo di lettere maiuscole

La variabile che si trova nella prima regola, a sinistra delle freccia si chiama **variabile iniziale**.
Generazione delle stringhe:

- Scrivo la **variabile iniziale**
- Rimpiazzo la variabile con la stringa che compare a **destra** di una delle regole
- Ripeto finché non ottengo **tutte variabili terminali**

Ad esempio la grammatica G_1 può generare la stringa 000#111. La serie di sostituzioni per ottenere questa stringa si chiama **derivazione**. La derivazione di 000#111 è la seguente:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Si possono rappresentare le stesse informazioni in modo grafico, attraverso un **albero sintattico**:

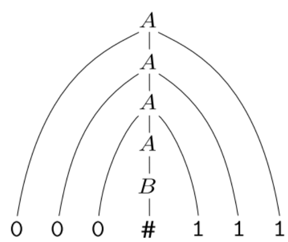


Figure 23: Albero sintattico della derivazione della stringa 000#111 usando la CFG G_1

Tutte le stringhe generate in questo modo costituiscono il **linguaggio della grammatica**.
Scriviamo $L(G_1)$ per rappresentare il linguaggio della grammatica G_1. Ogni linguaggio che può essere generato da una CFG viene chiamato **linguaggio acontestuale** (context-free language - CFL).

5.1 Definizione CFG

Una **grammatica acontestuale** (CFG) è una quartupla

$$(V, \Sigma, R, S)$$

Dove:

- V è l'insieme finito delle **variabili**
- Σ è l'insieme finito di **terminali**
  Si ha che $V \cap \Sigma = \emptyset$
- R è l'insieme finito di **regole**
- $S \in V$ è la **variabile iniziale**

Se u, v e w sono stringhe composte da variabili e terminali $(\Sigma \cup V)$ e $A \to v$ è una regola della grammatica, diciamo che uAw **produce** uvw, scritto come $uAw \Rightarrow uvw$.
Inoltre diciamo che u **deriva** v, scritto come $u \overset{*}{\Rightarrow} v$, se:

- $u = v$, oppure
- Esiste una sequenza di $u_1, u_2, \ldots u_k$ con $k \geq 0$ e

$$u \Rightarrow u_1 \Rightarrow u_2 \cdots \Rightarrow u_k \Rightarrow v$$

Tramite questa definizione, possiamo definire il **linguaggio associato alla grammatica** $G = (U, \Sigma, R, S)$ come:

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

5.1.0.1 Esempio CFG 1

Si consideri la grammatica $G_3 = (\{S\}, \{a, b\}, R, S)$. Le regole in R sono:

$$S \rightarrow aSb \mid SS \mid \epsilon$$

Questa grammatica può generare le seguenti stringhe: $abab, aaabbb, aababb$. Ma non genera ad esempio la seguente stringa: ba.
Possiamo pensare $L(G_3)$ come il linguaggio di tutte le stringhe contenenti parentesi aperte e chiuse in modo corretto.

5.1.0.2 Esempio CFG 2

Si consideri la grammatica $G_4 = (V, \Sigma, R, \langle\text{EXPR}\rangle)$.

- $V = \{\langle\text{EXPR}\rangle, \langle\text{TERM}\rangle, \langle\text{FACTOR}\rangle\}$
- $\Sigma = \{a, +, \times, (,)\}$
- R

$$\begin{aligned}
\langle\text{EXPR}\rangle &\rightarrow \langle\text{EXPR}\rangle + \langle\text{TERM}\rangle | \langle\text{TERM}\rangle \\
\langle\text{TERM}\rangle &\rightarrow \langle\text{TERM}\rangle \times \langle\text{FACTOR}\rangle | \langle\text{FACTOR}\rangle \\
\langle\text{FACTOR}\rangle &\rightarrow (\langle\text{EXPR}\rangle) | a
\end{aligned}$$

Le due stringhe $a + a \times a$ e $(a + a) \times a$ possono essere generate dalla grammatica G_4. Di seguito i rispettivi alberi sintattici:

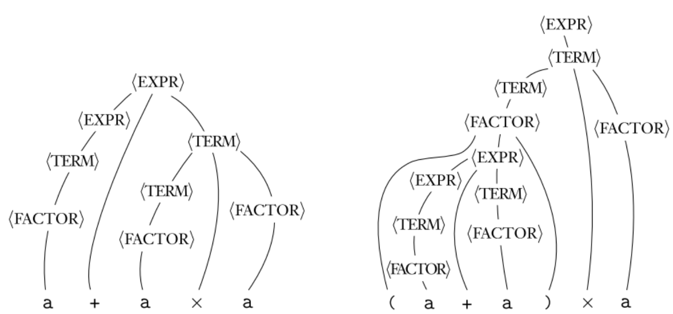


Figure 24: Alberi sintattici della derivazione delle stringa $a + a \times a$ e $(a + a) \times a$

5.1.1 Progettazione di grammatiche acontestuali

Alcune linee guida per facilitare la progettazione di grammatiche acontestuali:

5.1.1.1 Unione di grammatiche

Spesso molti linguaggi acontestuali sono l'unione di altri linguaggi acontestuali **più semplici**.
Supponiamo di avere delle grammatiche $G_i = (V_i, \Sigma_i, R_i, S_i)$ con associato $L(G_i)$. Risulta facile definire una grammatica $G = (V, \Sigma, R, S)$ tale che

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

- $V = \bigcup_i V_i \cup \{S\}$
- $\Sigma = \bigcup_i \Sigma_i$
- $R = \bigcup_i R_i \cup \{S \to S_1|S_2|\dots|S_k\}$

Devo mostrare che $L(G) \subseteq \bigcup_i L(G_i)$ e che $L(G) \supseteq \bigcup_i L(G_i)$.
Vediamo la seconda: sia $w \in \bigcup_i L(G_i)$, allora $\exists j : w \in L(G_j)$. Questo significa che $S_j \overset{*}{\Rightarrow} w$ in G_j. In G abbiamo $S \to S_j \overset{*}{\Rightarrow} w$ che implica che $w \in L(G)$.
L'altra direzione viene lasciata come esercizio.
Esempio: $L = \{0^n 1^n : n \geq 0\} \cup \{1^n 0^n : n \geq 0\}$

- Le regole di G_1 saranno: $S_1 \to 0S_11 \mid \epsilon$
- Le regole di G_2 saranno: $S_2 \to 1S_20 \mid \epsilon$
- Le regole di G saranno:

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to 0S_11 \mid \epsilon \\ S_2 &\to 1S_20 \mid \epsilon \end{aligned}$$

5.1.1.2 Passaggio da DFA a CFG Costruire una CFG per un linguaggio regolare è semplice se prima si costruisce un DFA che riconosce tale linguaggio.
Costruzione della CFG: sia $L \in \text{REG}$ ed $\exists \text{DFA } M : L(M) = L$

1. Introduco una **variabile** V_i per ogni **stato** q_i di M
2. Aggiungo la regola $V_i \to aV_j$ se in M vale $\delta(q_i, a) = q_j$
3. Se q_i è lo **stato di accettazione**, aggiungo la regola $V_i \to \epsilon$
4. V_0 è la **variabile iniziale** dove q_0 è lo **stato iniziale** di M

5.1.1.3 Memoria illimitata Alcuni linguaggi acontestuali contengono delle stringhe contenenti due sottostringhe collegate in modo tale che bisognerebbe ricordare un **numero illimitato di informazioni** su una delle sottostringhe per verificare che l'altra gli corrisponda in modo corretto. Questo accade ad esempio per il linguaggio

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

in quanto bisognerebbe ricordare il numero di 0 per verificare che sia pari al numero di 1. Si può generare la seguente CFG che sfruttando la **ricorsione** genera in modo corretto le possibili stringhe di questo formato:

$$R \to 0R1$$

Un altro esempio può essere il linguaggio

$$L = \{w \in \{0,1\}^* : w \text{ contiene almeno tre } 1\}$$

$$\begin{aligned} S &\to X1X1X1X \\ X &\to \epsilon \mid 0X \mid 1X \end{aligned}$$

5.1.2 Ambiguità

A volte una grammatica acontestuale può generare la stessa stringa in **modi differenti**. Una tale stringa avrà **diversi alberi sintattici** e di conseguenza **diversi significati**. In alcune applicazioni questo comportamento potrebbe portare dei risultati indesiderati.
Se una CFG genera la stessa stringa in modi differenti, diciamo che la stringa è **derivata in modo ambiguo** da tale grammatica. Se una CFG genera stringhe **ambigue**, diciamo che la **grammatica è ambigua**.
Esempio: consideriamo la seguente CFG G_5:

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid a$$

Questa grammatica genera la stringa $a + a \times a$ in modo ambiguo. Questi sono i due alberi sintattici che la generano:

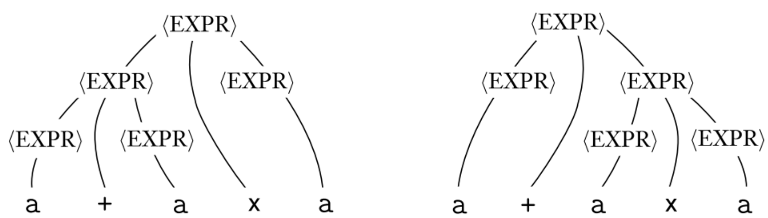


Figure 25: Alberi sintattici che generano la stringa ambigua $a + a \times a$

5.1.2.1 Definizione - Derivazione a sinistra

Una derivazione di una stringa in una CFG è una **derivazione a sinistra** se ad ogni passo la variabile sostituita è quella **più a sinistra**:

$$\begin{aligned} &\ldots \to aAbCc\ldots \\ &A \to \ldots \quad \text{(sostituisco } A \text{ per prima)} \end{aligned}$$

Una stringa w è derivata ambiguamente se ha due o più **derivazioni a sinistra differenti**. Se una tale stringa esiste, la CFG si dice **ambigua**.

5.1.3 Forma normale di Chomsky

Una CFG è in **forma normale di Chomsky** se ha solo regole del tipo:

$$\begin{aligned} A &\to BC \\ A &\to a \end{aligned}$$

Dove:

- a è un qualsiasi **terminale**

- A, B, C sono **variabili** qualsiasi, ma B e C non devono essere **variabili iniziali**
- É permessa la regola facoltativa $S \to \epsilon$, dove S è la **variabile iniziale**

5.1.3.1 Teorema - Ogni linguaggio acontestuale è generato da una grammatica acontestuale canonica

Ogni CFL è generato da una CFG canonica.
Idea: Convertiamo una qualsiasi CFG G in forma normale di Chomsky tramite una **serie di fasi** dove le regole che **violano le condizioni** della forma normale vengono **rimpiazzate** con delle **regole equivalenti** che le rispettano.

Nota: usiamo la seguente grammatica per i prossimi esempi:

$$\begin{aligned} &S \to ASA \mid aB \\ &A \to B \mid S \\ &B \to b \mid \epsilon \end{aligned}$$

Dimostrazione: Come prima cosa aggiungiamo una nuova **variabile iniziale** S_0 e la regola $S_0 \to S$, dove S è la **variabile iniziale originale**. Questo viene fatto per garantire che la variabile iniziale non appaia **mai a destra** di una qualsiasi regola:

$$\begin{aligned} &S_0 \to S \\ &S \to ASA \mid aB \\ &A \to B \mid S \\ &B \to b \mid \epsilon \end{aligned}$$

Le altre regole vengono cambiate nel seguente modo:

- **ϵ-regole**: rimuoviamo ogni ϵ-regola della forma $A \to \epsilon$ dove A non è una variabile iniziale. Dopodiché per ogni occorrenza di A a **destra** di una regola, aggiungiamo una nuova regola rimuovendo l'occorrenza di A:

  $$\begin{aligned} &R \to uAv \overset{\text{aggiungo}}{\rightsquigarrow} R \to uv \\ &\cancel{A \to \epsilon} \end{aligned}$$

  Questo viene ripetuto per ogni occorrenza, ad esempio:

  $$\begin{aligned} &R \to uAvAw \\ &A \to \epsilon \end{aligned}$$

  La prima regola e la presenza di $A \to \epsilon$ ci forzano ad aggiungere le regole:

  - $R \to uvAw$ (prima occorrenza di A rimossa)
  - $R \to uAvw$ (seconda occorrenza di A rimossa)

- $R \to uvw$ (tutte le occorrenze di A rimosse)

Rimuovendo la ϵ-regola $B \to \epsilon$, la nostra grammatica diventa:

$$
\begin{aligned}
&S_0 \to S \\
&S \to ASA \mid aB \mid a \\
&A \to B \mid S \mid \epsilon \\
&B \to b
\end{aligned}
$$

La rimozione ha generato una nuova ϵ-regola $A \to \epsilon$ che andiamo a rimuovere:

$$
\begin{aligned}
&S_0 \to S \\
&S \to ASA \mid aB \mid a \mid SA \mid AS \mid S \\
&A \to B \mid S \\
&B \to b
\end{aligned}
$$

Le ϵ-regole sono finite, quindi passiamo alla prossima fase.

- **Regole unitarie**: rimuoviamo una regola $A \to B$ e per ogni regola del tipo $B \to u$ che appare **aggiungiamo** la regola $A \to u$, a meno che non sia una regola unitaria **precedentemente già rimossa**. Come prima, il procedimento viene ripetuto per tutte le regole unitarie:

  - Rimuovo $S \to S, S_0 \to S$

$$
\begin{aligned}
&S_0 \to ASA \mid aB \mid a \mid SA \mid AS \\
&S \to ASA \mid aB \mid a \mid SA \mid AS \\
&A \to B|S \\
&B \to b
\end{aligned}
$$

  - Rimuovo $A \to B, A \to S$

$$
\begin{aligned}
&S_0 \to ASA \mid aB \mid a \mid SA \mid AS \\
&S \to ASA \mid aB \mid a \mid SA \mid AS \\
&A \to b \mid ASA \mid aB \mid a \mid SA \mid AS \\
&B \to b
\end{aligned}
$$

  Le regole unitarie sono finite, quindi passiamo alla prossima fase.

- **Regole rimanenti**: alla fine convertiamo le regole rimanenti nella forma corretta. Ogni regola della forma $A \to u_1 u_2 \dots u_k$ con $k \geq 3$ dove u_i può essere o una **variabile** o un **terminale** viene rimpiazzata con la formula:

$$A \to u_1 A_1, A_1 \to u_2 A_2, A_2 \to u_3 A_3, \dots, A_{k-2} \to u_{k-1} u_k$$

dove le A_i sono nuove variabili. Vengono rimpiazzati tutti i terminali u_i nelle regole precedenti con la nuova variabile U_i e aggiungiamo la regola $U_i \to u_i$. Convertiamo ora le regole rimanenti nella nostra grammatica di esempio:

$$
\begin{aligned}
&S_0 \to AA_1 \mid UB \mid a \mid SA \mid AS \\
&A_1 \to SA \\
&U \to a \\
&S \to AA_1 \mid UB \mid a \mid SA \mid AS \\
&A \to b \mid AA_1 \mid UB \mid a \mid SA \mid AS \\
&B \to b
\end{aligned}
$$

Questa grammatica è ora in **forma normale di Chomsky**.

6 Automa a pila - PDA

Nuovo modello di computazione chiamato **Automa a pila** (Pushdown Automata - PDA). Questi automi sono come degli NFA ma hanno un certo tipo di **memoria** chiamato **stack** che gli permette di **riconoscere** alcuni **linguaggi non regolari**. Vedremo che i PDA sono equivalenti alle grammatiche acontestuali. Questa equivalenza è molto utile in quanto per provare che un linguaggio è acontestuale possiamo fornire un grammatica acontestuale che lo **genera** oppure un PDA che lo **riconosce**. Un PDA può essere rappresentato tramite il seguente schema:

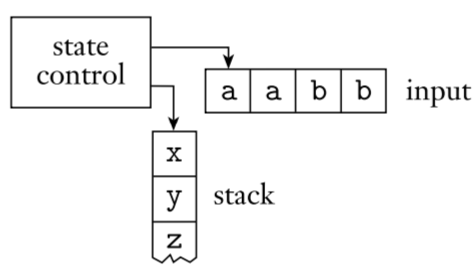


Figure 26: Schema di un PDA generico

Dove:

- Lo **state control** rappresenta gli stati e la funzione di transizione
- Il **nastro di input** contiene la stringa di input
- La **freccia** rappresenta la testina dell'input che punta al prossimo simbolo da leggere

- Lo **stack** (o pila) rappresenta la nuova memoria introdotta. Lo **stack** è un tipo di memoria LIFO (Last In, First Out), il che significa che tutte le operazioni che verranno tra poco descritte potranno essere fatte solamente in cima allo stack.

Le operazioni sullo stack che si possono eseguire sono:

- **Pop**: viene **rimosso** l'elemento in cima allo stack
- **Push**: viene **aggiunto** un elemento in cima allo stack

Esempio: Esempio di etichetta di un PDA

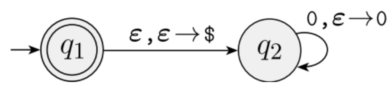


Figure 27: Esempio di operazioni su PDA

Il simbolo che si trova prima della virgola indica l'**input**, quello che si trova dopo la virgola indica l'operazione sullo stack:

- $b \to c$: **pop** di b e **push** di c contemporaneamente sullo stack
- $\epsilon \to c$: **push** di c sullo stack
- $b \to \epsilon$: **pop** di b sullo stack

Nell'esempio soprastante $\epsilon, \epsilon \to \$$ significa che se il PDA si trova nello stato q_1 e legge ϵ, allora passa allo stato q_2 e fa un **push** del simbolo \$ sullo stack.

6.1 Definizione - PDA

Una **PDA** è una sestupla

$$(Q, \Sigma, \Gamma, \delta, q_0, F)$$

Dove:

- Q è l'insieme finito di **stati**
- Σ è l'**alfabeto di input**
- Γ è l'**alfabeto finito dello stack**
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ è la **funzione di transizione**. Questo significa che lo stato corrente, il simbolo letto sul nastro di input e il simbolo che si trova in cima allo stack determinano la prossima mossa del PDA
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'insieme di **stati di accettazione**

Un PDA M accetta $w = w_1 w_2 \dots w_m,\ w_i \in \Sigma_\epsilon$ ed esistono una sequenza di stati $r_0, \dots, r_m \in Q$ e stringhe $s_0, s_1, \dots, s_m \in \Gamma^*$ tali che:

- $r_0 = q_0$ e $s_0 = \epsilon$. Questa condizione indica che M parte dallo stato iniziale e lo stack è vuoto
- $\forall i \in \{0, \dots, m-1\}$ si ha che $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, dove $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\epsilon$ e $t \in \Gamma^*$. Questa condizione indica che M si muove nel modo corretto in base allo stato, stack e prossimo simbolo
- $r_m \in F$. Questa condizione indica che uno stato di accettazione viene raggiunto quando l'input finisce

6.1.1 Esempio PDA 1

Esempio di PDA che riconosce il linguaggio non regolare

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

Idea: finché l'automa legge 0, aggiunge sullo stack il simbolo 0 (**push**). In seguito, per ogni 1 letto, toglie uno 0 dallo stack (**pop**). Se lo stack è vuoto e si trova in uno stato di accettazione, accetta. Altrimenti rifiuta.
Quindi:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$
- $F = \{q_1, q_4\}$
- δ:

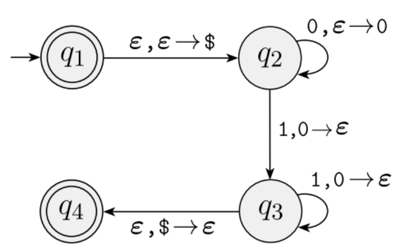


Figure 28: PDA che riconosce il linguaggio non regolare $L = \{0^n 1^n : n \geq 0\}$

Spiegazione:

- Inizia inserendo nello stack il simbolo \$ tramite un ϵ-arco. Questo simbolo viene utilizzato per vedere se a fine computazione lo **stack sarà vuoto**.

- Per ogni 0 che legge, fa **push** di uno 0 nello stack
- Successivamente, per ogni 1 che legge, fa **pop** di uno 0 dallo stack
- Infine fa **pop** di \$: se ci riesce vuol dire che lo stack è vuoto e quindi accetta, altrimenti rifiuta

6.1.2 Esempio PDA 2

PDA per il linguaggio:

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

Si ha:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, \$\}$
- $F = \{q_4\}$
- δ:

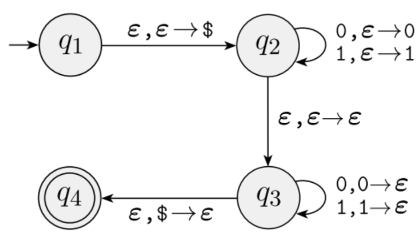


Figure 29: PDA che riconosce il linguaggio $L = \{ww^T : w \in \{0,1\}^*\}$

6.2 Teorema - Un linguaggio è acontestuale se e solo se $\exists$PDA P che lo riconosce

Un linguaggio è **acontestuale** se e solo se $\exists$PDA P che lo riconosce.

Lemma: Se un linguaggio L è **acontestuale**, allora esiste un PDA P che riconosce L.
Dimostrazione: Data L, sappiamo che c'è una CFG che lo genera, chiamiamola G.
Idea: Dato un input w, controllo non deterministicamente che esista una **serie di produzioni** in G che conduce a w.
Descrizione non formale di P:

- Inserisco il simbolo \$ nello stack

- Ripeto i seguenti step all'infinito:
  - Se sulla cima dello stack c'è una variabile A **uso il non determinismo** per sostituire A usando una delle regole di G. Una sostituzione equivale a fare un'operazione di **pop** e di **push**
  - Se in cima allo stack c'è a (terminale), faccio **pop** e controllo che a sia il carattere seguente dell'input
  - Se in cima allo stack c'è il simbolo \$, vado nello stato di accettazione e accetto solamente se tutto l'input è stato letto

<u>Costruzione di P</u>: per rendere la costruzione più chiara introduciamo una **notazione ridotta** della funzione di transizione che rende possibile **scrivere un'intera stringa sullo stack in un singolo passo della macchina**.
Quest'azione potrà poi essere simulata sul PDA aggiungendo stati aggiuntivi utilizzati per scrivere la stringa un simbolo alla volta.
Siano q e r stati del PDA e siano $a \in \Sigma_\epsilon$ e $s \in \Gamma_\epsilon$. Vogliamo che il PDA vada dallo stato q allo stato r quando legge il carattere a dall'input e fa un **pop** del carattere s. In più vogliamo che faccia un **push** dell'intera stringa $u = u_1 \dots u_l$ sullo stack nello stesso tempo. Questo comportamento può essere ottenuto **aggiungendo stati** $q_1, \dots, q_{l-1}$ e **settando la funzione di transizione** come segue:

$$
\begin{aligned}
&\delta(q, a, s) \text{ per contenere } (q_1, u_l),\\
&\delta(q_1, \epsilon, \epsilon) = \{(q_2, u_{l-1})\}\\
&\delta(q_2, \epsilon, \epsilon) = \{(q_3, u_{l-2})\}\\
&\vdots\\
&\delta(q_{l-1}, \epsilon, \epsilon) = \{(r, u_1)\}
\end{aligned}
$$

La notazione $(r, u) \in \delta(q, a, s)$ (dove u è una stringa) significa che quando l'automa si trova nello stato q, a è il prossimo simbolo da leggere e s è il simbolo che si trova in cima allo stack, allora il PDA legge a, fa un **pop** di s e di seguito un **push** della stringa u sullo stack ed infine va nello stato r. La seguente figura mostra questa implementazione:

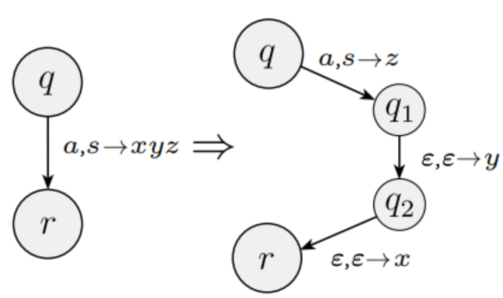


Figure 30: Implementazione dell'abbreviazione $(r, xyz) \in \delta(q, a, s)$

- Gli stati di P sono:

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

dove E è l'insieme contenente gli stati **intermedi** utilizzati per l'implementazione della forma abbreviata della funzione di transizione.

- q_{start} è lo stato iniziale
- q_{accept} è l'unico stato di accettazione
- La **funzione di transizione** è definita come segue:
    - Lo stack viene inizializzato inserendo i simboli \$ e S, quindi

    $$\delta(q_{\text{start}}, \epsilon, \epsilon) = \{(q_{\text{loop}}, S\$)\}$$

    - Per prima cosa viene gestito il caso in cui in cima allo stack c'è una **variabile**. Quindi:

    $$\delta(q_{\text{loop}}, \epsilon, A) = \{(q_{\text{loop}} : A \to w \text{ è una regola in}) \ R\}$$

    Poi viene gestito il caso in cui in cima allo stack c'è un **terminale**. Quindi:

    $$\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \epsilon)\}$$

    Infine viene gestito l'ultimo caso in cui il simbolo \$ si trova in cima allo stack, il che sta a significare che è stato svuotato completamente. Quindi:

    $$\delta(q_{\text{loop}}, \epsilon, \$) = \{(q_{\text{accept}}, \epsilon)\}$$

Il seguente diagramma mostra la funzione di transizione appena descritta:

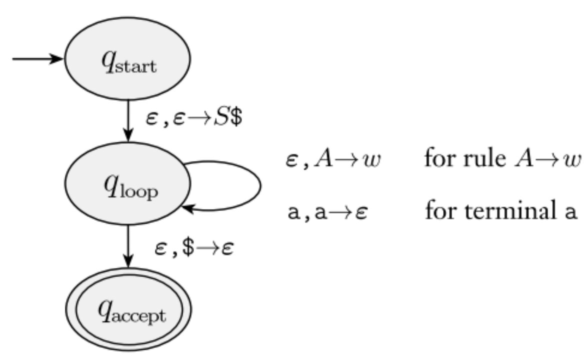


Figure 31: Diagramma di stato di P

6.2.1 Esempio di costruzione di PDA per una CFG

Esempio di costruzione del PDA P_1 per la CFG G usando il lemma precedente:

$$S \to aTb \mid b$$
$$T \to Ta \mid \epsilon$$

La corrispondente funzione di transizione è riportata nel seguente diagramma:

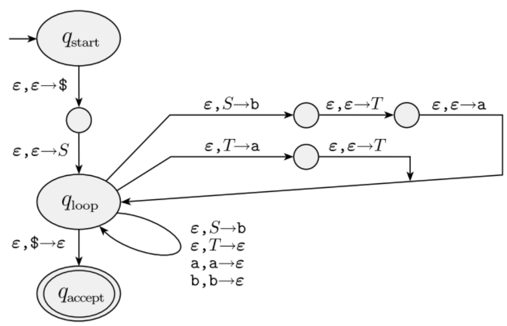


Figure 32: Diagramma a stati per P_1

<u>Lemma</u>: Se un linguaggio L è riconosciuto da un PDA P, allora L è **acontestuale**.
<u>Idea</u>: abbiamo un PDA P e vogliamo **creare** una CFG G che genera tutte le stringhe che P accetta, ovvero quelle stringhe che portano P dallo **stato iniziale** allo **stato di accettazione**.
Per semplificare il processo, modifichiamo il PDA P per far si che rispetti le seguenti tre proprietà:

1. Ha un **singolo stato di accettazione**, q_{accept}

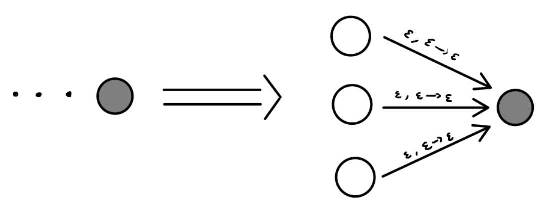


Figure 33: Implementazione della condizione 1

2. **Svuota sempre lo stack** prima di accettare

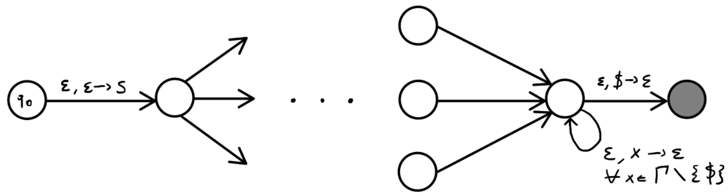


Figure 34: Implementazione della condizione 2

Il loop del penultimo stato **svuota lo stack**.

3. Per ogni transizione fa o **push** o **pop** di un simbolo sullo stack, ma mai **entrambe** le operazioni nella stessa transizione

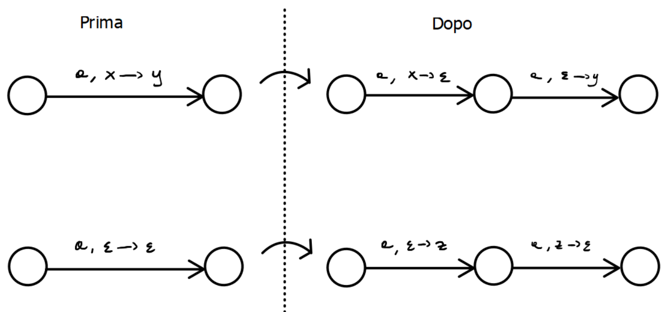


Figure 35: Implementazione della condizione 3

dove z è un **dummy character** in Γ

Per ogni coppia di stati p e q in P, la grammatica avrà una variabile A_{pq} che **genera tutte le stringhe** che possono portare P dallo stato p **partendo con lo stack vuoto** e arrivare in q con **lo stack vuoto**. Notiamo che le stesse stringhe possono portare P dallo stato q allo stato p lasciando lo stack inalterato.
Per progettare questa CFG G tale che A_{pq} **genera tutte le stringhe** che portano P da p a q finendo con lo **stack vuoto**, bisogna prima capire come P opera su tali stringhe. Per una qualsiasi stringa x, la prima mossa di P può essere solamente un **push**, in quanto ogni mossa può essere o un **pop** o un **push** e non si può fare un **pop** sullo stack vuoto. In modo analogo, l'ultima mossa su x sarà un **pop**.

Figure 36: Prima e ultima mossa di P che parte dallo stato p e raggiunge q con lo stack vuoto

Possono accadere due cose durante la computazione di x: il simbolo su cui è stato fatto **pop** all'inizio è lo stesso su cui è stato fatto **pop** alla fine o no.

- Se è **lo stesso**, lo stack può essere vuoto solamente all'**inizio** e alla **fine** della computazione di x.

  Questo caso viene simulato con la regola $A_{pq} \to aA_{rs}b$ dove:

  - a è l'input letto alla prima mossa
  - b è l'input letto all'ultima mossa
  - r è lo stato che si trova subito dopo lo stato p
  - s è lo stato che precede q

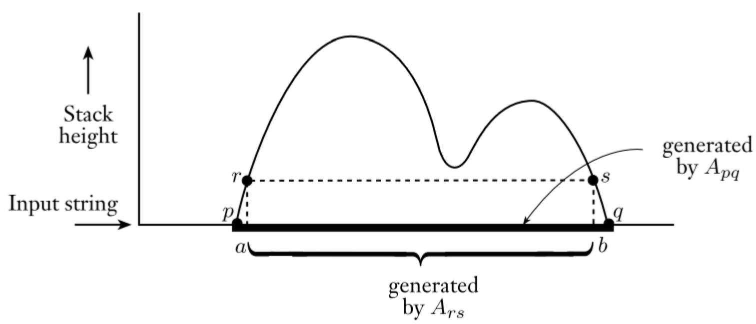


Figure 37: Computazione corrispondente alla regola $A_{pq} \to aA_{rs}b$

- Se **non è lo stesso**, il simbolo su cui è stato fatto **pop** all'inizio deve essere estratto con un **pop** prima della fine di x, rendendo lo stack vuoto in tale punto.

  Questo secondo caso viene simulato con la regola $A_{pq} \to A_{pr}A_{rq}$, dove r è lo stato in cui lo stack risulta essere vuoto.

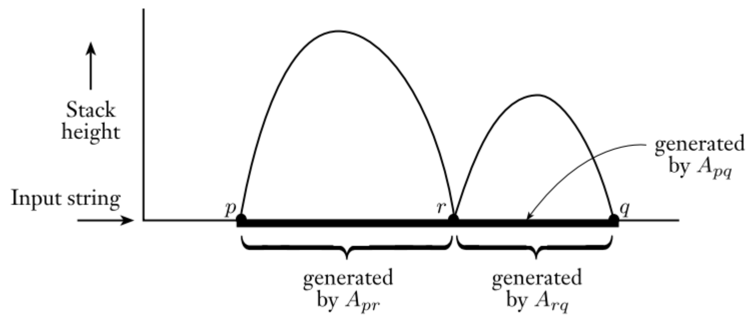


Figure 38: Computazione corrispondente alla regola $A_{pq} \to A_{pr}A_{rq}$

La **variabile iniziale** del PDA P sarà $S = A_{q_0,q_{\text{accept}}}$.

La dimostrazione formale non è stata fornita a lezione.

6.3 Pumping lemma per linguaggi acontestuali

Non tutti i linguaggi sono acontestuali. Questa tecnica ci aiuta a dimostrare che un linguaggio non è acontestuale.
Questa tecnica chiamata **pumping lemma per linguaggi acontestuali** dice che esiste un certo valore chiamato *pumping length* tale che tutte le stringhe di lunghezza maggiore di questo valore possono essere "pompate". A differenza dei linguaggi regolari, questa volta il significato di "pompare" una stringa è un po' più complesso: significa che la stringa può essere divisa in **cinque parti** tali che la **seconda** e la **quarta** parte possono essere **ripetute insieme** un qualsiasi numero di volte e la stringa risultate rimane comunque nel linguaggio.

6.3.1 Teorema - Pumping lemma per linguaggi acontestuali

Se A è un CFL allora esiste un valore p (*pumping length*) dove se $s \in A$ e $|s| \geq p$, allora s può essere divisa in cinque parti $s = uvxyz$ tali che:

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

Quando s viene spezzata, la condizione 2 impone che $v \neq \epsilon$ oppure $y \neq \epsilon$. La condizione 3 dice che i pezzi v, x e y insieme devono essere almeno di lunghezza p. Questa condizione a volte risulta essere utile per alcuni tipi di linguaggi.

6.3.1.1 Esempio pumping lemma per per CFL

Mostrare che $L = \{a^nb^nc^n : n \geq 0\}$ non è acontestuale.

Supponiamo che lo sia. Questo significa che esiste un valore p tale che soddisfa il **pumping lemma per linguaggi acontestuali**. Prendo la stringa $s = a^p b^p c^p$, con $|s| = 3p > p$ e considero i modi di scomporre s in $s = uvxyz$:

1. v e y contengono un solo tipo di simbolo:

   $$a\underbrace{aa}_{v}abbbbc\underbrace{c}_{y}cc$$

   oppure

   $$aaaa\underbrace{bbbb}_{v}cccc \qquad y = \epsilon$$

   Prendiamo $i = 2$, avremo uv^2xy^2z della seguente forma:

   $$a\underbrace{aaaa}_{v^2}abbbbc\underbrace{cc}_{y^2}cc$$

   Risulta che il numero di a e di c è maggiore al numero di b, quindi non è acontestuale.

2. v oppure y hanno almeno due simboli:

   $$a\underbrace{aab}_{v}\underbrace{b}_{y}bccc$$

   Prendiamo $i = 2$, avremo uv^2xy^2z della seguente forma:

   $$aa\underbrace{abaab}_{v^2}\underbrace{bb}_{y^2}bccc$$

   Risulta che il numero di a, b e c possono essere uguali, ma le lettere non sono nell'ordine corretto, quindi $s \notin L$ e quindi non è acontestuale.

6.4 Chiusura per le CFG

6.4.1 Chiusura rispetto a $\cup$

Come visto nella tecnica 5.1.1.1 (pag. 33), le CFG sono chiuse rispetto all'unione.

6.4.2 Chiusura rispetto a $\cap$

Le CFG **non sono chiuse** rispetto all'intersezione. Vediamo perché: sappiamo che $L = \{a^n b^n c^n : n \geq 0\}$ non è acontestuale. Prendiamo in considerazione i seguenti due linguaggi:

- $L_1 = \{a^n b^n c^i : n \geq 0, i \geq 0\}$
- $L_2 = \{a^i b^n c^n : n \geq 0, i \geq 0\}$

Come si può facilmente vedere, L_1 e L_2 sono acontestuali:

- CFG per L_1:

$$
\begin{aligned}
S &\to TU \\
T &\to aTb \mid \epsilon \\
U &\to cU \mid \epsilon
\end{aligned}
$$

- CFG per L_2:

$$
\begin{aligned}
S &\to UT \\
T &\to bTc \mid \epsilon \\
U &\to aU \mid \epsilon
\end{aligned}
$$

Ma $L_1 \cap L_2 = L$, che abbiamo detto essere non acontestuale, quindi le CFG non sono chiuse rispetto all'intersezione.

6.4.3 Chiusura rispetto al complemento

Le CFG **non sono chiuse** rispetto al complemento. Questo è una conseguenza di non essere chiuse rispetto all'intersezione. Siano A, B due CFG:

$$\overline{\overline{A \cap B}} = \overline{\overline{A} \cup \overline{B}} = A \cap B$$

Ma siccome le CFG non sono chiuse rispetto all'intersezione allora non possono esserlo nemmeno rispetto al complemento.

7 Macchine di Turing - TM

Abbiamo visto che i DFA sono dei piccoli automi senza memoria che sono equivalenti ai **linguaggi regolari** e i PDA sono degli automi con memoria infinita accessibile solamente in maniera LIFO che sono equivalenti alle **grammatiche acontestuali**.

Vediamo ora le **macchine di Turing** (TM), un modello di computazione molto più potente proposto da *Alan Turing* nel 1936. Una macchina di Turing può fare tutto ciò che un vero computer può fare, tuttavia alcuni problemi non possono essere risolti nemmeno da una macchina di Turing.

Una TM ha un **nastro infinito** come memoria illimitata, ha una **testina** che viene usate per **leggere** e **scrivere** simboli e **continua la propria computazione** finché non decide di produrre un output. Gli output sono **accept** e **reject** e vengono ottenuti entrando in degli stati appositi. Se nessuno di questi stati viene raggiunto, la macchina continua a **computare all'infinito senza mai fermarsi**.
<u>Differenze con un automa a stati finiti</u>:

- Una TM può sia **leggere** che **scrivere** sul nastro

- La testina può muoversi sia a **destra** che a **sinistra** sul nastro
- Il nastro è **infinito**

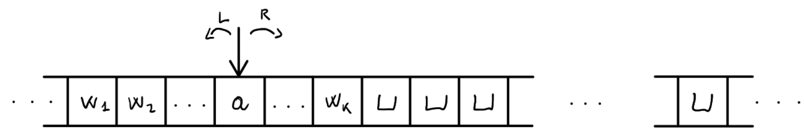


Figure 39: Schema del nastro della TM

- Gli stati speciali per accettare o rifiutare hanno un **effetto immediato**

7.1 Definizione - Macchina di Turing

Una **TM** è una settupla

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

dove Q, Σ, Γ sono tutti insiemi finiti e:

- Q è l'insieme degli **stati**
- Σ è l'**alfabeto di input** che non contiene il simbolo $\sqcup$ (simbolo **blank**)
- Γ è l'**alfabeto di nastro**, dove $\sqcup \in \Gamma,\ \Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ è la **funzione di transizione**
- q_0 è lo **stato iniziale**
- q_{accept} è lo **stato di accettazione**
- q_{reject} è lo **stato di rifiuto**, dove $q_{\text{reject}} \neq q_{\text{accept}}$

Computazione di una TM:

- La TM parte nello stato q_0 e riceve l'input $w = w_1 w_2 \dots w_n \in \Sigma^*$ nei primi n blocchi del nastro di input, il resto sarà popolato dal simbolo $\sqcup$. In quanto Σ non contiene $\sqcup$, la prima volta che si incontra questo simbolo denota la fine dell'input
- La computazione segue la δ. La testina non muove a sinistra se si trova all'estremità sinistra del nastro anche se la funzione di transizione indica L.
- La computazione termina una volta raggiunti q_{accept} o q_{reject}. Se ciò non accade mai, la TM va in loop senza mai fermarsi

Mentre la TM computa, ci sono dei cambiamenti dello stato corrente, del contenuto del nastro e della posizione della testina. Uno **snapshot** di questi tre elementi è una **configurazione** della macchina che viene rappresentata in un modo speciale: per uno stato q e due stringhe $u, v \in \Gamma$, si scrive $u\ q\ v$ per la configurazione in cui q è lo stato attuale, il contenuto del nastro è uv e la testina si trova sul primo simbolo di v. Ad esempio:

$$1011q_7 01111$$

rappresenta la configurazione in cui sul nastro è presente la stinga 101101111, lo stato attuale è q_7 e la testina si trova sul secondo 0 della stringa.

Più in generale, diciamo che una configurazione C_1 **produce** una configurazione C_2 se la TM può andare **legalmente** da C_1 a C_2 in **un singolo step**. Supponiamo di avere $a, b, c \in \Gamma,\ u, v \in \Gamma^*$ e due stati q_i, q_j. Supponiamo di avere due configurazioni uaq_ibv e aq_jacv. Diciamo che

$$uaq_ibv \textbf{ produce } uq_jacv$$

se la funzione di transizione $\delta(q_i, b) = (q_j, c, L)$, ovvero se nello stato q_i legge b, lo rimpiazza con c, muove la testina a sinistra e passa nello stato q_j. Altrimenti diciamo

$$uaq_ibv \textbf{ produce } uacq_jv$$

se $\delta(q_i, b) = (q_j, c, R)$.
Una macchina di Turing M accetta $w \in \Sigma^*$ se esiste una sequenza di configurazioni $C_1, C_2, \ldots C_k$ dove:

- $C_1 = q_0w$ è la configurazione iniziale di M su input w
- Ogni C_i produce C_{i+1}
- C_k è una configurazione accettante

La collezione di stringhe accettate da M è il **linguaggio di** M, denotato con $L(M)$.

7.1.1 Linguaggi Turing riconoscibili

Un linguaggio è detto **Turing riconoscibile** se esiste una TM che lo riconosce.

7.1.2 TM decisore

Una TM M è un **decisore** se non va in loop. Diciamo che una TM **decide** un linguaggio L se M è un decisore e riconosce L.

7.1.3 Linguaggi Turing decidibili

Un linguaggio è **Turing decidibile** se esiste una TM che lo decide. Tutti i linguaggi decidibili sono anche Turing riconoscibili.

7.1.3.1 Esempio di TM

Si consideri il linguaggio $L = \{0^n 1^n : n \geq 0\}$. La seguente TM M_1 riconosce tale linguaggio:
Pseudocodice per M_1 su input w:

1. Se viene letto il carattere 0, scrivi x e muovi a destra finché non viene incontrato il prossimo 1
2. Se viene letto il carattere 1, scrivi y e muovi a sinistra finché non viene incontrato il prossimo 0
3. Se al primo passo verso destra viene letto y, continua a scorrere verso destra e se vengono letti solo y fino alla fine della stringa **accetta**, altrimenti se viene letto anche solo un 1 **rifiuta**.

$$\begin{gathered} 00001111\sqcup \\ x000y111\sqcup \\ xx00yy11\sqcup \\ \vdots \\ xxxxyyyy\sqcup \end{gathered}$$

Descrizione formale di M_1:

- $Q = \{q_0, q_1, q_2, q_3, q_{\text{acc}}\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{x, y, \sqcup\}$
- δ definita nella figura 40
- q_0 è lo stato iniziale
- q_{acc} è lo stato di accettazione
- Lo stato q_{reject} manca in quanto se un ramo di computazione non è definito nella δ, allora la macchina rifiuta

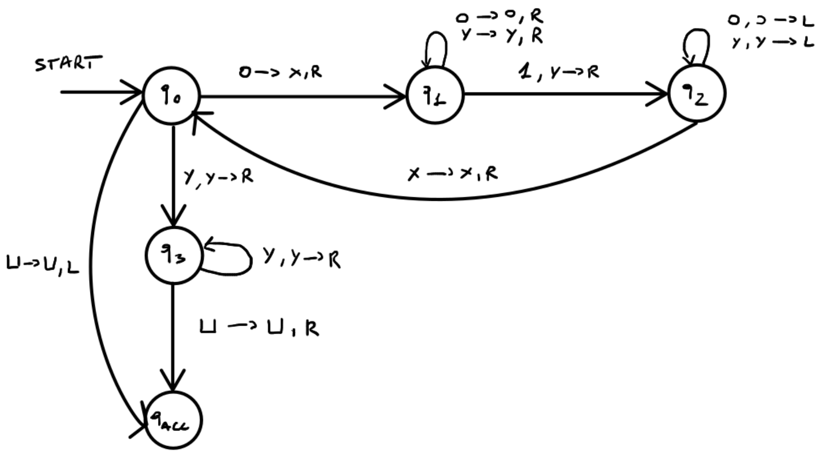


Figure 40: Diagramma degli stati della TM

7.1.4 Consigli mentre si crea un TM

7.1.4.1 Riconoscere la fine del nastro Viene aggiunto un nuovo carattere in Γ, chiamiamolo \$, che viene messo in prima posizione del nastro e tutto il suo contenuto viene *shiftato* a destra di una posizione:

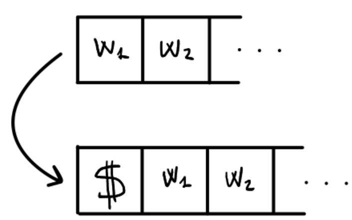


Figure 41: Inserimento del carattere \$ per riconoscere la fine del nastro

Una TM che sfrutta questa tecnica può essere ad esempio la seguente:

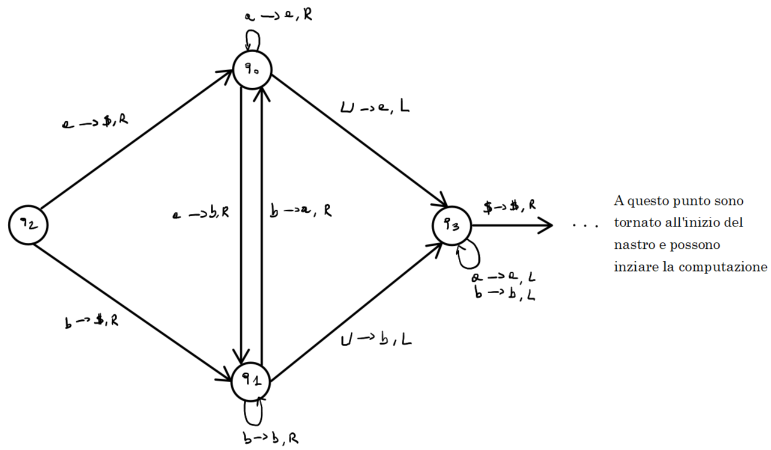


Figure 42: Implementazione di una macchina che riconosce la fine del nastro

7.1.4.2 Utilizzo di una TM come subroutine Si consideri il linguaggio $L = \{0^n 1^n 0^n : n \geq 0\}$ e supponiamo di avere una TM per che riconosce il linguaggio $L' = \{0^n 1^n : n \geq 0\}$ che lavora nel seguente modo:

$$000111000\sqcup$$
$$\vdots$$
$$xxxyyy000\sqcup$$

A questo punto viene rilanciata per controllare che dopo l'ultima x ci sia $y^n 0^n$.

7.1.4.3 Marcare elementi In Γ si aggiungono gli elementi che si vogliono marcare seguiti da un *: sia $\Gamma = \{a, b, c, \#\}$ e si vogliono marcare i simboli a, b e c. Quindi Γ diventerà $\Gamma = \Gamma \cup \{a^*, b^*, c^*\}$, ovvero $\Gamma = \{a, b, c, a^*, b^*, c^*, \#\}$.

7.1.4.4 Stay put A volte può capitare che dopo la lettura di un simbolo non si voglia muovere la testina.
Questo viene fatto modificando $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ nel seguente modo:

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

Dove S viene simulato con un movimento a destra seguito da un movimento a sinistra subito dopo (o viceversa) aggiungendo un nuovo stato r che sposta la testina nella posizione precedente:

$$\delta(q, a) = (r, b, \mathrm{R})$$
$$\delta(r, *) = (q_1, *, \mathrm{L})$$

7.2 Varianti di TM

7.2.1 Macchina di Turing multinastro

Una **Macchina di Turing multinastro** è come una TM normale ma ha diversi nastri, ognuno dei quali ha la **propria testina** per leggere e scrivere. Inizialmente l'input appare solamente sul primo nastro mentre tutti gli altri restano vuoti.

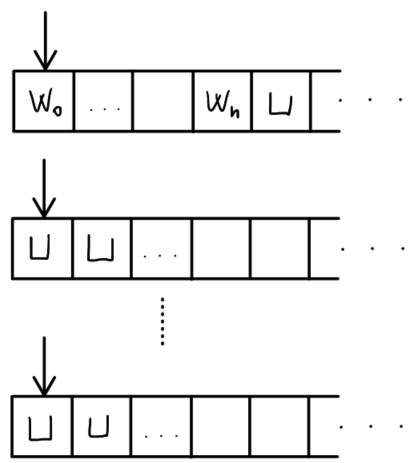


Figure 43: Macchina di Turing con più di un nastro

La **funzione di transizione** viene modificata per permettere la lettura, scrittura e movimento delle testine su tutti i nastri contemporaneamente:

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

dove k è il numero dei nastri. Un esempio di transizione è la seguente:

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \mathrm{L}, \mathrm{R}, \ldots, \mathrm{L})$$

A prima vista sembrerebbe che le TM multinastro siano **più potenti** di quelle a nastro singolo, ma in realtà hanno la stessa potenza. Ricordiamo che due macchine sono equivalenti se riconoscono lo stesso linguaggio.

7.2.1.1 Teorema - TM multinastro equivalente a TM singolo nastro

Per ogni TM M multinastro, esiste una TM M' a singolo nastro equivalente.

<u>Dimostrazione</u>: memorizzo tutte le informazioni necessarie ad eseguire M su un singolo nastro, andando a separare i k nastri con il simbolo "#":

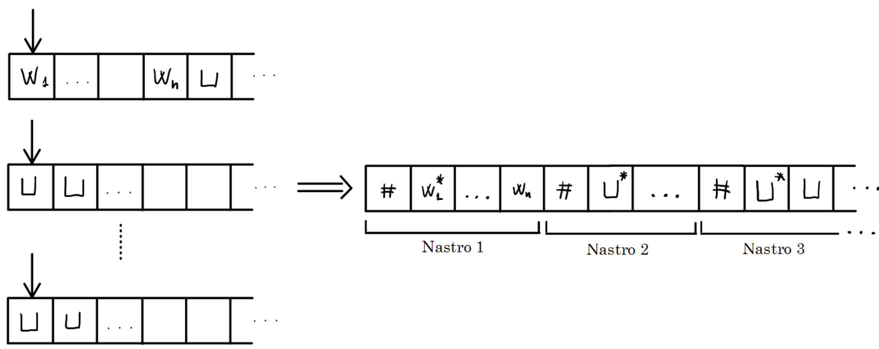


Figure 44: Trasformazione di una TM multinastro in una a singolo nastro

I **simboli marcati** sono quelli che corrispondono alle posizioni delle testine nei nastri multipli
Descrizione della TM M':

1. Mette il nastro in forma $\#w_1^* \dots w_n \# \sqcup^* \dots$
2. Simulo una singola mossa di M:
   (a) Scansiono il nastro dal primo $\#$ al $k-1$ esimo per determinare i simboli in lettura
   (b) Torno indietro al primo $\#$
   (c) Faccio un secondo passaggio aggiornando il contenuto del nastro e posizioni delle testine come dettato dalla δ
3. Se M deve spostare la testina su $\#$, scrivo un $\sqcup$ e traslo a destra il contenuto del nastro

7.2.1.2 Corollario Un linguaggio L è **Turing riconoscibile** se e solo se $\exists$ una TM multinastro che lo riconosce.

7.2.2 Macchina di Turing non deterministica - NTM

Una **Macchina di Turing non deterministica** (NTM) potrebbe procedere secondo diverse possibilità in un punto qualsiasi della computazione. La funzione di transizione sarà quindi la seguente:

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

Dove si può verificare la seguente situazione:

$$\delta(q, a) = (q', b, \mathrm{L})$$
$$\delta(q, a) = (q'', c, \mathrm{R})$$

La computazione di un NTM è descritta da un **albero**, dove ogni nodo contiene una **configurazione**:

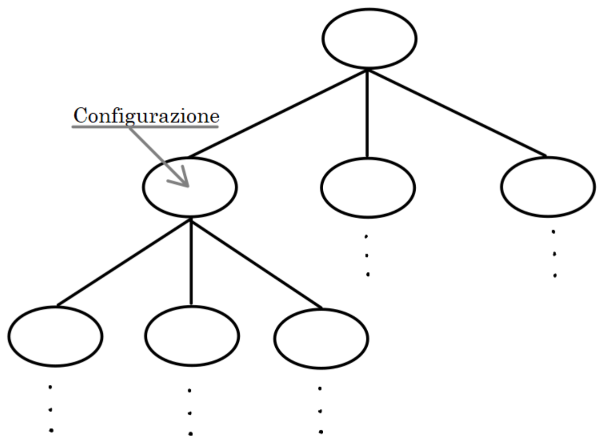


Figure 45: Albero di computazione di una macchina di Turing non deterministica

La NTM accetta se uno qualsiasi dei rami di computazione accetta.

7.2.2.1 Teorema - NTM equivalente a TM

Per ogni NTM N, esiste una TM M deterministica equivalente.

Idea: La TM M esplora tutti i cammini di computazione di N alla ricerca di un cammino accettante: se lo trova **accetta**, altrimenti la simulazione non termina. L'albero viene esplorato tramite l'utilizzo di una *breadth-first search*, in modo tale da eliminare la possibilità di entrare in rami di computazione infiniti che non accetteranno mai.
Dimostrazione: Definisco una TM a 3 nastri:

- Il **primo nastro** contiene l'**input**
- Il **secondo nastro** è il **nastro di lavoro**
- Il **terzo nastro** contiene l'**indirizzo**, che sarebbe il **ramo parziale di computazione** che deve essere esplorato.

Un indirizzo corrisponde al **cammino** che devo prendere per arrivare a tale nodo partendo dalla radice. Prendiamo per esempio il seguente albero degli indirizzi:

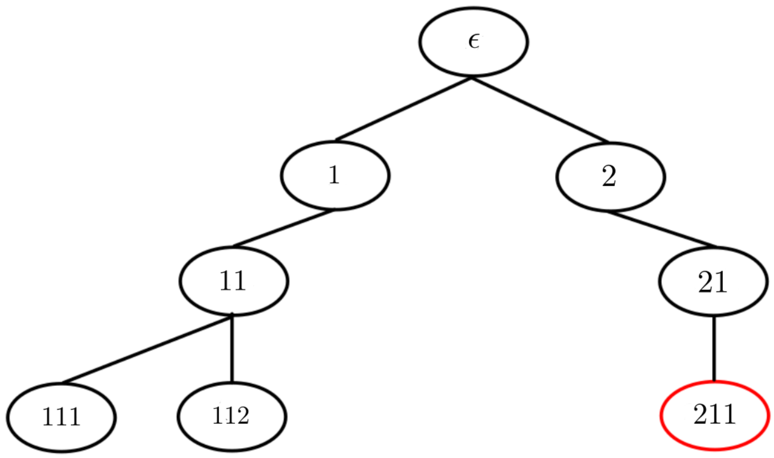


Figure 46: Esempio di albero degli indirizzi di una NTM

Per raggiungere il nodo cerchiato di rosso (in basso a destra) con indirizzo 211, devo prendere il **secondo** (2) figlio della radice, passare per il **primo** e unico figlio del secondo figlio della radice (21) ed infine prendere il **primo** e unico figlio del nodo con indirizzo 21, ovvero il nodo 211. Denotiamo con b il **numero massimo di figli** che un nodo può avere e viene dato dalla funzione di transizione.
La TM M procede nel seguente modo:

1. Il **primo nastro** contiene l'input w, mentre il **secondo** è vuoto ed il **terzo** contiene la stringa vuota ϵ
2. Copio il **primo nastro** sul **secondo**
3. Eseguo la subroutine **Step(N,w,i)** sul **secondo nastro**.

   Step(N,w,i) simula l'esecuzione di N sul **secondo nastro** usando il cammino dalla radice al nodo i. Se la subroutine accetta vuol dire che N ha accettato e quindi anche M **accetta**. Se la subroutine rifiuta vuol dire che N ha rifiutato. In questo caso si passa direttamente allo step 4.
4. Calcolo il prossimo indirizzo secondo la *visita in ampiezza* e lo scrivo sul **terzo nastro**. Poi torno al passo 2.

7.2.2.2 Corollario Un linguaggio è **Turing riconoscibile** se e solo se un macchina di Turing non deterministica lo riconosce.

7.2.3 Enumeratori

Un **enumeratore** è una macchina di Turing con una **stampante** collegata. Questa stampante stampa solamente stringhe in output generando un linguaggio.

Un enumeratore parte con il suo nastro vuoto. Se non si ferma mai, potrebbe stampare una lista infinita di stringhe. L'ordine delle stringhe è arbitrario.
Nota: I linguaggi Turing riconoscibili vengono detti anche **ricorsivamente enumerabili**.

7.2.3.1 Teorema - Enumeratori e linguaggi Turing riconoscibili

Un linguaggio è Turing riconoscibile se e solo se $\exists$ un enumeratore che lo enumera.

Dimostrazione:

- $\Leftarrow$ Supponiamo esista un enumeratore E che enumera il linguaggio L. Allora c'è una TM M che riconosce L.

  La TM su input w simula E e ogni volta che E produce un output controlla se questo è uguale a w. Se lo è, **accetta**.

  È chiaro che M accetta tutte le stringhe che appaiono nella lista di E.

- $\Rightarrow$ Se una TM M riconosce il linguaggio L, allora possiamo costruire il seguente enumeratore E per A. Siano $s_1, s_2, s_3, \dots$ la lista di tutte le possibili stringhe in Σ^*.

  Pseudocodice di E:

  for $i = 1, 2, 3, \dots$ (loop infinito)
    for $j = 1$ to i
      Simula M usando s_j come input per i passi
      Se M accetta s_j (entro i passi)
        Stampa s_j

  Se M accetta una particolare stringa s, allora prima o poi apparirà nella lista generata da E. In realtà apparirà un numero infinito di volte in quanto ogni volta M riparte dall'inizio per ogni stringa per ogni ripetizione del primo for. $\square$

8 Decidibilità

Con la **decidibilità** si studia il potere ed i limiti che hanno gli algoritmi. Inoltre verranno studiati dei problemi che sono risolvibili mediante un algoritmo ed altri che non possono essere risolti.

Esempio: Il decimo problema di *Hilbert*. Il polinomio

$$p(x, y, z) = 6x^3yz^2 + 3xy^2 - x^3 - 10$$

ha **radici intere**? Sì: $x = 5, y = 3, z = 0$.

Si cerca il *processo in base al quale il problema è risolto in un* ***numero finito di passi***.
Riscriviamo il decimo problema di Hilbert con la notazione che verrà usata per i prossimi esempi:

$$D = \{\langle p \rangle : p \text{ polinomio con radici intere}\}$$

Dove $\langle p \rangle$ indica la **codifica binaria di** p. Il decimo problema di Hilbert essenzialmente chiede se D è **decidibile**. La risposta è **no**. Però è **Turing riconoscibile**.

Nota: Le macchine di Turing sono equivalenti agli *algoritmi*.

$$\textit{Macchina di Turing} \equiv \textit{Algoritmo}$$

Quindi d'ora in poi non ci soffermeremo più sull'implementazione a basso livello della macchina ma verranno fornite delle **descrizioni** di ciò che la macchina dovrà fare.

8.1 Codifica dell'input di un TM

L'input di una TM è **sempre una stringa**. Se vogliamo fornire in input qualcosa di diverso da una stringa, deve essere prima **codificato**. In generale, dato un oggetto O (che può essere un *grafo*, TM, DFA, NFA,...) indico con $\langle O \rangle$ la sua **codifica binaria**. Se si hanno molteplici oggetti $O_1, O_2, \ldots, O_k$, denotiamo la loro codifica in un'unica stringa $\langle O_1, O_2, \ldots, O_k \rangle$. La codifica stessa può essere fatta in tanti modi ragionevoli, ma a noi non interessa quale viene scelta in quanto una TM può **sempre tradurre una codifica in un'altra**.

8.2 Problemi di decidibilità riguardanti linguaggi regolari

8.2.1 Problema dell'accettazione per DFA

Questo problema si concentra sul verificare se un DFA accetti o meno una data stringa. Questo problema può essere espresso tramite il seguente linguaggio:

$$A_{\text{DFA}} = \{\langle B, w \rangle : B \text{ è un DFA che accetta la stringa in input } w\}$$

dove:

- A sta per *"Acceptance"*
- B è la definizione di un DFA
- w è una stringa

Testare se un DFA B accetta un input w è equivalente al problema di testare se $\langle B, w \rangle$ è appartiene al linguaggio A_{DFA}.

8.2.1.1 Teorema - $A_{\mathbf{DFA}}$ è decidibile

A_{DFA} è **decidibile**.

Dimostrazione: definisco una TM M che decide A_{DFA}:

1. Su input $\langle B, w\rangle$ interpreta B come automa e w come stringa (ad. esempio $B = (Q, \Sigma, \delta, q_0, F)$ e la TM può avere un nastro per memorizzare la δ, uno per l'input, uno per lo stato corrente, ecc.)

   Se non posso, **rifiuto**

2. Simulo B su input w

3. Se la simulazione termina in uno stato accettante, **accetto**. Altrimenti **rifiuto**.

8.2.2 Problema dell'accettazione per NFA

Come prima, ma stavolta ci occupiamo di NFA:

$$A_{\text{NFA}} = \{\langle B, w\rangle : B \text{ è un NFA che accetta la stringa in input } w\}$$

8.2.2.1 Teorema - $A_{\mathbf{NFA}}$ è decidibile

A_{NFA} è **decidibile**.

Dimostrazione: creiamo una TM N che decide A_{NFA}. Si potrebbe creare N che operi come M, simulando un NFA piuttosto che un DFA. Invece, per illustrare una nuova idea, facciamo che N usa M come subroutine:

1. Converto l'NFA B nel DFA equivalente B'

2. Eseguo M del teorema 8.2.1.1 su input $\langle B', w\rangle$

3. Se M accetta, **accetto**. Altrimenti **rifiuto**

8.2.3 Problema dell'accettazione per REX

Determinare se un'espressione regolare genera una data stringa. Sia:

$$A_{\text{REX}} = \{\langle R, w\rangle : R \text{ è un espressione regolare che genera la stringa } w\}$$

8.2.3.1 Teorema - $A_{\mathbf{REX}}$ è decidibile

A_{REX} è **decidibile**.

Dimostrazione: creo una TM R che decide A_{REX} nel seguente modo:

1. Converto l'espressione regolare R nell'NFA B'' equivalente

2. Eseguo N del teorema 8.2.3.1 si input $\langle B'', w\rangle$

3. Se N accetta, **accetto**. Altrimenti **rifiuto**

8.2.4 Esercizio - PATH

Dimostrare che

$$\text{PATH} = \{\langle G, s, t\rangle : G \text{ è un grafo tale che esiste un cammino } s \rightsquigarrow t\}$$

è **decidibile**.
Definisco una TM che decide PATH:

1. Interpreto $\langle G, s, t\rangle$ come $G = (V, E)$ e s, t nodi in V.
   Se l'input non è di questo formato, **rifiuto**.
2. Marco la sorgente s
3. Marco i nodi che hanno degli archi provenienti da un nodo marcato
4. Se t è marcato, **accetto**. Altrimenti **rifiuto**

8.2.5 Test del vuoto

Il **test del vuoto** si occupa di verificare se un DFA accetta almeno una stringa.
Sia:

$$E_{\text{DFA}} = \{\langle A\rangle : A \text{ è un DFA e } L(A) = \emptyset\}$$

Dove E sta per *"Empty"*.
Un DFA accetta almeno una stringa se e solo se dallo stato iniziale si può **raggiungere** uno stato di accettazione.

8.2.5.1 Teorema - E_{DFA} è decidibile

E_{DFA} è **decidibile**.

Dimostrazione: Progettiamo una TM M che utilizza l'**algoritmo di marcatura** visto nell'esercizio 8.2.4:

1. Su input $\langle A\rangle$ dove A è un DFA
2. Marco lo stato iniziale di A
3. Ripeti finché non viene marcato nessun nuovo stato:
4.   Marco i nodi che hanno degli archi provenienti da un nodo marcato
5. Se lo stato di accettazione è marcato allora **rifiuto**, altrimenti **accetto**.

Nota: i teoremi 8.2.1.1, 8.2.2.1, 8.2.3.1 illustrano che, per criteri di decidibilità, è equivalente usare un DFA, NFA o espressione regolare in quanto la macchina di Turing può benissimo passare da una codifica ad un'altra.

8.2.6 Test di eguaglianza di linguaggi

Il prossimo teorema afferma che determinare se due DFA riconoscono esattamente lo stesso linguaggio è decidibile. Sia:

$$EQ_{\text{DFA}} = \{\langle A, B\rangle : A, B \text{ sono DFA e } L(A) = L(B)\}$$

8.2.6.1 Teorema - $EQ_{\textbf{DFA}}$ è decidibile

EQ_{DFA} è **decidibile**.

<u>Dimostrazione</u>: costruiamo un nuovo DFA C derivante da A e B, dove C accetta solamente quelle stringhe che vengono accettate o da A o da B, ma mai tutti e due insieme. Quindi se A e B riconoscono lo stesso linguaggio, C non riconoscerà nulla. Il linguaggio di C è il seguente:

$$L(C) = L(A)\triangle L(B) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

Dove $\triangle$ è la ***differenza simmetrica***.

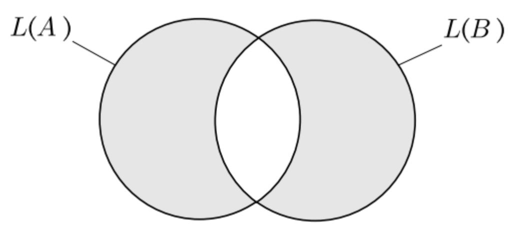


Figure 47: In grigio chiaro, la differenza simmetrica di $L(A)$ e $L(B)$ che da vita a $L(C)$

Osservazioni:

1. Dalle proprietà di **chiusura dei linguaggi regolari** rispetto a **unione**, **intersezione** e **complemento** posso costruire il DFA C tale che

$$L(C) = L(A)\triangle L(B)$$

2. La ***differenza simmetrica*** risulta essere comoda in quanto

$$L(C) = \emptyset \Leftrightarrow L(A) = L(B)$$

Dalle queste due osservazioni, notiamo immediatamente che possiamo costruire una TM che decide EQ_{DFA} usando la TM M del **test del vuoto** (teorema 8.2.5.1).
Pseudocodice:

1. Su input $\langle A, B\rangle$ dove A e B sono DFA

2. Costruisco C che accetta $L(C) = L(A) \triangle L(B)$, come visto nella prima osservazione
3. Eseguo la TM M del test del vuoto su input $\langle C \rangle$
4. Se M accetta, **accetto**, altrimenti **rifiuto**

8.3 Problemi di decidibilità su linguaggi acontestuali

8.3.1 Problema della generazione di una stringa in una CFG

Il problema della **generazione di una stringa in una CFG** è quello di capire se una data CFG genera una data stringa. Sia:

$$A_{\text{CFG}} = \{\langle G, w \rangle : G \text{ è una CFG che genera la stringa } w\}$$

8.3.1.1 Teorema - A_{CFG} è decidibile

A_{CFG} è **decidibile**.

Idea: la prima idea (errata) sarebbe quella di scorrere tutte le derivazioni e vedere se una di esse corrisponde alla stringa w. Questo non può funzionare in quanto ci potrebbero essere **infinite derivazioni** e, se G non genera w, la TM non terminerebbe mai e quindi non sarebbe un **decisore** ma un **riconoscitore**. Per far si che la TM sia un **decisore**, possiamo usare il fatto che se G è in **forma normale di Chomsky**, ogni derivazione di w ha $2n - 1$ passi, dove $n = |w|$. Questo significa che se dopo aver simulato $2n - 1$ passi di derivazione in G non abbiamo incontrato nessuna derivazione di w, allora M rifiuta.
Dimostrazione: costruiamo la TM M per decidere A_{CFG} come segue:

1. Su input $\langle G, w \rangle$ dove G è un CFG e w è un stringa
2. Converto G in una grammatica equivalente in forma normale di Chomsky
3. Listo tutte le derivazioni di $2n - 1$ passi dove $n = |w|$, altrimenti di un passo se $n = 0$
4. Se una qualsiasi di queste derivazioni genera w **accetto**, altrimenti **rifiuto**

8.3.2 Test del vuoto per CFG

Come per i DFA, possiamo dimostrare che il problema di **test del vuoto per CFG**, ovvero determinare se un grammatica genera almeno una stringa, è decidibile. Sia:

$$E_{\text{CFG}} = \{\langle G \rangle : G \text{ è una CFG e } L(G) = \emptyset\}$$

8.3.2.1 Teorema - E_{CFG} è decidibile

E_{CFG} è **decidibile**.

Idea: la prima idea (errata) è quella di usare la TM usata nel teorema 8.3.1.1 per **generare** una ad una **tutte le possibili** w e verificare che **non vengano accettate**. Questo però non funziona in quanto le stringhe potrebbero essere **infinite** e quindi la TM potrebbe **non terminare**.
Per determinare se il linguaggio di una grammatica è vuoto, bisogna controllare se la **variabile iniziale** sia in grado di generare una stringa di **soli terminali**. Quello che si va a fare è: per ogni variabile, determina se genera una stringa di soli terminali. Se la genera, marca tale variabile e prosegue finché non ci sono più variabili da marcare. Se la variabile iniziale non è marcata accetta, altrimenti rifiuta.
Dimostrazione: costruiamo la TM M che riconosce E_{CFG} come segue:

1. Su input $\langle G \rangle$ dove G è un CFG
2. Marco tutti i simboli terminali di G
3. Ripeto finché nessuna nuova variabile viene marcata:
4. Marco tutte le variabili A tale che in G c'è una regola $A \to U_1, \ldots, U_k$ e tutti i simboli $U_1, \ldots, U_k$ sono già marcati
5. Se la variabili iniziale non è marcata **accetto**, altrimenti **rifiuto**

8.3.3 Test di eguaglianza di CFG

Consideriamo il problema di determinare se due CFG generano lo stesso linguaggio. Sia:

$$EQ_{\text{CFG}} = \{\langle G, H \rangle : G, H \text{ sono CFG e } L(G) = L(H)\}$$

Nota: non possiamo usare la stessa tecnica utilizzata per EQ_{DFA} (teorema 8.2.6.1) in quanto la classe delle CFG **non è chiusa** rispetto alle operazioni di **intersezione** e **complemento**.

Dalla nota si conclude che il test di eguaglianza di CFG **non è decidibile**. Vedremo come dimostrare la non decidibilità a breve.

8.3.4 Teorema - Ogni linguaggio acontestuale è decidibile

Ogni linguaggio acontestuale (CFL) è decidibile.

Dimostrazione: sia A un CFL, vogliamo far vedere che A è decidibile. Siccome A è un CFL, esiste una CFG G che genera A. Usiamo la TM M che decide A_{CFG} (teorema 8.3.1.1) per decidere A.
Costruiamo la TM M_G che riconosce A. Creiamo una copia di G su M_G. Proseguiamo come segue:

1. Su input w
2. Eseguo TM M su input $\langle G, w\rangle$
3. Se la macchina accetta **accetto**, altrimenti **rifiuto**

8.3.5 Relazioni tra classi di linguaggi

Sia DEC la classe dei linguaggi **decidibili**, ovvero decisi da TM e sia T-RIC la classe dei linguaggi **Turing riconoscibili**. Ci sono linguaggi **non decidibili** che sono **Turing riconoscibili** e anche linguaggi che sono **non Turing riconoscibili**. Quindi la relazione tra classi di linguaggi è la seguente:

$$\text{REG} \subseteq \text{CFL} \subseteq \text{DEC} \subseteq \text{T-RIC}$$

9 Indecidibilità

La **indecidibilità** studia i problemi **algoritmicamente irrisolvibili**.

9.1 Problema dell'accettazione di una TM

Vediamo il **problema dell'accettazione di una TM**: data una TM, vogliamo decidere se accetta o meno una data stringa. Sia:

$$A_{\text{TM}} = \{\langle M, w\rangle : M \text{ è una TM e accetta } w\}$$

Prima di arrivare alla dimostrazione, osserviamo che A_{TM} è **Turing riconoscibile**. Questo significa che i **riconoscitori** sono più potenti dei **decisori**. Richiedere che la macchina non vada in *loop* restringe i linguaggi riconosciuti.
La seguente TM U riconosce A_{TM}:

1. Su input $\langle M, w\rangle$ dove M è una TM e w è una stringa
2. Simulo M su input w
3. Se M entra nel suo stato di accettazione **accetto**, se M entra nel suo stato di rifiuto **rifiuto**.

Notiamo che U va in loop su input $\langle M, w\rangle$ se M va in loop su w, che è proprio il motivo per cui **non decide** A_{TM}. La TM U è un esempio di **macchina di Turing universale**, così chiamata in quanto è in grado di simulare una qualsiasi altra macchina di Turing partendo dalla sua descrizione. Vediamo ora la tecnica per dimostrare che un problema è indecidibile chiamata **diagonalizzazione**.

9.1.1 Diagonalizzazione

Tecnica scoperta da *Cantor* per comparare la dimensione di insiemi infiniti. Verrà da noi utilizzata per dimostrare che due insiemi infiniti non hanno la stessa cardinalità.
Siano A e B due insiemi e $f : A \to B$ una funzione:

- f è **iniettiva** se non mappa mai due elementi diversi nello stesso elemento, quindi deve necessariamente essere $f(a) \neq f(b)$ ogni volta che $a \neq b$
- f è **suriettiva** se tocca ogni elemento di B. Ovvero se

$$\forall b \in B \ \exists a \in A : f(a) = b$$

- f è **biiettiva** se è sia **inieittiva** che **suriettiva**

A e B hanno la **stessa cardinalità** se esiste una **funzione biiettiva** da A a B che è **totale** su A, ovvero il cui dominio è tutto A.

9.1.1.1 Esempio 1

Siano $\mathbb{N}$ l'insieme dei numeri naturali e $\mathbb{E}$ l'insieme dei numeri naturali pari. $\mathbb{N}$ e $\mathbb{E}$ hanno la stessa cardinalità:

n	$f(n)$
0	0
1	2
2	4
3	6
$\vdots$	$\vdots$

La tabella è la funzione **biiettiva** e **totale** su $\mathbb{N}$

$$f : \mathbb{N} \to \mathbb{E}$$

definita da $f(n) = 2n$.

$\mathbb{E}$ viene detto **numerabile** in quanto ha la stessa cardinalità di $\mathbb{N}$. Tutti gli insiemi finiti o con questa caratteristica prendono tale nome.

9.1.1.2 La diagonalizzazione

Il matematico *Cantor* introdusse la diagonalizzazione per mostrare che l'insieme dei numeri reali $\mathbb{R}$ non è numerabile.

Per dimostrarlo ci limiteremo all'intervallo $[0, 1]$ di $\mathbb{R}$. La dimostrazione è per **contraddizione**: supponiamo per assurdo che esista una biiezione f tra $\mathbb{N}$ e $[0, 1]$. Mostreremo poi che esiste un elemento $d \in [0, 1]$ che non è accoppiato tramite f con nessun elemento di $\mathbb{N}$, il che è impossibile e quindi la biiezione non esiste. Tale elemento d è l'***elemento diagonale***.

Dimostrazione: supponiamo che esista la biiezione $f : \mathbb{N} \to [0, 1]$.
Sia ad esempio:

$$
\begin{aligned}
f(1) &= 0.\underline{5}105110\ldots \\
f(2) &= 0.4\underline{1}32043\ldots \\
f(3) &= 0.82\underline{4}5026\ldots \\
f(4) &= 0.233\underline{0}126\ldots \\
f(5) &= 0.4107\underline{2}46\ldots \\
f(6) &= 0.99378\underline{3}8\ldots \\
f(7) &= 0.010513\underline{5}\ldots \\
&\vdots
\end{aligned}
$$

Costruiamo $d \in [0, 1]$ in modo tale che non esiste alcun $n \in \mathbb{N}$ tale che $f(n) = d$, ovvero faremo in modo tale che l'i-esima cifra decimale di d è diversa dall'i-esima cifra decimale di $f(i)$. Un esempio potrebbe essere $d = 0.437 \cdots \in [0, 1]$ in quanto:

- $4 \neq 5$ che è la **prima** cifra decimale di $f(1)$
- $3 \neq 1$ che è la **seconda** cifra decimale di $f(2)$
- $7 \neq 4$ che è la **terza** cifra decimale di $f(3)$
- ...

Siccome f è una biiezione $f : \mathbb{N} \to [0, 1]$ e $d \in [0, 1]$, deve esistere un elemento $n \in \mathbb{N}$ tale che $f(n) = d$. Ma per **costruzione** di d, la n-esima cifra decimale di d deve essere diversa dalla n-esima cifra decimale di $f(n)$. Ma $f(n)$ però è proprio d, quindi stiamo chiedendo che l'n-esima cifra decimale di d sia **diversa da se stessa**, il che è **impossibile** (contraddizione). Questo significa che la biiezione non esiste e che $\mathbb{R}$ non è numerabile.

Questa tecnica ha un'applicazione estremamente importante per la teoria della computazione. Mostra che alcuni linguaggi sono indecidibili e anche non Turing riconoscibili in quanto c'è una **quantità non numerabile di linguaggi** e una quantità **finita di macchine di Turing**. Siccome una TM può riconoscere un solo linguaggio e ci sono più linguaggi che TM, questo implica che ci sono alcuni linguaggi che non vengono riconosciuti da nessuna TM, ovvero i linguaggi non Turing riconoscibili.

9.1.2 Teorema - Esistono linguaggi non Turing riconoscibili e non decidibili

Esistono linguaggi **non Turing riconoscibili** e **non decidibili**.

Dimostrazione: va dimostrato che esiste una quantità **numerabile** di macchine di Turing e una quantità **non numerabile** di linguaggi:

1. **La quantità di macchine di Turing è numerabile**: Consideriamo l'insieme $\mathcal{M} = \{\langle M \rangle : M \text{ è una TM}\}$. Ricordiamo che $\langle M \rangle$ è un **stringa**. Introduciamo un ordinamento, chiamiamolo $\prec$:

   (a) Ordinamento lessicografico $\prec$ su Σ che può essere esteso a Σ^* per parole di uguale lunghezza

   (b) Siano $x, y \in \Sigma^*$, avremo che

   $$x \prec y \text{ se e solo se } \begin{cases} |x| < |y| \\ |x| = |y| \quad x \prec y \end{cases}$$

   Si può vedere che $\prec$ è **totale** e **lineare**:

   $$x_1 \prec x_2 \prec \cdots \prec x_\alpha \prec \ldots$$

   Questo significa che esiste una $f : \mathbb{N} \to \Sigma^*$ biiettiva definita come

   $$f(i) = i\text{-esimo elemento in } \Sigma^*$$

   Siccome $\mathcal{M} \subseteq \Sigma^*$ e Σ^* abbiamo dimostrato essere numerabile, allora anche $\mathcal{M}$ è numerabile.

2. **La quantità di linguaggi non è numerabile**: sia $\mathcal{L} = \{L\}$ l'insieme di tutti i linguaggi sull'alfabeto Σ. Sia $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Creiamo la stringa χ_L nel seguente modo:

   $$\begin{array}{lcccccccccccc} \Sigma^* & = & \{ & \epsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \ldots & \} \\ L & = & \{ & -, & 0, & -, & 00, & 01, & -, & -, & 000, & -, & \ldots & \} \\ \chi_L & = & & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \ldots & \end{array}$$

   ovvero l'i-esimo valore di χ_L è 1 se la corrispondente stringa in Σ^* è presente anche il L, 0 altrimenti.

   Sia $\mathcal{B}$ l'insieme di tutte le stringhe di lunghezza infinita. Questo insieme non è numerabile (dimostrabile tramite la diagonalizzazione).
   Esiste $f : \mathcal{L} \to \mathcal{B}$ definita come $f(L) = \chi_L$ biiettiva. Siccome $\mathcal{B}$ non è numerale, ne deduciamo che anche $\mathcal{L}$ non lo sia. $\square$

Siccome abbiamo dimostrato che la quantità di linguaggi è non numerabile, mentre la quantità di TM è numerabile, non è possibile mettere in corrispondenza le TM con i linguaggi e quindi ci devono essere per forza linguaggi non Turing riconoscibili.

9.1.3 Teorema - A_{TM} è indecidibile

A_{TM} è **indecidibile**.

$$A_{\text{TM}} = \{\langle M, w \rangle : M \text{ è una TM che accetta } w\}$$

Dimostrazione: per assurdo assumiamo che A_{TM} sia decidibile. Sia H il suo decisore:

$$H(\langle M, w\rangle) = \begin{cases} \textit{accetta} & \text{se } M \text{ accetta } w \\ \textit{rifiuta} & \text{se } M \text{ rifiuta } w \end{cases}$$

Costruiamo ora la TM D ("Diagonale") che usa H come subroutine. Questa nuova TM chiama H per determinare cosa fa M quando riceve come input la sua stessa descrizione $\langle M\rangle$. Quando D lo ha determinato, fa l'opposto:

1. Su input $\langle M\rangle$ dove M è una TM
2. Eseguo H su input $\langle M, \langle M\rangle\rangle$
3. Se H accetta **rifiuto**, mentre se H rifiuta **accetto**

Ricapitolando, abbiamo:

$$D(\langle M\rangle) = \begin{cases} \textit{accetta} & \text{se } M \text{ rifiuta } \langle M\rangle \\ \textit{rifiuta} & \text{se } M \text{ accetta } \langle M\rangle \end{cases}$$

Ma D è un decisore e quindi sta in A_{TM}, quindi cosa succede se proviamo ad eseguire D con input $\langle D\rangle$?

$$D(\langle D\rangle) = \begin{cases} \textit{accetta} & \text{se } D \text{ rifiuta } \langle D\rangle \\ \textit{rifiuta} & \text{se } D \text{ accetta } \langle D\rangle \end{cases}$$

Che è una **contraddizione** in quanto D fa sempre l'opposto di se stessa. Questo significa che A_{TM} non può essere deciso.

Analisi di come è stata usata la diagonalizzazione: iniziamo vedendo il comportamento di H e D tramite delle tabelle. Sulle righe sono presenti le TM, mentre sulle colonne le loro descrizioni. Le celle della tabella indicano se una TM di una data riga accetta l'input riportato sulla data colonna: se c'è *accept* vuol dire che ha accettato, mentre se è vuoto vuol dire che non ha accettato o è andata in loop:

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	...
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					...
M_4	*accept*	*accept*			
$\vdots$	$\vdots$				

La seguente tabella è il risultato dell'esecuzione di H sulla tabella precedente:

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	...
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	
M_3	*reject*	*reject*	*reject*	*reject*	...
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$	$\vdots$				

Siccome anche D è una TM, prima o poi deve per forza apparire nella lista delle $M_1, M_2, \dots,$ quindi vediamo cosa succede aggiungendo D e la sua descrizione nella tabella di H:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\dots$	$\langle D \rangle$	$\dots$
M_1	$\underline{accept}$	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	$\underline{accept}$	*accept*	*accept*		*accept*	
M_3	*reject*	*reject*	$\underline{reject}$	*reject*		*reject*	$\dots$
M_4	*accept*	*accept*	*reject*	$\underline{reject}$		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		$\underline{?}$	
$\vdots$			$\vdots$				$\ddots$

La contraddizione avviene proprio dove si trova il carattere ?, in quanto in tale posizione dovrebbe accettare e rifiutare nello stesso momento. Impossibile.

9.2 Linguaggi non Turing riconoscibili

Il seguente teorema mostra che se un linguaggio ed il suo complemento sono entrambi Turing riconoscibili, allora tale linguaggio è decidibile. Questo significa che per ogni linguaggio indecidibile si ha che o il linguaggio stesso o il suo complemento non è Turing riconoscibile.
Un linguaggio si dice **co-Turing riconoscibile** se è il complemento di una linguaggio Turing riconoscibile.

9.2.1 Teorema - Un linguaggio è decidibile se e solo se è Turing riconoscibile e co-Turing riconoscibile

Un linguaggio è **decidibile** se e solo se è **Turing riconoscibile** e **co-Turing riconoscibile**.

Dimostrazione: due direzioni

- $\Rightarrow$ Se L è decidibile allora sia L che $\overline{L}$ sono Turing riconoscibili. Ogni linguaggio decidibile è anche Turing riconoscibile e il complemento di un linguaggio decidibile è anch'esso decidibile.

- $\Leftarrow$ Siano L e $\overline{L}$ Turing riconoscibili e siano M_1 e M_2 le corrispettive TM che li riconoscono. Costruisco una TM M che decide L:

    1. Su input w
    2. Eseguo M_1 e M_2 su input w in parallelo
    3. Se M_1 accetta **accetto**, se M_2 accetta **rifiuto**

    Analisi: $\forall w$ si ha che $w \in L$ oppure $w \in \overline{L}$. Questo significa che uno tra M_1 e M_2 accetta w. Siccome M si ferma quando M_1 si ferma oppure M_2 si ferma, abbiamo la certezza che non vada in loop e quindi è un **decisore**.

In più M accetta solamente le stringhe in L e rifiuta tutte le stringhe in $\overline{L}$, quindi M decide L e quindi L è decidibile.

9.2.2 Corollario - $\overline{A_{\text{TM}}}$ è non Turing riconoscibile

Dal teorema precedente si evince che $\overline{A_{\text{TM}}}$ è **non Turing riconoscibile**.

Dimostrazione: sappiamo che A_{TM} è Turing riconoscibile. Se lo fosse anche $\overline{A_{\text{TM}}}$, allora A_{TM} sarebbe decidibile. Ma abbiamo dimostrato che A_{TM} è indecidibile, il che significa che $\overline{A_{\text{TM}}}$ è necessariamente non Turing riconoscibile.

10 Tecnica della riduzione

La **tecnica della riduzione** è un modo per convertire un problema in un altro in modo tale che una soluzione al secondo problema può essere utilizzata per risolvere il primo.
Siano A e B due problemi. Diremo che

$$A \leq B \quad (A \text{ si riduce a } B)$$

Posso usare le soluzioni di B per risolvere A. Se $A \leq B$ e B è **indecidibile**, allora anche A è **indecidibile**.
Esempi informali:

- Calcolare l'area di un rettangolo si riduce a misurare la lunghezza dei suoi lati
- Orientarsi in una nuova città si riduce a comprare una mappa di quella città
- Come abbiamo visto nel teorema 8.2.2.1, A_{NFA} si riduce a A_{DFA}

10.1 Problemi indecidibili

10.1.1 Halting problem

Il problema HALT_{TM} è il problema di verificare se una TM si ferma o meno su un dato input (sia che accetta, sia che rifiuta), noto anche come ***halting problem***. Sia:

$$\text{HALT}_{\text{TM}} = \{\langle M, w\rangle : M \text{ è una TM e } M \text{ si ferma su input } w\}$$

Usiamo l'indecidibilità di A_{TM} per dimostrare che anche HALT_{TM} lo è.

10.1.1.1 Teorema - $\mathbf{HALT_{TM}}$ è indecidibile

$\mathrm{HALT_{TM}}$ è **indecidibile**.

Dimostrazione: Per riduzione mostro che $A_{\mathrm{TM}} \leq \mathrm{HALT_{TM}}$ (A_{TM} si riduce a $\mathrm{HALT_{TM}}$). Assumiamo che $\mathrm{HALT_{TM}}$ sia decidibile e che la TM R sia il suo decisore. Costruisco S che decide A_{TM}:
$S =$ "input $\langle M, w\rangle$ dove M è una TM e w è una stringa

1. Eseguo la TM R su input $\langle M, w\rangle$
2. Se R rifiuta **rifiuto**, ovvero M non si è fermata su input w e quindi sicuramente non lo accetta
3. Se R accetta, simula M su input w finché non si ferma (garantito da R)
4. Se M ha accettato **accetto**, altrimenti **rifiuto**"

N.B.: stiamo **assumendo** che R esista! L'unica cosa che ci dice R è se M si sia fermata o meno (accettando o rifiutando) su input w.
Chiaramente, se R decide $\mathrm{HALT_{TM}}$, allora S decide A_{TM}. Siccome A_{TM} è indecidibile deve esserlo anche $\mathrm{HALT_{TM}}$.

Questo tipo di prova viene utilizzata nella maggior parte delle dimostrazioni di indecidibilità di un linguaggio, tranne per A_{TM} che viene dimostrato direttamente tramite la diagonalizzazione.

10.1.2 Test del vuoto

Ricordiamo che il test del vuoto consiste nel verificare se il linguaggio riconosciuto da una TM è vuoto. Sia:

$$E_{\mathrm{TM}} = \{\langle M\rangle : M \text{ è una TM e } L(M) = \emptyset\}$$

10.1.2.1 E_{TM} è indecidibile

E_{TM} è **indecidibile**.

Idea: assumiamo che E_{TM} sia decidibile e proseguiamo dimostrando che anche A_{TM} lo è, giungendo quindi ad una contraddizione. Supponiamo ci sia una TM R che decide E_{TM} e la usiamo per creare la TM S che decide A_{TM}. Su R non verrà lanciata direttamente $\langle M\rangle$, ma una versione modificata di $\langle M\rangle$ dove viene garantito che tutte le stringhe diverse da w vengano rifiutate, mentre su input w la macchina funziona normalmente. Chiamiamo questa macchina modificata M_1. Avremo quindi:

$$L(M_1) = \begin{cases} \{w\} & \text{se } M \text{ accetta } w \\ \emptyset & \text{se } M \text{ rifiuta } w \end{cases}$$

Dimostrazione: costruiamo le versione modificata di M. Sia questa chiamata M_1:

1. Su input x, se $x \neq w$ **rifiuto**
2. Se $x = w$, simulo M su input w e **accetto** se e solo se M accetta

Questo viene fatto in quanto M potrebbe accettare cose che non sono w, andando a "contaminare" il risultato. Costruisco ora S che decide A_{TM} come segue:
$S =$ "su input $\langle M, w\rangle$ dove M è un TM e w è una stringa

1. Uso la descrizione di M e di w per costruire la TM M_1
2. Eseguo R su input $\langle M_1\rangle$ per verificare se $L(M_1) = \emptyset$
3. Se R accetta **rifiuto**, se R rifiuta **accetto**"

N.B.: come prima, stiamo **assumendo** l'esistenza di R, non la stiamo esplicitando. Il suo unico scopo è decidere se il linguaggio di una TM è vuoto.
Se R fosse realmente un decisore per E_{TM}, S lo sarebbe per A_{TM}. Ma sappiamo che A_{TM} è indecidibile e quindi anche E_{TM} deve per forza esserlo.

10.1.3 TM che riconosce un linguaggio regolare

Questo problema si occupa di determinare se una data TM riconosce un linguaggio che può essere riconosciuto anche da un modello di computazione più semplice. Ad esempio, sia:

$$\text{REG}_{\text{TM}} = \{\langle M\rangle : M \text{ è una TM e } L(M) \in \text{REG}\}$$

Questo linguaggio comprende le macchine di Turing che riconoscono lo stesso linguaggio riconosciuto da un automa a stati finiti, ovvero un linguaggio regolare.

10.1.3.1 REG_{TM} è indecidibile

REG_{TM} è **indecidibile**.

Idea: assumiamo che REG_{TM} si regolare e proseguiamo dimostrando che anche A_{TM} lo è, giungendo quindi ad una contraddizione. Supponiamo che la TM R decida REG_{TM} e usiamola per costruire una TM S che decide A_{TM}. L'idea è quella di far prendere ad S il proprio input $\langle M, w\rangle$ e fargli modificare M in modo tale che la TM risultate riconosca un linguaggio regolare se e solo se M accetta w. Sia questa macchina modificata M_2. Abbiamo due casi:

- Se M non accetta w, M_2 riconosce un linguaggio non regolare
- Se M accetta w, M_2 simula M su w

Scegliamo $\{0^n 1^n : n \geq 0\}$ come linguaggio non regolare da far riconoscere ad M_2. Costruzione di M_2:
$M_2 =$ "su input x:

1. Se x è della forma $0^n 1^n$ **accetto**

2. Se x non ha tale forma, simulo M su w e **accetto** x se M accetta w"

Siccome nel secondo passaggio l'accettazione dell'input w da parte di M è scollegato dall'input x, si ha che se M accetta w allora tutte le x che vengono passate in input verranno accettate, andando a formare Σ^*. Il linguaggio di M_2 sarà quindi in seguente:

$$L(M_2) = \begin{cases} \{0^n 1^n : n \geq 0\} & \text{se } M \text{ rifiuta } w \\ \Sigma^* & \text{se } M \text{ accetta } w \end{cases}$$

Ci resta quindi solo da testare se $L(M_2) \in \text{REG}$ per sapere se M accetta w.
Dimostrazione: Sia R un decisore di REG_{TM} e costruiamo la TM S che decide A_{TM}. S viene costruito nella seguente maniera:
$S =$ "su input $\langle M, w\rangle$ dove M è una TM e w è una stringa:

1. Costruisco M_2
2. Eseguo R su input $\langle M_2\rangle$ per verificare se $L(M_2) \in \text{REG}$
3. Se R accetta **accetto**, altrimenti **rifiuto**"

Analisi: Se S accetta $\Rightarrow$ R accetta su input $\langle M_2\rangle \Rightarrow L(M_2)$ è regolare $\Rightarrow M$ accetta w. Ma siccome A_{TM} è indecidibile, allora anche REG_{TM} deve esserlo.

10.1.4 Equivalenza di TM

A volte usare una riduzione ad E_{TM} piuttosto che A_{TM} è più conveniente, come ad esempio nel seguente caso. Sia:

$$EQ_{\text{TM}} = \{\langle M_1, M_2\rangle : M_1, M_2 \text{ sono TM e } L(M_1) = L(M_2)\}$$

10.1.4.1 EQ_{TM} è indecidibile

EQ_{TM} è **indecidibile**.

Idea: mostriamo che se EQ_{TM} fosse decidibile allora lo sarebbe anche E_{TM} tramite la tecnica della riduzione da E_{TM} a EQ_{TM}. E_{TM} è il problema di determinare se il linguaggio di una TM è vuoto. EQ_{TM} è il problema di verificare i linguaggi di due TM siano equivalenti, ma se uno dei due linguaggi risulta essere $\emptyset$, restiamo con il problema di determinare se il linguaggio dell'altra macchina sia vuoto. In altri termini, E_{TM} è un caso speciale di EQ_{TM} dove una delle due macchine è impostata per riconoscere il linguaggio $\emptyset$.
Dimostrazione: sia la TM R un decisore di EQ_{TM} e costruiamo la TM S che decide E_{TM} nel seguente modo:
$S =$ "su input $\langle M\rangle$ dove M è una TM

1. Eseguo R su input $\langle M, M_1\rangle$, dove M_1 è una macchina che rifiuta ogni tipo di input
2. Se R accetta **accetto**, altrimenti **rifiuto**"

Analisi: Se R decide $EQ_{\text{TM}} \Rightarrow S$ decide E_{TM}. Ma E_{TM} è indecidibile $\Rightarrow EQ_{\text{TM}}$ è indecidibile.

10.2 Mapping reduction

Altro metodo di riduzione dove la TM calcola una funzione $f(n)$ tale che:

1. Inizia con n sul nastro (di input)
2. Termina con $f(n)$ sul nastro (di output)

10.2.1 Definizione - Funzione calcolabile

Una **funzione** $f : \Sigma^* \to \Sigma^*$ (che da stringhe va in stringhe) è **calcolabile** se esiste una TM tale che $\forall w \in \Sigma^*$ questa termina con solo $f(w)$ sul nastro.

Esempio: La funzione che prende in input w e restituisce la descrizione di una TM $\langle M' \rangle$ se $w = \langle M \rangle$ è una codifica di una TM M.

10.2.2 Definizione - Mapping reduction

Un linguaggio A è (mapping) riducibile ad un linguaggio B se esiste una **funzione calcolabile** $f : \Sigma^* \to \Sigma^*$ tale che

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

f viene detta **riduzione** da A a B e si scrive $A \leq_m B$.
Essenzialmente quello che f fa è convertire istanze del problema A in istanze del problema B.

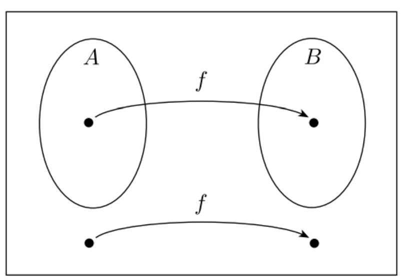


Figure 48: Una funzione f che riduce A a B

Una mapping reduction da A a B fornisce un modo per convertire quesiti sulla membership di A in questi sulla membership di B. In altre parole, se si vuole testare $w \in A$, usiamo la riduzione f per mappare w in $f(w)$ per verificare se $f(w) \in B$. Se un problema è mapping riducibile ad un secondo problema di cui si conosce la soluzione, si ottiene automaticamente la soluzione al problema originale.

10.2.3 Teorema - Se $A \leq_m B$ e B è decidibile, allora A è decidibile

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

Dimostrazione: sia R la TM che decide B e f la funzione che riduce A a B. Costruiamo il decisore S per A come segue:

1. Su input w calcolo $f(w)$
2. Eseguo R su input $f(w)$
3. Se R accetta **accetto**, altrimenti **rifiuto**

Correttezza: se $w \in A$ allora $f(w) \in B$ in quanto f è una riduzione da A a B. Quindi R accetta $f(w)$ ogni qualvolta che $w \in A$ e quindi S funziona come volevamo.

10.2.4 Corollario - Se $A \leq_m B$ e A è indecidibile, allora B è indecidibile

Se $A \leq_m B$ e A è indecidibile, allora B è indecidibile.

10.2.5 Esempi

10.2.5.1 $A_{\mathbf{TM}} \leq_m \mathbf{HALT_{TM}}$

Cerchiamo una **funzione calcolabile** $f : \Sigma^* \to \Sigma^*$ che prende input nella forma $\langle M, w \rangle$ e restituisce output nella forma $\langle M', w' \rangle$, tale che

$$\langle M, w \rangle \in A_{\text{TM}} \text{ se e solo se } \langle M', w' \rangle \in \text{HALT}_{\text{TM}}$$

oppure in altri termini

$$\forall x \in \Sigma^* \quad x = \langle M, w \rangle \in A_{\text{TM}} \text{ se e solo se } f(x) = f(\langle M, w \rangle) \in \text{HALT}_{\text{TM}}$$

Sia F la TM che calcola tale f:

1. Su input $\langle M, w \rangle$ dove M è una TM e w è una stringa
2. Costruisco la seguente TM M'
   (a) Su input x stringa
   (b) Eseguo M su input x
   (c) Se M accetta **accetto**, altrimenti entro in un loop
3. Metto in output $\langle M', w \rangle$

Correttezza: se $\langle M, w \rangle \in A_{\text{TM}}$, allora M accetta l'input w e di conseguenza M' termina, quindi $\langle M', w \rangle \in \text{HALT}_{\text{TM}}$. Se invece $\langle M, w \rangle \notin A_{\text{TM}}$, allora M non accetta w oppure va in loop e di conseguenza anche M', il che significa che $\langle M', w \rangle \notin \text{HALT}_{\text{TM}}$. □

10.2.5.2 $E_{\mathbf{TM}} \leq_m EQ_{\mathbf{TM}}$

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

$$\langle M \rangle \in E_{\text{TM}} \text{ se e solo se } \langle M, M' \rangle = f(\langle M \rangle) \in EQ_{\text{TM}}$$

Sia F la TM che calcola tale f:

1. Su input $\langle M \rangle$ dove M è una TM
2. Costruisco $\langle M' \rangle$ codifica di TM M' tale che $\forall x$ M' rifiuta
3. Output: $\langle M, M' \rangle$

Correttezza: se $\langle M \rangle \in E_{\text{TM}}$, allora $L(M) = \emptyset$. Per definizione, $L(M') = \emptyset$ in quanto è la TM che rifiuta ogni tipo di input, quindi $L(M) = L(M')$ e di conseguenza $\langle M, M' \rangle \in EQ_{\text{TM}}$.
Se $f(\langle M \rangle) \in EQ_{\text{TM}} \Rightarrow L(M) = L(M')$, ma $L(M') = \emptyset$ e quindi per forza $L(M) = \emptyset$. Ciò significa che $\langle M \rangle \in E_{\text{TM}}$. $\square$

10.3 Mapping reduction per dimostrare la Turing riconoscibilità

La **mapping reduction** risulta particolarmente utile per dimostrare la Turing riconoscibilità dei linguaggi. Ancora più utile invece per dimostrare la **non Turing riconoscibilità**.

10.3.1 Teorema - Se $A \leq_m B$ e B è Turing riconoscibile, allora anche A è Turing riconoscibile

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

10.3.2 Corollario - Se $A \leq_m B$ e A è non Turing riconoscibile, allora anche B è non Turing riconoscibile.

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

Applicazione: sappiamo che $\overline{A_{\text{TM}}}$ è **non Turing riconoscibile**. Lo uso per mostrare che anche altri linguaggi sono non Turing riconoscibili tramite una **mapping reduction**. Risulta però difficile lavorare su $\overline{A_{\text{TM}}}$ quindi, con il seguente teorema, possiamo semplificare un po' il lavoro.

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

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

In altri termini, se $\overline{A}$ è non Turing riconoscibile e $A \leq_m B$ allora anche $\overline{B}$ è non Turing riconoscibile.

10.3.3.1 Esempio

Vogliamo dimostrare che EQ_{TM} è non Turing riconoscibile. Sappiamo che $\overline{A_{\text{TM}}}$ è non Turing riconoscibile. Facciamo quindi vedere che:

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

Nota: al posto di lavorare con $\overline{A_{\text{TM}}}$, grazie al teorema 10.3.3

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

possiamo lavorare direttamente con A_{TM} che risulta essere più facile.
Cerchiamo una **funzione calcolabile** $f : \Sigma^* \to \Sigma^*$ tale che:

$$\langle M, w\rangle \in A_{\text{TM}} \text{ se e solo se } f(\langle M, w\rangle) \in \overline{EQ_{\text{TM}}}$$

Sia F la TM che calcola tale f:

1. Su input $\langle M, w\rangle$ dove M è una TM e w è una stringa
2. Costruisco $\langle M_1, M_2\rangle$ dalle seguenti TM:
   (a) M_1 rifiuta sempre
   (b) M_2 su input x esegue M su w e se M accetta w, M_2 accetta x
3. Output: $\langle M_1, M_2\rangle$

Correttezza:

- $\Rightarrow$ Se $\langle M, w\rangle \in A_{\text{TM}}$, allora M_2 accetta sempre e M_1 rifiuta sempre, quindi $L(M_1) = \emptyset \neq L(M_2) = \Sigma^* \Rightarrow f(\langle M, w\rangle) \in \overline{EQ_{\text{TM}}}$.
- $\Leftarrow$ Se $\langle M_1, M_2\rangle \in \overline{EQ_{\text{TM}}}$, allora $L(M_1) \neq L(M_2)$. Siccome M_1 è stata costruita per rifiutare sempre, segue che M_2 accetta, che per costruzione accetta quando M accetta w e quindi $\langle M, w\rangle \in A_{\text{TM}}$. □

10.4 Esercizi su riduzione

10.4.1 Esercizio 1

Si consideri il seguente linguaggio:

$$L = \{\langle M\rangle : M \text{ è una TM tale che accetta tutte e sole le stringhe di lunghezza dispari}\}$$

Dimostrare che L è indecidibile.
Dimostriamo che $A_{\text{TM}} \leq_m L$, ovvero che esiste una **funzione calcolabile** $f : \Sigma^* \to \Sigma^*$ tale che

$$\forall x \in \Sigma^* \quad x = \langle M, w\rangle \in A_{\text{TM}} \Leftrightarrow f(\langle M, w\rangle) \in L$$

Algoritmo:

1. Su input $\langle M, w\rangle$ dove M è una TM e w una stringa
2. Costruisco $\langle M'\rangle$ dove M' è una TM tale che
   (a) Su input x se $|x|$ è pari rifiuta
   (b) Altrimenti esegue M su input w e se M accetta, M' **accetta**.
   Se M rifiuta, M' **rifiuta**.
3. Output: $\langle M'\rangle$

<u>Analisi</u>: se $\langle M, w\rangle \in A_{\text{TM}}$ allora M accetta la stringa w. $L(M')$ contiene tutte e sole le $x \in \Sigma^* : |x|$ è dispari, quindi $\langle M'\rangle \in L$.
Se $\langle M, w\rangle \notin A_{\text{TM}}$ allora M rifiuta l'input w o va in loop. In questo caso M' rifiuta e quindi $\langle M'\rangle \notin L$.

10.4.2 Esercizio 2

Si consideri il seguente linguaggio:

$$V = \{\langle T, T'\rangle : T, T' \text{ sono TM e } L(T) \cup L(T') = \Sigma^*\}$$

Dimostrare che V è indecidibile.
Dimostro che $A_{\text{TM}} \leq_m V$, ovvero che esiste una **funzione calcolabile** $f : \Sigma^* \to \Sigma^*$ tale che:

$$\forall x \in \Sigma^* \quad \langle M, w\rangle \in A_{\text{TM}} \Leftrightarrow f(\langle M, w\rangle) = \langle T, T'\rangle \in V$$

<u>Codice di T</u>:

1. Su input x **rifiuta**.

<u>Codice di T'</u>:

1. Su input x, esegue M su w e **accetta** x se M accetta w.
   Altrimenti **rifiuta**.

<u>Correttezza</u>:

$\Rightarrow$ Se $\langle M, w\rangle \in A_{\text{TM}}$ allora M accetta l'input w e quindi si ha:

$$L(T) = \emptyset$$
$$L(T') = \Sigma^*$$
$$L(T) \cup L(T') = \Sigma^*$$

e quindi $\langle T, T'\rangle \in V$.

$\Leftarrow$ Se $\langle M, w\rangle \notin A_{\text{TM}}$ allora M rifiuta l'input w oppure va in loop. In questo caso:

$$L(T) = \emptyset$$
$$L(T') = \emptyset$$
$$L(T) \cup L(T') \neq \Sigma^*$$

e quindi $\langle T, T'\rangle \notin V$.

10.4.3 Esercizio 3

Se il linguaggio A è Turing-riconoscibile e $A \leq_m \overline{A}$, allora A è decidibile.

Soluzione: siccome $A \leq_m \overline{A}$, allora $\overline{A} \leq_m A$ usando la stessa riduzione. Sappiamo che A è Turing-riconoscibile e quindi $\overline{A}$ è anch'essa Turing-riconoscibile. Ma allora A è sia Turing-riconoscibile che coTuring-riconoscibile e quindi A è decidibile (teorema 9.2.1, pag. 71).

11 Complessità

Anche se un problema è decidibile e quindi computabile, potrebbe essere non risolubile nella pratica in quanto potrebbe richiedere un quantità spropositata di tempo o spazio.

11.1 Definizione - Tempo di esecuzione di una TM

Sia M un decisore. La **complessità di tempo** di M è una funzione

$$T : \mathbb{N} \to \mathbb{N} \text{ tale che } T(n) = \max_{\substack{x \in \Sigma^* \\ |x| = n}} \# \text{ di passi di } M(x)$$

In altri termini, $T(n)$ è il massimo numero di step che M esegue su un input di lunghezza n. Questo numero di step può variare in base al tipo di input che riceve, come ad esempio nel caso di un grafo, può dipendere dal numero di nodi, archi ed il grado massimo. Per semplicità calcoliamo il tempo di esecuzione di un algoritmo in base alla dimensione della stringa che codifica l'input. Inoltre effettueremo sempre una ***worst case analysis***, ovvero andremo a considerare il tempo di esecuzione più lungo di tutti gli input di una particolare lunghezza.

Esempio: Sia M una TM. M su input w muove la testina a destra fino a che non legge il carattere "$\sqcup$". Poi accetta. In questo caso:

$$T(n) = n$$

11.2 Notazione O grande

Nelle nostre analisi, le costanti non ci interessano e terremo conto solamente del termine di ordine maggiore, in quanto è quello che andrà a dominare sugli altri termini quando gli input sono di grandi dimensioni.
Ad esempio, $f(n) = 6n^3 + 2n^2 + 20n + 45$ ha 4 termini, ma terremo conto solamente di $6n^3$, di cui scarteremo il coefficiente 6, rimanendo con n^3.

11.2.1 Definizione - O grande

Siano f e g due funzioni $f, g : \mathbb{N} \to \mathbb{R}^+$. Allora $f(n) = O(g(n))$ se esistono $c, n_0 \in \mathbb{N}^+$ tali che per ogni intero $n \geq n_0$

$$f(n) \leq c \cdot g(n)$$

Quando $f(n) = O(g(n))$, diciamo che $g(n)$ è un **limite asintotico superiore** di $f(n)$.

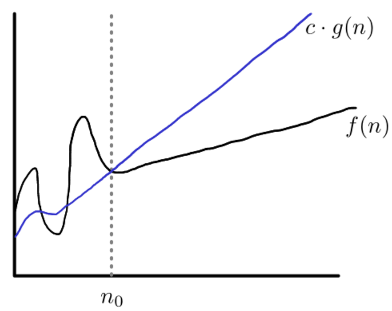


Figure 49: $f(n) = O(g(n))$

11.2.2 Esempio di analisi di un algoritmo

Analizziamo il seguente algoritmo dato per il riconoscimento del linguaggio $A = \{0^n 1^n : n \geq 0\}$:
Sulla stringa di input w:

1. Scorri tutto il nastro e **rifiuta** se uno 0 viene trovato dopo un 1
2. Ripeti finché ci sono 0 e 1 sul nastro:
3. Scorri il nastro ed elimina un singolo 0 ed un singolo 1
4. Se rimangono degli 0 dopo che tutti gli 1 sono stati eliminati o viceversa **rifiuta**, altrimenti se non sono rimasti ne 0 ne 1, **accetta**

Analisi: per l'analisi consideriamo ogni stage separatamente:

- Nello stage 1. la macchina scorre tutto il nastro per verificare che l'input sia nella forma 0*1*, impiegando n step. Dopodiché riposiziona la testina all'inizio del nastro, che richiede nuovamente n step, per un totale di $2n$ step. Con la notazione O grande diciamo questo stage impiega $O(n)$ step. Notare come l'utilizzo della notazione O grande ci permette di eliminare dalla descrizione dell'algoritmo alcuni dettagli che incrementano il numero di step di un numero costante di passi

- Negli stage 2. e 3. la macchina scorre ripetutamente il nastro eliminando un singolo 0 ed un singolo 1, dove ogni scorrimento costa $O(n)$ passi. Siccome ogni scorrimento elimina 2 simboli, ci sono solo $\frac{n}{2}$ scorrimenti. Quindi il totale è $\frac{n}{2}O(n) = O(n^2)$ passi.

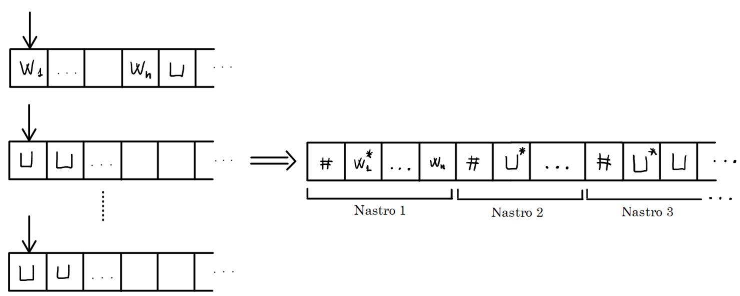
- Nello stage 4. la macchina scorre il nastro per decidere se accettare o meno. Richiede $O(n)$ passi.

Il tempo totale è quindi $O(n) + O(n^2) + O(n) = O(n^2)$.

11.3 Teorema - Sia M una TM multi nastro con complessità $T(n)$. Allora esiste una TM M' singolo nastro con complessità $O(T^2(n))$ tale che $L(M) = L(M')$

Sia M una TM **multi nastro** con complessità $T(n)$. Allora esiste una TM M' **singolo nastro** con complessità $O(T^2(n))$ tale che $L(M) = L(M')$.

Dimostrazione (bozza): abbiamo già visto come simulare una TM M multi nastro con una TM M' singolo nastro:



Se M va in tempo $T(n)$, la lunghezza del nastro sarà $\leq T(n)$. Questo significa che il nastro di M' avrà lunghezza $\leq kT(n) + k = O(T(n))$, dove k è il numero di nastri di M. Per simulare un singolo passo di M, M' richiede $O(T(n))$ passi in quanto deve scansionare tutto il nastro per aggiornare tutti i simboli marcati ed il contenuto del nastro. Dopodiché M' simula i $T(n)$ passi di M con $O(T(n))$ passi per ogni passo di M, quindi in $T(n) \times O(T(n)) = O(T^2(n))$. Il tempo di M' sarà quindi:

$$O(n) + O(T^2(n)) = O(T^2(n)) \text{ se } T \geq n$$

dove l'$O(n)$ iniziale deriva dall'operazione iniziale di formattazione del nastro nel formato corretto.

Con questa dimostrazione possiamo quindi osservare che passare da una TM multi nastro a una TM singolo nastro fa aumentare la complessità di tempo di un fattore quadratico.

11.3.1 Esempio - PALINDROMES

Si consideri il seguente linguaggio:

$$\text{PALINDROMES} = \{w : w \text{ è palindroma}\}$$

Esiste una TM singolo nastro che decide PALINDROMES in $O(n^2)$ passi. Se passiamo ad una TM con due nastri però, questa deciderà PALINDROMES in $O(n)$ passi in quanto:
Sulla stringa di input w:

1. Scorri il primo nastro fino alla fine e ricopia il contenuto sul secondo nastro. Dopodiché riposiziona la testina del primo nastro all'inizio
2. Scorri la testina del primo nastro a destra di una posizione e quella del secondo nastro verso sinistra di una posizione, se i due caratteri sono diversi **rifiuta**. Se la tesina del primo nastro ha raggiungo la fine dell'input e la testina del secondo nastro ha raggiunto la fine del nastro, **accetta**

Chiaramente questa TM esegue $O(n)$ passi.

11.4 Definizione - La classe di tempo DTIME

Sia $t : \mathbb{N} \to \mathbb{R}^+$. La classe di tempo **DTIME** (*Deterministic Time*) è definita come segue:

$$\text{DTIME}(t(n)) = \{L \in \Sigma^* : \exists \text{ TM } M \text{ deterministica che decide } L \text{ in } O(t(n)) \text{ passi}\}$$

In altri termini, DTIME($t(n)$) è l'**insieme** di linguaggi decisi da una TM in $t(n)$ passi.

11.5 Definizione - La classe P

La classe P è l'insieme dei linguaggi decidibili in tempo **polinomiale**, ovvero in $O(n^k)$ passi per qualche k:

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

Un punto chiave di questa classe è la possibilità di **ignorare il tipo di modello** della TM.

11.6 Teorema - Gerarchia di tempo

Esiste un linguaggio L tale che $L \notin \text{DTIME}(n^{2.999\ldots})$ ma $L \in \text{DTIME}(n^8)$.
<u>Task</u>: simula una TM per n^3 passi:

- Fattibile in tempo polinomiale in n^3
- Non fattibile in meno di n^3 passi

11.6.1 Teorema - Funzione tempo costruttibile

Se $t(n)$ è *"**tempo costruttibile**"* (supponiamo per adesso n^3), allora esiste una TM U_t che data $< M, w >$ simula $M(w)$ per $t(n)$ passi in tempo

$$O(|\langle M\rangle|^4 \cdot t(n) \cdot \log t(n))$$

Dove $\log t(n)$ indica il numero di bit impiegati per il contatore utilizzato per fermare la macchina.

Torniamo alla dimostrazione del teorema 11.6
Dimostrazione: consideriamo una TM D che si comporta nel seguente modo:

1. Su input $\langle M, w\rangle$ dove M è una TM e w è l'input con $|w| = n$
2. Usa U_{n^3} per simulare $M(\langle M, w\rangle)$ per n^3 passi, dove $n = |\langle M, w\rangle|$
3. Se U_{n^3} accetta, D **rifiuta**
   Se U_{n^3} rifiuta (o non termina), D **accetta**

D è un **decisore** e quindi definisce L, ovvero le stringhe accettate da D. Da una parte abbiamo che $L \in \text{DTIME}(n^8)$.
Claim: $L \notin \text{DTIME}(n^2)$ (infatti è in $\text{DTIME}(n^{2.999\cdots})$).
Per contraddizione. Sia S il decisore per L con tempo $t(n) = O(n^2)$. Lanceremo S su input $\langle S, 0^l\rangle$ con $l \in \mathbb{N}$, dove $\langle S, 0^l\rangle$ significa:

$$\langle S, 0^l\rangle = \langle S\rangle, 0, \ldots, 0$$

Se $c = |\langle S\rangle|$, S termina in $\leq t(l + c)$ passi, dove:

$$t(l + c) = O((l + c)^2) \leq (l + c)^3 \text{ per } l \text{ grande } (l \geq l^*)$$

Siccome S **decide** L, S equivale a D: $S(\langle S, 0^{l^*}\rangle) = D(\langle S, 0^{l^*}\rangle)$.
Si ha quindi che S non va in timeout in quanto $(l^* + c)^3 \geq O((l + c)^2)$. Siccome D usa U_t per simulare S e rifiuta quando S accetta

$$D(\langle S, 0^{l^*}\rangle) = \neg S(\langle S, 0^{l^*}\rangle)$$

che è una contraddizione. □
L non è naturale, quindi considero:

$$\text{BA}_{n^3} = \{\langle M, w\rangle : M(w) \text{ accetta in} \leq |w^3| \text{ passi}\}$$

dove BA sta per *"**Bounded Acceptance**"*.
Claim: $\text{BA}_{n^3} \in \text{DTIME}(n^8)$
Claim: $\text{BA}_{n^3} \notin \text{DTIME}(n^2)$
Dimostrazione: Per riduzione, sia B un decisore per BA_{n^3} in $O(n^2)$ passi. Decido L del teorema 11.6 usando B su input $\langle M, w\rangle$ tale che $|\langle M, w\rangle| = n$, ovvero devo decidere se $M(\langle M, w\rangle)$ accetta in n^3 passi:

- Preparo $\langle M, \langle M, w\rangle\rangle$ che richiede $O(n^2)$ passi
- Lancio $B(\langle M, \langle M, w\rangle\rangle)$ e faccio l'opposto.

Tempo: $O(|\langle \underbrace{M}_{\leq n}, \langle \underbrace{M}_{\leq n}, w\rangle\rangle|^2) = O(n^2)$.

11.7 La classe EXP

Per il teorema di gerarchia di tempo, sappiamo che esiste un linguaggio L tale che $L \notin P$ ma è decidibile in tempo esponenziale. Ad esempio:

$$\mathrm{BA}_{2^n} = \{\langle M, w\rangle : M \text{ è una TM che accetta } M(w) \text{ in } \leq 2^{|w|} \text{ passi}\}$$

Con il teorema di gerarchia è facile vedere che $\mathrm{BA}_{2^n} \notin \mathrm{P}$. Usando la TM universale si ha

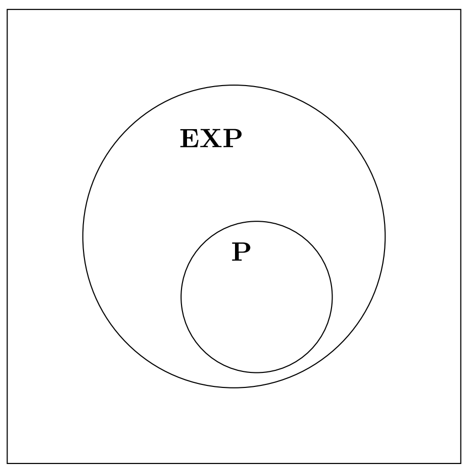
$$\mathrm{BA}_{2^n} \in \mathrm{DTIME}(2^n \cdot n^5) \subseteq \mathrm{DTIME}(2^{O(n)})$$

11.7.1 Definizione - La classe EXP

$$\mathrm{EXP} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}\left(2^{n^k}\right)$$

11.7.2 Corollario - P $\subsetneq$ EXP

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



Talvolta può accadere che $L \in \mathrm{P}$ grazie ad un algoritmo ”***furbo***”.

11.8 Problemi in P

11.8.1 PATH

Si consideri il seguente linguaggio:

$$\mathrm{PATH} = \{\langle G, s, t\rangle : G = (V, E) \text{ è un grafo, } s, t \in V \text{ ed esiste un cammino } s \rightsquigarrow t\}$$

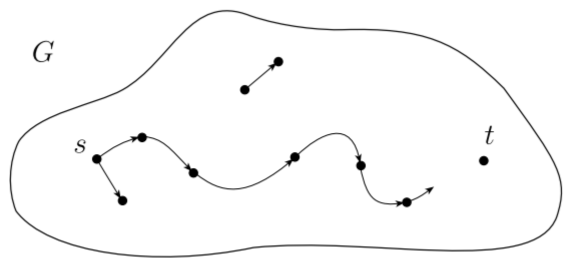


Figure 50: Esiste un cammino da s a t?

11.8.1.1 Teorema - PATH $\in$ P

PATH $\in$ P.

Idea: un primo algoritmo che potrebbe venire in mente sarebbe un algoritmo di **brute-force**, ma non sarebbe abbastanza veloce. Questo algoritmo considererebbe **tutti quanti i possibili cammini nel grafo**, verificando se ne esiste uno che porta dal nodo s al nodo t. Il problema di questo algoritmo è che il **numero totale** di cammini in un grafo corrisponde circa a n^n, dove n è il numero di nodi presenti all'interno del grafo e di conseguenza:

$$\text{PATH} \in \text{EXP}$$

Per raggiungere un tempo **polinomiale** per PATH, dobbiamo fare qualcosa che eviti l'utilizzo di brute-force. Un metodo potrebbe essere quello di usare una tecnica di ricerca su grafi come ad esempio una breadth-first search.
Dimostrazione: Algoritmo **polinomiale** per PATH:

1. Su input $\langle G, s, t \rangle$ dove G è un grafo e s, t sono nodi in G
2. Marca il nodo s
3. Ripeti:
   (a) Scansiona tutti gli archi (u, v) di G
   (b) Se il nodo u è marcato ma il nodo v non lo è, marca v
   (c) Stop se non c'è nessun nuovo nodo marcato

Questo algoritmo lavora in tempo $O(m \cdot n) = O(n^3)$, dove $n^2 \geq m =$ numero di archi in G e quindi

$$\text{PATH} \in \text{P}$$

11.8.1.2 Scelta della codifica

Nel problema PATH, qual è la codifica di G? Ce ne sono molte, ma noi con la notazione "$\langle\ \rangle$" intendiamo scegliere una codifica **ragionevole**. Con ragionevole si intende una codifica che permette di codificare e decodificare in una rappresentazione interna naturale in un tempo **polinomiale**.
Esempio: $\langle G \rangle =$ **matrice di adiacenza**, dove la complessità è in funzione di:

- n = numero di vertici (V)
- m = numero di archi (E)

Quindi $O(n) \leq |\langle G \rangle| \leq O(n^2)$.

11.8.2 2col

Si consideri il seguente linguaggio:

$$2\text{col} = \{\langle G \rangle : G = (V, E) \text{ è un grafo 2 colorabile}\}$$

11.8.2.1 Teorema - 2col $\in$ P

2col $\in$ P.

Idea: coloro ogni nodo di rosso o blu in modo tale che non c'è nessun arco che collega due nodi dello stesso colore. Ancora una volta, il primo algoritmo che potrebbe venire in mente è quello di brute-force dove si considerano **tutte le possibili** 2 colorazioni, andando a verificare se ne **esiste una** che non contraddica la regola. Il problema con questo algoritmo è il **numero totale di 2 colorazioni** è 2^n, in quanto per ogni nodo ho **due possibilità**, o rosso o blu e **controllare** se la colorazione è valida richiede $O(n)$, richiedendo un tempo totale di $O(n \cdot 2^n)$. Quindi

$$2\text{col} \in \text{EXP}$$

Invece di provarle tutte e verificare se ne esiste una che vada bene, un algoritmo **polinomiale** sarebbe quello di iniziare a colorare e fermarsi alla prima contraddizione della regola, in quanto significherebbe che il grafo non è 2 colorabile.

Dimostrazione: Algoritmo **polinomiale** per 2col:

1. Su input $\langle G \rangle$ dove G è un grafo
2. Per ogni componente connessa di G ripeti i seguenti punti:
   (a) Prendi il nodo v e coloralo di rosso
   (b) Colora i vicini di v di blu
   (c) Colora i vicini dei vicini di v di rosso
   (d) Se c'è un contraddizione con la regola, **rifiuta**

Questo algoritmo ha un **tempo polinomiale** nel numero di archi e di nodi del grafo, scritto anche come poly(n, m). Quindi

$$2\text{col} \in \text{P}$$

11.8.3 3col

Si consideri il seguente linguaggio:

$$3\text{col} = \{\langle G \rangle : G = (V, E) \text{ è un grafo 3 colorabile}\}$$

È noto che 3col $\in$ EXP, ma ancora non si è riuscito a trovare un modo per far sì che 3col $\in$ P. Qualcuno nel 2005 ha trovato che 3col $\in$ DTIME$(1.333^n) \subset$ EXP.

11.8.4 Longest Common Subsequence - LCS

La **Longest Common Subsequence** (LCS) funziona nel seguente modo:

- Prende in input due stringe $x, y \in \Sigma^n$
- Restituisce la più lunga sottosequenza di caratteri in comune tra x e y non adiacente

  Esempio:

$$\begin{array}{rcl} x & = & \text{\underline{A}LGORI\underline{T}HM} \\ y & = & \text{AT\underline{A}V\underline{I}S\underline{T}IC} \end{array}$$

  In questo caso si ha LCS=3

Anche in questo caso l'algoritmo brute-force è quello di generare tutte le possibili sottosequenze e verificarne la validità. In questo caso si hanno 2^n sottosequenze totali e controllare la validità di una LCS richiede tempo $O(n)$, richiedendo un tempo totale di $O(n \cdot 2^n)$. Quindi

$$\text{LCS} \in \text{EXP}$$

Grazie alla programmazione dinamica però è stato trovato un algoritmo che calcola la LCS in $O(n^2)$ e quindi

$$\text{LCS} \in \text{P}$$

Non è però ancora noto un algoritmo che calcoli LCS in $O(n)$.

11.8.5 CLIQUE

Si consideri il seguente linguaggio:

$$\text{CLIQUE} = \{\langle G \rangle : G \text{ è un grafo e contiene almeno un triangolo}\}$$

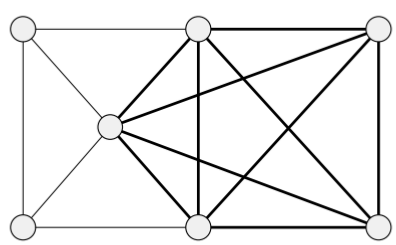


Figure 51: Esempio di CLIQUE, in questo caso ci sono 5 triangoli

Il numero di triangoli in un grafo non orientato è $O(n^3)$ e controllare la correttezza costa $O(1)$, quindi

$$\text{CLIQUE} \in \text{P}$$

Non si sa però se c'è un algoritmo che risolve il problema in $O(n^2)$. I migliori trovati fin'ora sono $O(n^{2.39})$ e $O(\frac{n^3}{\log^2 n})$.
Varianti di CLIQUE:

- 4-CLIQUE $\in$ DTIME(n^4), ma non si sa se $O(n^3)$
- 5-CLIQUE $\in$ DTIME(n^5) (esempio nella figura 51)
- k-CLIQUE $\in$ DTIME($n^k \cdot k^2$), ma se $k = \frac{n}{2}$ allora k-CLIQUE $\in$ EXP e non si sa se sta in P

11.9 Problemi SAT

I problemi SAT (Satisfiability) si concentrano sul determinare l'esistenza di un'interpretazione che soddisfa una data formula booleana. Ne esistono tante varianti:

- CIRCUIT-SAT
- FORMULA-SAT
- CNF-SAT
- K-SAT
- 3-SAT

Alcuni fatti:

- È facile vedere che 3-SAT $\in$ EXP, infatti 3-SAT $\in$ DTIME(1.34^n)
- 4-SAT $\in$ DTIME($1.5^n \cdot \text{poly}(n)$)
- 5-SAT $\in$ DTIME($1.6^n \cdot \text{poly}(n)$)
- k-SAT $\in$ DTIME($(2 - \frac{2}{k})^n \cdot \text{poly}(n)$)

La domanda che ci poniamo è: SAT $\in$ P?
Vediamo prima tutte le varianti.

11.9.1 CIRCUIT-SAT

Circuito booleano dove si hanno delle variabili di input $x_1, x_2, \ldots, x_n$ collegate tramite porte $\wedge, \vee, \neg$:

$$\text{CIRCUIT-SAT} = \{\langle C, x\rangle : \exists\, x \in \{0,1\}^* \text{ tale che } C(x) = 1\}$$

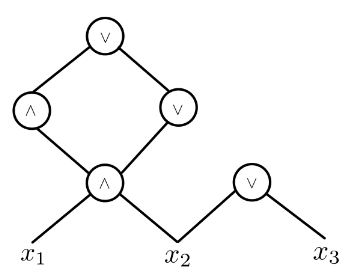


Figure 52: Esempio di circuito CIRCUIT-SAT

Dove:

- Il numero di porte è pari ad un numero m
- Il numero di variabili in input è $n \leq O(m) \leq |\langle C \rangle|$
- Il numero di cavi in ingresso (fan-in) per ogni porta è:
  - $\wedge$ ha 2 cavi in ingresso
  - $\vee$ ha 2 cavi in ingresso
  - $\neg$ ha 1 cavo in ingresso
- Il numero di cavi in uscita (fan-out) di ogni porta può essere qualsiasi

La risoluzione di CIRCUIT-SAT corrisponde dunque a trovare una funzione $f_C : \{0,1\}^n \to \{0,1\}$, ovvero una funzione che ha come dominio una lista di 0 e 1 di dimensione pari al numero di variabili di input del circuito C e come codominio 0 o 1, che stanno ad indicare rispettivamente l'**insoddisfacibilità** o la **soddisfacibilità** del circuito.
Valutare se un circuito è soddisfacibile equivale al problema chiamato

$$\text{CIRCUIT-EVAL} = \{\langle C, x \rangle : C(x) = 1\}$$

che è noto essere in P. Trovare tale x che rende il circuito soddisfacibile però è molto più complesso, infatti richiede circa $O(2^n \cdot \text{poly}(n))$ passi e quindi:

$$\text{CIRCUIT-SAT} \in \text{EXP}$$

Non è noto alcun algoritmo per cui CIRCUIT-SAT $\in$ P.

11.9.2 FORMULA-SAT

Esattamente come CIRCUIT-SAT, con l'unica differenza che le porte hanno fan-out=1.

11.9.3 CNF-SAT

Prende "$\wedge$" di clausole, dove le clausole sono "$\vee$" di letterali.
Esempio:

$$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_7) \wedge \ldots$$

Dove:

- m = numero di clausole
- n = numero di variabili

11.9.4 k-SAT

Equivalente a CNF-SAT ma le clausole contengono un numero di letterali minore o pari a k

11.9.5 3-SAT

Equivalente a k-SAT dove $k = 3$

11.10 Teorema - 2-SAT $\in$ P

2-SAT $\in$ P.

<u>Dimostrazione</u>: sia $\phi(x_1, \ldots, x_n)$ una **formula** con n variabili ed m clausole contenenti al più 2 letterali. Ad ogni clausola $x \vee y$ è associata un'implicazione logica del tipo $\overline{x} \to y$ e $\overline{y} \to x$.
<u>Esempio</u>: sia $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2) \wedge (x_3 \vee x_2) \wedge (\overline{x_3})$. Le implicazioni saranno:

x	y	$x \vee y$	$x \to y$	$\overline{x} \to y$	$\overline{y} \to x$
0	0	0	1	0	1
0	1	1	1	1	1
1	0	1	0	1	1
1	1	1	1	1	0

Per ogni clausola $a \vee b$ in ϕ costruisco il grafo con vertici $\{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\}$ aggiungendo gli archi $\overline{a} \to b$ e $\overline{b} \to a$.
<u>Esempio</u>: utilizzando la ϕ di sopra, si ha:

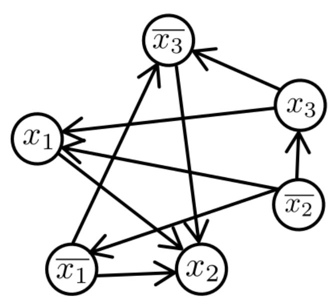


Figure 53: Costruzione del grafo corrispondente alla formula ϕ

Gli archi sono stati aggiunti tramite la seguente logica:

- $(x_1 \vee x_2)$ mi fa aggiungere gli archi: $\overline{x_1} \to x_2$ e $\overline{x_2} \to x_1$
- $(x_1 \vee \overline{x_3})$ mi fa aggiungere gli archi: $\overline{x_1} \to \overline{x_3}$ e $x_3 \to \overline{x_1}$
- $(\overline{x_1} \vee x_2)$ mi fa aggiungere gli archi: $x_1 \to x_2$ e $\overline{x_2} \to \overline{x_1}$
- $(x_3 \vee x_2)$ mi fa aggiungere gli archi: $\overline{x_3} \to x_2$ e $\overline{x_2} \to x_3$
- $(\overline{x_3})$ mi fa aggiungere gli archi: $x_3 \to \overline{x_3}$ e $\overline{\overline{x_3}} \to \overline{x_3}$ che sono però **equivalenti**.

Osserviamo che se $\exists x_i \to x_j \Rightarrow \overline{x_j} \to \overline{x_i}$. $(*)$
Lemma: ϕ è **soddisfacibile** se e solo se nessuna componente fortemente connessa di G (ogni vertice è raggiungibile da ogni altro vertice) contiene una variabile e la sua negazione.
Dimostrazione: due versi da dimostrare

- $\Rightarrow$ Sia ϕ **soddisfacibile** e consideriamo un arco $a \to b$. Questo arco corrisponde ad $\overline{a} \vee b$, quindi se $a = 1$, anche b deve esserlo, in quanto $0 \vee 1 = 1$.

  Consideriamo il cammino $x \rightsquigarrow \overline{x}$. Allora è facile vedere che $x = 0$.

  Consideriamo il cammino $\overline{x} \rightsquigarrow x$. Allora è facile vedere che $x = 1$.

  Ma questo è **impossibile** perché x non può valere sia 0 che 1 allo stesso tempo. Questo significa che non possono esserci entrambi i cammini e quindi nessuna componente fortemente connessa può contenere sia il cammino $x \rightsquigarrow \overline{x}$ che il cammino $\overline{x} \rightsquigarrow x$.

- $\Leftarrow$ Assumiamo che nessuna componente fortemente connessa contiene x_i e $\overline{x_i}$.

  Possiamo ordinare le componenti fortemente connesse $C_1, \ldots, C_m$ in **ordine topologico** con il seguente assegnamento:

  - Per ogni x, scelgo $x = 1$ se e solo se x appare dopo $\overline{x}$
  - $x = 0$ altrimenti

  Affermazione: per nessun arco $a \to b$ di G si può avere $a = 1$ e $b = 0$ (in quanto $1 \to 0 = 0$). Questo significa che l'assegnamento soddisfa ϕ in quanto se c'è una clausola $x \vee y$ tale che $x = y = 0$, allora avremo un arco $\overline{x} \to y$ tale che $\overline{x} = 1$ e $y = 0$.

  Dimostrazione (affermazione): supponiamo che non sia vero e quindi che ci sia un arco $a \to b$ tale che $a = 1$ e $b = 0$. Supponiamo che a si trovi nella componente generica C_i.
  Siccome c'è l'arco $a \to b$, c'è la clausola $\overline{a} \vee b$ e c'è anche l'arco $\overline{a} \to \overline{b}$ (per l'osservazione $(*)$). Siccome $a = 1$ allora $\overline{a} = 0$ e quindi $\overline{a}$ si trova nella componente C_j con $j < i$. D'altra parte, siccome $b = 0$ e $\overline{b} = 1$, si ha che $\overline{b}$ si trova nella componente C_k con $k > i$. Ciò implica che l'arco $a \to b$ **contraddice** l'ordine topologico della componente fortemente connessa.

11.11 NP

Il termine **NP** significa *Nondeterministic Polynomial time*.
Ci sono due definizioni equivalenti della classe NP, iniziamo con la prima introducendo il concetto di **verificatore**.
Per molti problemi esistono un numero **esponenziale** di soluzioni che possono essere **enumerate** e **verificate**. Alcuni esempi sono:

- **st-PATH**:
  - Soluzione: sequenza di nodi $v_1, \ldots, v_l$

- Verifica: $s = v_1, t = v_l$ con $(v_i, v_j) \in E$ (tutti gli archi devono essere archi validi nel grafo)

- **hamiltonian-PATH**: un hamiltonian path in un grafo è un cammino che attraversa ogni nodo del grafo esattamente una volta, partendo da un nodo s e arrivando ad un nodo t.
  - Soluzione: sequenza di nodi $v_1, \ldots, v_l$
  - Verifica: $s = v_1, t = v_l$ con $(v_i, v_j) \in E$ e $V = (v_1, \ldots, v_l)$ (non ci sono ripetizioni nei nodi attraversati)

- **3col**:
  - Soluzione: $c_1, \ldots, c_n$ con $c_i \in \{R, G, B\}$
  - Verifica: $\forall (v_i, v_j) \in E,\ c_i \neq c_j$ (non ci sono archi che collegano due nodi dello stesso colore)

Osservazione: le soluzioni si possono codificare come **stringhe di lunghezza polinomiale** ed esiste un **algoritmo efficiente** per **verificarne una**.
I linguaggi nella classe NP hanno tutti la caratteristica sopra citata.
Nota: sembrerebbe che non tutti i linguaggi abbiano tale caratteristica, come ad esempio $\overline{\text{hamiltonian-PATH}}$, in quanto definire che un grafo non abbia un hamiltonian path richiede l'utilizzo dello stesso algoritmo esponenziale per determinare se ne esiste uno (**spoiler**: NP non è chiuso rispetto al complemento).

11.11.1 Definizione - NP

NP è la classe di linguaggi L tali che L ammette **verificatore** con tempo polinomiale.

Tutti i problemi in questa classe hanno una caratteristica comune chiamata ***verificabilità polinomiale***. Questo significa che anche se non si conoscono algoritmi veloci per determinare una soluzione a questi problemi, è facile però verificare se una data soluzione ottenuta in un qualsiasi modo risolve il problema semplicemente mostrandola. In altri termini, **verificare** se una data soluzione risolve un problema è estremamente più facile di determinarne l'esistenza.

11.11.2 Verificatore

Un **verificatore** per un linguaggio L è una TM $V(\langle x, y\rangle)$ dove $\langle x\rangle$ è l'input ed $\langle y\rangle$ è la **soluzione candidata** (o **certificato**). Inoltre V deve essere polinomiale.

- Se $x \in L$, allora esiste una y tale che $V(\langle x, y\rangle)$ accetta
- Se $V(\langle x, y\rangle)$ accetta, allora $x \in L$.

Il linguaggio L viene detto **polinomialmente verificabile**.

11.11.2.1 Definizione - Verificatore

Una TM V è un **verificatore** per un linguaggio L se:

- V ha input $\langle x, y\rangle$, dove x è la stringa del linguaggio L da verificare ed y è il **certificato di appartenenza** del linguaggio
- $\forall x, x \in L \Leftrightarrow \exists y : V(\langle x, y\rangle)$ accetta
  - $\Rightarrow$ Se $x \in L, \exists y : V(\langle x, y\rangle)$ **accetta** (chiamato **yes case**)
  - $\Leftarrow$ Se $x \notin L, \forall y\ V(\langle x, y\rangle)$ **rifiuta** (chiamato **no case**)
- V ha tempo **polinomiale**

Nota: la lunghezza di y deve essere **necessariamente polinomiale**, altrimenti V non avrebbe tempo di leggerla.
Esempi di **certificati**: un esempio di **certificato** per hamiltonian-PATH è semplicemente un hamiltonian path valido che va dal nodo s al nodo t. Un certificato per 3col è una colorazione valida del grafo.

11.12 Problemi in NP

11.12.1 SQUARES

Si consideri il linguaggio:

$$\text{SQUARES} = \{x : x = \langle B\rangle, B \in \mathbb{N}, B \text{ è un quadrato}\} = \{0, 1, 100, 1001, ...\}$$

11.12.1.1 Proposizione - SQUARES $\in$ NP

SQUARES $\in$ NP.

Dimostrazione: consideriamo l'algoritmo $V(\langle x, y\rangle)$:

1. Interpreta $x = \langle B\rangle$ con $B \in \mathbb{N}$ e $y = \langle D\rangle$ con $D \in \mathbb{N}$ tale che $2 \leq D \leq B$
2. Calcola D^2
3. **Accetta** se e solo se $D^2 = B$, altrimenti **rifiuta**

La moltiplicazione richiede $O(n^2)$ passi, dove $n = |\langle B\rangle|$ e quindi è **polinomiale**.
Inoltre V è un **verificatore**:

- **Yes case**: $x \in$ SQUARES, $x = \langle B\rangle$ tale che $B = C$ per $C \in \mathbb{N}$, quindi V accetta con $y = \langle C\rangle$
- **No case**: $V(\langle x, y\rangle)$ accetta, allora $y = \langle D\rangle$ tale che $D^2 = B$ e $B \in$ SQUARES

11.12.2 3col $\in$ NP

3col $\in$ NP.

Dimostrazione: consideriamo il verificatore $V(\langle x, y\rangle)$:

1. Interpreta $\langle x\rangle = \langle G\rangle$ e $y = \langle C_1, \ldots, C_n\rangle, C_i \in \{R, G, B\}, n = |V|$ (numero di vertici)
2. Scansiona gli archi $(v_i, v_j) \in E$ e rifiuta se e solo se $c_i = c_j$, ovvero se due nodi adiacenti hanno lo stesso colore

V è polinomiale e inoltre:

- **Yes case**: Se $G \in$ 3col allora esiste una colorazione $c_1, \ldots, c_n$ valida e quindi V accetta con $y = (c_1, \ldots, c_n)$
- **No case**: Se V accetta con y, allora y per definizione è la descrizione di una 3 colorazione di G e $G \in$ 3col

11.13 P vs NP

Riassunto di cosa sono le classi P e NP:

$$\begin{aligned} \text{P} &= \text{la classe di linguaggi } \textit{decidibili} \text{ in maniera rapida (polinomialmente)} \\ \text{NP} &= \text{la classe di linugaggi che ammettono } \textit{verificatore} \\ &\quad\ \text{in tempo polinomiale} \end{aligned}$$

La domanda è: P = NP? Non si sa.
Alcune congetture dicono che P $\neq$ NP, ma sono estremamente difficili da dimostrare. Fin'ora abbiamo visto che P $\subset$ EXP e P $\neq$ EXP.

11.13.1 Teorema - P $\subseteq$ NP $\subseteq$ EXP

P $\subseteq$ NP $\subseteq$ EXP.

Dimostrazione: mostriamo che P $\subseteq$ NP. Sia $L \in$ P un linguaggio. Dobbiamo mostrare che $L \in$ NP: siccome $L \in$ P, allora esiste una TM M tale che $x \in L$ se e solo se $M(\langle x\rangle)$ accetta. Costruiamo allora un **verificatore** $V(\langle x, y\rangle)$ per L:

1. Ignora y
2. Esegui $M(\langle x\rangle)$

V è polinomiale ed inoltre:

- **Yes case**: se $x \in L$ allora $V(x, y = \epsilon) = M(x)$ accetta
- **No case**: se $V(\langle x, y\rangle)$ accetta, anche $M(x)$ accetta e quindi $x \in L$

11.14 Caratterizzazione alternativa di NP

Vediamo ora il secondo modo di definire la classe NP:

Un linguaggio è in NP se è decidibile in tempo polinomiale da una TM non deterministica.

Ripasso: una TM non deterministica accetta se e solo se almeno un ramo di computazione è accettante. In un decisore tutti i rami terminano ed il tempo di esecuzione totale della NTM è il tempo massimo di computazione tra tutti i rami.
Esempio: FORMULA-SAT $\in$ NP. Costruisco una NTM N che decide FORMULA-SAT:
Su input $\langle \phi \rangle$ con n variabili:

1. Alloca il vettore $x[1 \dots n] = 0$ e imposta la variabile $i = 0$
2. "*Top*". Incrementa la variabile i di uno. Se $i > n$ allora vai all'etichetta "*Check*".
   goto-both "*Write 0*", "*Write 1*" (azione eseguita non deterministicamente)
3. "*Write 0*". Imposta $x[i] = 0$
   goto "*Top*"
4. "*Write 1*". Imposta $x[i] = 1$
   goto "*Top*"
5. "*Check*". Se $\phi(x_1, \dots, x_n) = 1$ **accetta**, altrimenti **rifiuta**

Analisi: ci sono 2^n rami, ma il tempo di esecuzione è polinomiale nella lunghezza della codifica della formula, ovvero poly($|\langle \phi \rangle|$). Questo significa che ogni ramo fa n passi e in più controlla se la formula è soddisfatta o meno. Inoltre $N(\langle \phi \rangle)$ accetta se e solo se ϕ è **soddisfacibile**.

11.14.1 Definizione - NTIME

$$\text{NTIME}(f(n)) = \{L \subseteq \Sigma^* : \exists \text{ NTM } N \text{ che decide } L \text{ in tempo } O(f(n))\}$$

11.14.2 Definizione - NP (alternativa)

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

11.15 Teorema - Le due definizioni di NP sono equivalenti

Le due definizioni di NP sono **equivalenti**.

Idea: mostriamo come **convertire** un **verificatore** che gira in tempo polinomiale in una **NTM** che gira sempre in tempo polinomiale equivalente e viceversa:

- La **NTM** simula il verificatore "indovinando" il certificato
- Il **verificatore** simula la NTM utilizzando il ramo accettante come certificato.

Dimostrazione:

- sia $V(\langle x, y\rangle)$ un **verificatore** per un linguaggio $L \in \text{NP}$ con tempo $\text{poly}(|x|)$, sia questo tempo $k|x|^k$ con $k \in \mathbb{N}$. Vogliamo definire una NTM N che **decide** L in tempo $\text{poly}(|x|)$.

  $N =$ "Su input x di lunghezza n:

  1. Indovina non deterministicamente y
  2. Esegui V su input $\langle x, y\rangle$ deterministicamente in ogni ramo e **accetta** se e solo se $V(\langle x, y\rangle)$ accetta."

  Osserviamo che $|y| \leq k|x|^k$ in quanto $V(\langle x, y\rangle)$ è eseguibile in tempo **polinomiale**. Se invece fosse $|y| > k|x|^k$, V non avrebbe tempo di leggerlo.

  Tempo: il tempo di esecuzione è **polinomiale**, in quanto la computazione su ogni ramo è $\text{poly}(|x|)$. Inoltre si ha che $x \in L \Leftrightarrow$ almeno un ramo è accettante $\Leftrightarrow V(\langle x, y\rangle)$ accetta per qualche y.

- Assumiamo ora che la NTM N sia il decisore del linguaggio L in tempo polinomiale e costruiamo un verificatore V come riportato di seguito. Il certificato sarà la lista delle scelte non deterministiche effettuate.

  $V =$ "Su input $\langle x, y\rangle$ dove sia x che y sono stringhe:

  1. Interpreta $y \in \{0,1\}^{k|x|^k}$
  2. Simula $N(x)$: all'i-esima "*goto-both*" usa y_i per decidere quale ramo seguire"

  Chiaramente $V(\langle x, y\rangle)$ ha tempo $\text{poly}(|x|)$. Inoltre: $x \in L \Leftrightarrow N(x)$ accetta $\Leftrightarrow$ esiste un ramo accettante $\Leftrightarrow V(\langle x, y\rangle)$ accetta per qualche y.

11.15.1 Esempio

Si consideri il linguaggio:

$$k\text{-CLIQUE} = \{\langle G, k\rangle : G \text{ è un grafo non diretto con una } k\text{-clique}\}$$

Dimostriamo che k-CLIQUE $\in$ NP.

Idea: la **clique** è il certificato
Dimostrazione: il seguente è il verificatore V per k-CLIQUE:
$V =$ "Su input $\langle\langle G, k\rangle, c\rangle$:

1. Controlla se c è un sottografo con k nodi di G

2. Controlla se G contiene tutti gli archi che connettono i nodi in c

3. Se entrambi i controlli vanno a buon fine **accetta**, altrimenti **rifiuta**"

Alternativa: pensiamo ora NP in termini di NTM a tempo polinomiale, fornendo una NTM N che decide k-CLIQUE:
$N =$ "Su input $\langle G, k\rangle$ dove G è un grafo:

1. Scegli in modo non deterministico un sottoinsieme c contenente k nodi di G

2. Controlla se G contiene tutti gli archi che connettono i nodi in c

3. Se il test va a buon fine **accetta**, altrimenti **rifiuta**"

11.16 Definizione - NEXP

$$\text{NEXP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(2^{|x|^k})$$

<u>Fatto interessante</u>: in NEXP sono presenti linguaggi che sicuramente non sono presenti in NP.

12 NP-completezza

Un grande passo avanti nella domanda P vs NP è avvenuto nel 1970 con il lavoro di *Stephen Cook* e *Leonid Levin*. I due hanno scoperto alcuni problemi in NP la quale complessità è legata a tutti gli altri problemi nella classe e trovando un algoritmo polinomiale per uno di questi, tutti i problemi in NP diventerebbero risolvibili in tempo polinomiale.
Questi linguaggi vengono chiamati **NP-completi**.
Un esempio di linguaggio NP-completo è 3-SAT, quindi se riuscissimo a trovare un algoritmo polinomiale per 3-SAT, allora avremmo che 3-SAT $\in$ P, ma siccome 3-SAT è NP-completo, avremmo che tutti i linguaggi in NP sono in P e quindi P=NP.

12.1 Definizione - Poly-time mapping reduction

Siano A e B due linguaggi. A ha una **poly-time mapping reduction** a B (scritto come $A \leq_m^p B$) se esiste una funzione $R : \{0,1\}^* \to \{0,1\}^*$ calcolabile in tempo polinomiale da una TM tale che

$$w \in A \Leftrightarrow R(x) \in B \quad \forall x \in \{0,1\}^*$$

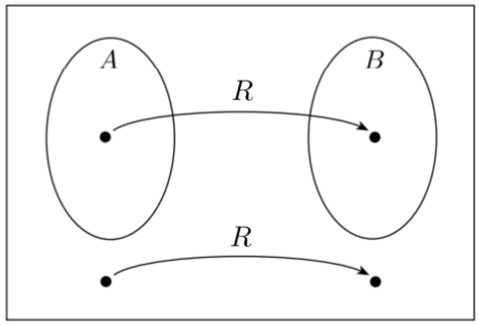


Figure 54: Funzione poly-time R che riduce A a B

Come una normale mapping reduction che trasforma problemi di A in problemi di B, ma questa volta la conversione viene fatta **efficacemente**. Se un linguaggio è riducibile poly-time ad un linguaggio che è già noto sia risolvibile in tempo polinomiale, allora si ottiene automaticamente la soluzione del problema originale anch'essa con tempo polinomiale.

12.2 Teorema - 4-COL $\leq_m^p$ SAT

4-COL $\leq_m^p$ SAT. Questo implica che se SAT $\in$ P allora anche 4-COL $\in$ P.

Dimostrazione: dato un grafo $G = (V, E)$ con $|V| = n$, vogliamo codificare l'affermazione che "*G è 4 colorabile*" con una formula ϕ_G tale che

$$\text{" } G \text{ è 4 colorabile"} \Leftrightarrow \text{"}\phi_G \text{ è soddisfacibile"}$$

Le variabili di ϕ_G saranno: $x_1, x_1', x_2, x_2', \ldots, x_n, x_n'$ con il seguente significato:

x_i	x_i'	colore
F	F	R
F	T	B
T	F	Y
T	T	W

Ovvero ad ogni nodo del grafo G vengono assegnate due variabili che ne indicano il colore.
Una colorazione valida è la seguente:

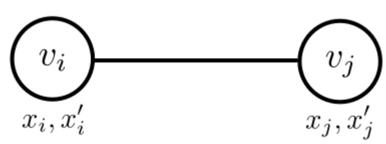


Figure 55: Colorazione valida per 4-col

dove $(x_i, x'_i) \neq (x_j, x'_j)$. Questo succede solo nel caso in cui **non è vero che entrambi i nodi hanno lo stesso colore**, ovvero in formule:

$$\neg((x_i \Leftrightarrow x_j) \wedge (x'_i \Leftrightarrow x'_j))$$

Ricordiamo che

$$\begin{aligned} p \Leftrightarrow q &\equiv (p \Rightarrow q) \wedge (q \Rightarrow p) \\ &\equiv (\neg p \vee q) \wedge (\neg q \vee p) \end{aligned}$$

Questo significa che l'operatore "$\Leftrightarrow$" può essere scritto in termini di $\neg, \wedge$ e $\vee$ e quindi è una formula valida per SAT.
Diciamo che $\neg((x_i \Leftrightarrow x_j) \wedge (x'_i \Leftrightarrow x'_j))$ corrisponde ad una sottoformula ϕ_{ij}. Quindi l'intero problema si può risolvere nel seguente modo:

$$(x_i, x'_i) \neq (x_j, x'_j) \Rightarrow \phi_G = \bigwedge_{(i,j) \in E} \phi_{ij}$$

Infine, si ha che $|\langle \phi_G \rangle| = O(m \log n) = \text{poly}(|\langle G \rangle|)$, dove $\log n$ è il numero di bit utilizzati per ogni vertice.

12.3 Teorema - Se $A \leq_m^p B$ e $B \in$ P, allora $A \in$ P.

Se $A \leq_m^p B$ e $B \in$ P, allora $A \in$ P.

Dimostrazione: sia M_B la TM che decide B in tempo polinomiale ed R la poly-time reduction da A a B. Abbiamo che $M_B(R(x))$ decide $x \in A$ in tempo poly(poly(n)) = poly(n).

12.4 Teorema - $\leq_m^p$ è transitiva

$\leq_m^p$ è transitiva:

$$A \leq_m^p B \text{ e } B \leq_m^p C \Rightarrow A \leq_m^p C$$

Dimostrazione: sia R_1 una poly-time mapping reduction da A a B e R_2 una poly-time mapping reduction da B a C. Definisco une funzione R poly-time mapping reduction come $R(x) = R_2(R_1(x))$. Avremo quindi:

$$x \in A \Leftrightarrow R_1(x) \in B \Leftrightarrow R_2(R_1(x)) \in C \Leftrightarrow R(x) \in C$$

12.5 Teorema - 3col $\leq_m^p$ 4col

3col $\leq_m^p$ 4col.

Dimostrazione: dobbiamo progettare una funzione R poly-time mapping reduction tale che $R(\langle G \rangle) = \langle H \rangle$, dove G è un grafo 3col e H è un grafo 4col. Diciamo che

$$G \text{ è 3 colorabile} \Leftrightarrow H \text{ è 4 colorabile}$$

Idea per H: aggiungiamo un nuovo nodo, collegandolo a tutti i nodi di G. Tempo: poly(m).

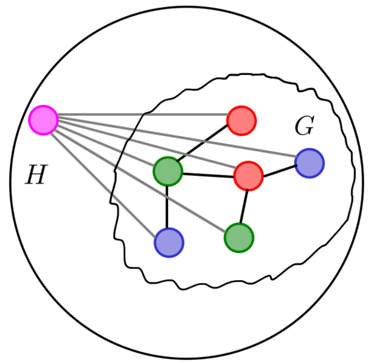


Figure 56: 4 colorazione partendo da una 3 colorazione

Correttezza: data una 3 colorazione per G, otteniamo una 4 colorazione per H utilizzando la stessa 3 colorazione aggiungendo però un nuovo colore per il nuovo nodo.
Se H è 4 colorabile allora il nuovo nodo deve avere per forza un colore diverso da tutti gli altri e quindi G è 3col.

12.5.1 Riduzioni utilizzate come "oracolo"

Le riduzioni possono essere più generali, ad esempio utilizzando le TM come **oracolo**. Un esempio può essere il seguente:

$$\text{4-CHROMA} = \{\langle G \rangle : G \text{ è un grafo e } \chi(G) = 4\}$$

dove $\chi(G) = 4$ sta a significare che G è 4 colorabile ma non 3 colorabile.
Utilizziamo la seguente nuova riduzione:

$$\text{4-CHROMA} \leq_T^p \text{SAT}$$

dove la T sta per Turing machine, che è una forma di riduzione più generale.
Dato $\langle G \rangle$ consideriamo la seguente riduzione:

- Usa un oracolo SAT su $R_{\text{4col-SAT}}(\langle G \rangle)$. Se l'output è reject, **rifiuta**
- Usa un oracolo SAT su $R_{\text{3col-SAT}}(\langle G \rangle)$. Se l'output è reject **accetta**, altrimenti **rifiuta**

L'oracolo nel primo caso accetta se e solo se il grafo è 4 colorabile, mentre nel secondo caso accetta se e solo se il grafo non è 3 colorabile, che è proprio la proprietà che cercavamo.

12.6 Teorema - Se $A \leq_m^p B$ e $B \in$ NP, allora $A \in$ NP

Se $A \leq_m^p B$ e $B \in$ NP, allora $A \in$ NP.

Dimostrazione: siccome $A \leq_m^p B$, allora esiste una funzione R poly-time mapping reduction tale che $x \in A \Leftrightarrow R(x) \in B$. Inoltre, siccome $B \in$ NP, esiste una NTM N_B che decide B (per definizione di appartenenza a NP). Vogliamo quindi una NTM N_A che decide A, quindi $N_A(x) = N_B(R(x))$. Avremo quindi:

$$x \in A \Leftrightarrow x' = R(x) \in B \Leftrightarrow N_B(x') \text{ accetta} \Leftrightarrow N_A(x) \text{ accetta}$$

12.7 Teorema - SAT $\leq_m^p$ CIRCUIT-SAT

SAT $\leq_m^p$ CIRCUIT-SAT.

La dimostrazione è banale in quanto una formula booleana è equivalente ad un circuito.

12.8 Teorema - CIRCUIT-SAT $\leq_m^p$ 3SAT

CIRCUIT-SAT $\leq_m^p$ 3SAT.

Dimostrazione: sia ϕ_C una formula 3SAT tale che

$$\exists x : C(x) = 1 \Leftrightarrow \exists w : \phi_C(w) = 1$$

ovvero esiste un assegnamento x che rende vero il circuito C se e solo se esiste un assegnamento w che rende vera la formula ϕ_C.

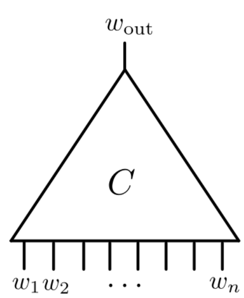


Figure 57: Esempio di circuito in CIRCUIT-SAT

Idea: usare una variabile per ogni cavo del circuito C, quindi gli ingressi $w_1, w_2, \ldots, w_n$ del circuito diventano i letterali $x_1, x_2, \ldots, x_n$ della formula 3SAT.
Vediamo ora come trasformare le varie porte CIRCUIT-SAT nelle rispettive formule in 3SAT:

- Porta "$\neg$":

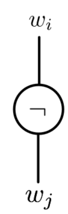


Figure 58: Porta NOT in CIRCUIT-SAT

Diventa: $w_j \Leftrightarrow \overline{w_j} \equiv (w_i \vee w_j) \wedge (\overline{w_i} \vee \overline{w_j})$. Questo perché $x \Leftrightarrow y \equiv (\overline{x} \vee y) \wedge (\overline{y} \wedge x)$.

- Porta "$\wedge$":

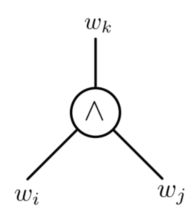


Figure 59: Porta AND in CIRCUIT-SAT

Diventa:

$$\begin{aligned}((\overline{w_i} \wedge \overline{w_j}) \Rightarrow \overline{w_k}) \wedge ((\overline{w_i} \wedge w_j) \Rightarrow \overline{w_k}) \wedge \\ \wedge ((w_i \wedge \overline{w_j}) \Rightarrow \overline{w_k}) \wedge \\ \wedge ((w_i \wedge w_j) \Rightarrow w_k)\end{aligned}$$

che riportati in notazione con sole porte $\wedge, \vee$ e $\neg$:

$$\begin{aligned}(w_i \vee w_j \vee \overline{w_k}) \wedge (w_i \vee \overline{w_j} \vee \overline{w_k}) \wedge \\ \wedge (\overline{w_i} \vee w_j \vee \overline{w_k}) \wedge \\ \wedge (\overline{w_i} \vee \overline{w_j} \vee w_k)\end{aligned}$$

Ricordiamo che $a \Rightarrow b \equiv \overline{a} \vee b$ e che $\overline{a \wedge b} \equiv \overline{a} \vee \overline{b}$ (per la legge di De Morgan).

- Porta "$\vee$":

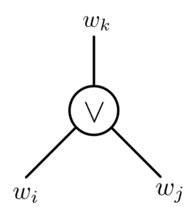


Figure 60: Porta OR in CIRCUIT-SAT

Diventa:

$$\begin{aligned}((\overline{w_i} \wedge \overline{w_j}) \Rightarrow \overline{w_k}) &\wedge ((\overline{w_i} \wedge w_j) \Rightarrow w_k)\wedge \\ &\wedge ((w_i \wedge \overline{w_j}) \Rightarrow w_k)\wedge \\ &\wedge ((w_i \wedge w_j) \Rightarrow w_k)\end{aligned}$$

che riportati in notazione con sole porte $\wedge, \vee$ e $\neg$:

$$\begin{aligned}(w_i \vee w_j \vee \overline{w_k}) &\wedge (w_i \vee \overline{w_j} \vee w_k)\wedge \\ &\wedge (\overline{w_i} \vee w_j \vee w_k)\wedge \\ &\wedge (\overline{w_i} \vee \overline{w_j} \vee w_k)\end{aligned}$$

12.8.1 Osservazioni

Con i teoremi 12.2, 12.5, 12.7, 12.8 abbiamo visto che:

$$\text{3col} \leq_m^p \text{4col} \leq_m^p \text{SAT} \leq_m^p \text{CIRCUIT-SAT} \leq_m^p \text{3SAT}$$

Importante: tutti questi problemi sono **equivalenti** e se uno di questi è in P (quindi si trova una TM che risolve uno di questi problemi in tempo polinomiale), allora tutti i problemi NP sono in P.

12.9 Teorema - Teorema di Cook-Levin

$\forall L \in$ NP, $L \leq_m^p$ SAT.

12.9.1 Corollario - Se SAT $\in$ P, allora P=NP

Se SAT $\in$ P, allora P=NP.

Possiamo pensare a SAT come il "*super*" problema NP, nel senso che tutte le difficoltà di tutti i problemi in NP sono direttamente collegate in SAT.

12.10 Definizione - NP hard

Un linguaggio S è **NP-hard** se $\forall L \in$ NP, $L \leq_m^p S$.

Quindi 3col, 4col e 3SAT sono NP-hard.
Nota: NP-hard non significa che $S \in$ NP.

12.11 Definizione - NP-completezza

Un linguaggio S è **NP-completo** se e solo se

1. S è **NP-hard**, ovvero se ogni altro linguaggio in NP è riducibile in tempo polinomiale ad S
2. $S \in$ NP

12.11.1 Teorema - Se un linguaggio S è NP-completo, allora $S \in$ P se e solo se P=NP

Se un linguaggio S è NP-completo, allora $S \in$ P se e solo se P=NP.

<u>Dimostrazione</u>: due versi da dimostrare

- $\Rightarrow$ $\forall L \in$ NP, $L \leq S$. Se $S \in$ P allora $L \in$ P perché L è riducibile ad S in tempo polinomiale. Quindi P=NP
- $\Leftarrow$ Se P=NP e S è NP-completo, allora $S \in$ NP. Quindi $S \in$ P.

12.12 Dimostrazione del Teorema di Cook-Levin

Sia un linguaggio $A \in$ NP, mostriamo che $A \leq$ SAT, ovvero che esiste una NTM N tale che N decide A in tempo n^k (polinomiale).
Considero il **tableau** di dimensioni $n^k \times n^k$ di N:

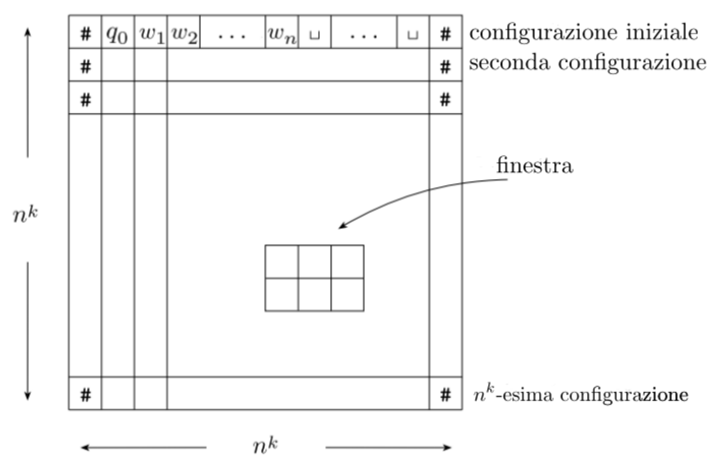


Figure 61: Un tableau è una tabella $n^k \times n^k$ di configurazioni

Un **tableau** di N per w è una tabella di dimensioni $n^k \times n^k$ le quali righe sono configurazioni dei rami di computazione di N su input w.

La prima riga del tableau è la **configurazione iniziale** di N su w ed ogni riga segue la precedente in base alla funzione di transizione di N. Il tableau è **accettante** se una qualsiasi riga è in una configurazione accettante.
Ogni **tableau accettante** per N su w corrisponde ad un **ramo di computazione accettante** di N su w, quindi ora il problema di determinare se N accetta w equivale al problema di determinare se esiste un tableau accettante per N su input w.
Passiamo ora alla descrizione della poly-time reduction da A a SAT.
Idea: costruiamo una formula SAT ϕ tale che:

$$\phi \text{ è soddisfacibile} \Leftrightarrow \text{esiste un tableau valido accettante}$$

Su input w, la riduzione produce la formula ϕ.
Descrizione di ϕ: la NTM N ha Q come insieme degli stati e Γ come alfabeto di nastro.

- Definiamo i **simboli del tableau** come $C = \Gamma \cup Q \cup \{\#\}$
- Ognuna delle $(n^k)^2$ caselle del tableau viene chiamata **cella**, la cella che si trova nella riga i e nella colonna j viene chiamata $cell[i, j]$ e contiene un simbolo s in C
- Per ogni i e j tra 1 e n^k e per ogni $s \in C$ si ha una variabile $x_{i,j,s}$ nella formula ϕ
- Ogni variabile $x_{i,j,s} = 1 \Leftrightarrow cell[i, j] = s$

Ora realizziamo ϕ in modo che un assegnamento che la soddisfa corrisponde ad un **tableau accettante** per N su input w: ϕ è un AND di quattro parti:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

Vediamole nel dettaglio:

- ϕ_{cell}: un assegnamento pone ad 1 una variabile per cella

  $$\phi_{\text{cell}} = \bigwedge_{i,j \in [n^k]} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigvee_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

  Che significa che ogni cella deve contenere **almeno** un simbolo e **solo** un simbolo
- ϕ_{start}: la prima riga contiene la configurazione iniziale di N su input w

  $$x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k,\#}$$
- ϕ_{accept}: c'è una configurazione accettante nel tableau

  $$\bigvee_{i,j \in [n^k]} x_{i,j,q_{\text{accept}}}$$

- ϕ_{move}: corrisponde al fatto che ogni riga segue dalla precedente in accordo alla δ di N. Questo viene fatto assicurandosi che ogni finestra di 2×3 celle è **legale**. Una finestra è **legale** se non viola le azioni specificate dalla δ di N

$$\bigwedge_{i,j \in [n^k]} (\text{la finestra } (i,j) \text{ è legale})$$

dove "la finestra (i,j) è legale" corrisponde a:

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{finestra legale}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j+1,a_5} \wedge x_{i+1,j+1,a_6})$$

dove con (i,j) viene considerato il simbolo che si trova nella cella in mezzo della prima riga, come mostrato nella seguente figura:

$(i,j-1)$	(i,j)	$(i,j+1)$
$(i+1,j-1)$	$(i+1,j)$	$(i+1,j+1)$

Figure 62: Coordinate di una finestra

Esempio: siano $a, b, c \in \Gamma$ e $q_1, q_2 \in Q$ e nella funzione di transizione si ha

$$\delta(q_1, a) = \{(q_1, b, R)\}$$
$$\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$$

Esempi di **finestre legali sono**:

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Figure 63: Finestre legali

Le finestre (a) e (b) sono legali in quanto la δ permette ad N di muoversi nei modi indicati. Anche la finestra (c) è legale in quanto siccome q_1 è nell'angolo in altro a destra della finestra, non sappiamo quale simbolo sta leggendo. (d) è legale in quanto le righe sono identiche. Ciò accade quando la testina non è adiacente alla posizione della finestra. (e) è legale in quanto q_1 che legge b potrebbe accadere subito alla destra della prima riga, muovendo la testina a sinistra e andando nello stato q_2 come si vede nella seconda riga. (f) è legale in quanto lo stato q_1 potrebbe trovarsi subito alla sinistra della prima riga, scambiando la b con c e successivamente muoversi a sinistra.

Esempi di **finestre non legali**:

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_2	a	a

(c)

b	q_1	b
q_2	b	q_2

Figure 64: Finestre non legali

La finestra (a) non è legale in quanto il simbolo centrale della prima riga non può cambiare in quanto non ha nessuno stato adiacente. La finestra (b) non è legale in quanto la δ specifica che b viene cambiato con una c e non una a. La finestra (c) non è legale in quanto appaiono due stati nella riga sotto.

Analizziamo ora la complessità della riduzione per mostrare che opera in tempo polinomiale. Lo facciamo esaminando la dimensione di ϕ.

- Il tableau ha $n^k \times n^k$ celle ed ogni cella ha l variabili associate, dove l è il numero di simboli presenti in C. Siccome l dipende solamente dalla NTM N e non dalla lunghezza dell'input, il numero totale di variabili è $\underline{O(n^{2k})}$
- Stimiamo ora le dimensioni di ognuna delle parti di ϕ:
  - ϕ_{cell} contiene un frammento di dimensione prefissata della formula per ogni cella del tableau, quindi ha dimensioni $\underline{O(n^{2k})}$
  - ϕ_{start} contiene un frammento per ogni cella della prima riga, quindi la sua dimensione è $\underline{O(n^k)}$
  - ϕ_{move} e ϕ_{accept} contengono entrambe un frammento di dimensione prefissata per ogni cella del tableau, quindi la loro dimensione è $\underline{O(n^{2k})}$

Quindi la dimensione totale di ϕ è $\underline{O(n^{2k})}$ e quindi è **polinomiale**, quello che volevamo.

Con questa dimostrazione abbiamo dimostrato che SAT è NP-completo. Quindi, per dimostrare che un linguaggio è NP-completo basta mostrare una poly-time reduction a SAT, o meglio, 3SAT in quanto solitamente è più semplice.

12.13 Cosa accadrebbe se P=NP

Se fosse vero che P=NP, sarebbe facile fare "*search to decision*", ovvero **cercare** una soluzione del problema.
Esempio: se fosse vero CIRCUIT-SAT $\in$ P, allora esisterebbe una TM M_{CSAT} in tempo polinomiale per un circuito C tale che:

$$M_{\text{CSAT}} = \begin{cases} 1 & \text{se } C \text{ è soddisfacibile} \\ 0 & \text{altrimenti} \end{cases}$$

Come trovare l'assegnamento?

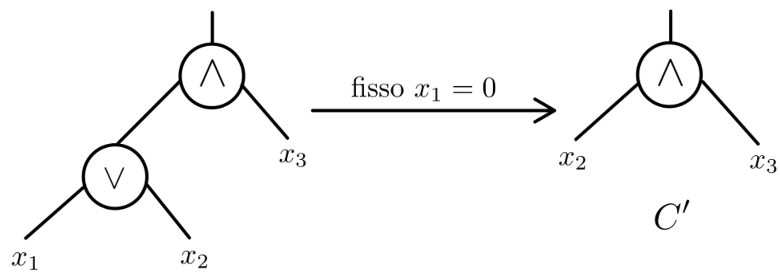


Figure 65: Ricerca di un assegnamento

Ora se $M_{\text{CSAT}}(C') = 1$ esisterebbe un assegnamento valido con $x_1 = 0$.
Questo significa che CIRCUIT-SAT è **self-reducible**. Questo vale anche per 3COL.

12.13.1 Teorema - *Self-reducibility*

Sia consideri un linguaggio $L \in$ P. Se P=NP con verificatore $V(\langle x, y\rangle)$ allora esiste una TM poly-time S tale che $\forall x \in L$ $S(x) = y$ con $V(\langle x, y\rangle) = 1$. In altri termini, oltre a verificare l'appartenenza della stringa nel linguaggio, riusciamo anche a trovarla in tempo polinomiale.

Fatto: sia M un TM con tempo $T(n)$. Esiste un circuito C con dimensioni $O(T(n^2))$ tale che $C(x) = 1$ se e solo se $M(x)$ accetta.
Posso convertire il verificatore $V(\langle x, y\rangle)$ nel circuito C e trovare la y tramite la **self-reducibility** di CIRCUIT-SAT se P=NP.

12.13.2 Teorema - P=NP $\Rightarrow$ EXP=NEXP

Se P=NP, allora anche EXP=NEXP.

Tecnica del padding (riempimento): ovviamente si ha che EXP $\subseteq$ NEXP, quindi faccio vedere ce NEXP $\subseteq$ EXP utilizzando la tecnica del padding.

Sia $L \in$ NEXP, allora esiste una NTM M che decide L in tempo 2^{n^k}, $n = |x|$. Siccome vogliamo $L \in$ EXP, definiamo:

$$L_{\text{PAD}} = \{\langle x, 1^{2^{|x|^k}} \rangle : x \in L\}$$

Claim: $L_{\text{PAD}} \in$ NP. Definisco una NTM M' per L_{PAD}:
$M' =$ "su input y:

1. Controlla che $y = x1^{2^{|x|^k}}$, assumendo che 1 non sia nel linguaggio di M'. Se non accade, **rifiuta**

2. Altrimenti esegui $M(x)$"

Analisi: il primo step richiede $O(|y|) = O(N)$, mentre il secondo step richiede $O(2^{n^k}) = O(N)$, quindi M' è **polinomiale** in $|y| = N$.
Ma allora L_{PAD} è anch'esso in P, ovvero esiste una TM A poly-time che decide L_{PAD} (questo perché stiamo assumendo che P=NP).

Claim: $L \in$ EXP. Definiamo una TM A' per L che dato x genera $(x, 1^{2^{|x|^k}}) = y$ ed esegue $A(y)$. A' decide L in tempo $O(2^{n^k}) + \text{poly}(2^{n^k}) = O(2^{n^k})$. Banale in quanto generare una stringa di lunghezza esponenziale richiede tempo esponenziale.

12.14 Teoremi di dicotomia

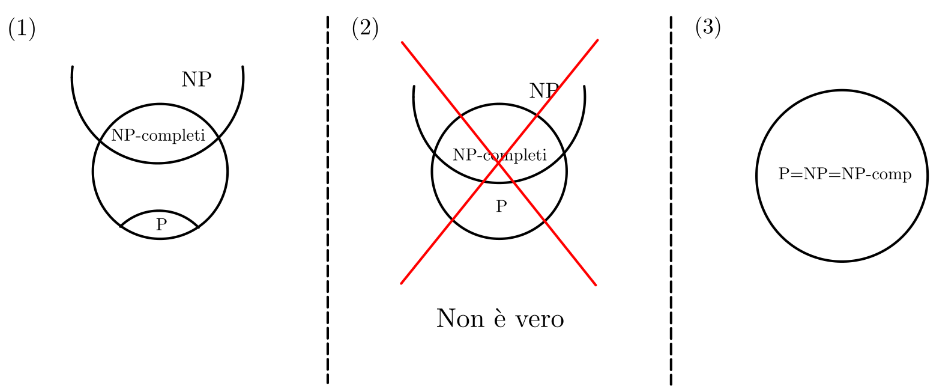


Figure 66: Teoremi di dicotomia

12.14.1 (2) Teorema

Se P≠NP esiste un linguaggio $L \in$ NP tale che $L \notin$ P ma L non è NP-completo.

12.14.2 (3)

È vero per i linguaggi **diversi** da $\emptyset$ e Σ^*. È chiaro che NP-completo è contenuto in NP (definizione 12.11, pag. 106), vediamo ora che NP $\subseteq$ NP-completo.
Sia $A \in$ NP. Per ogni linguaggio $L \in$ NP tale che $L \leq A \neq \emptyset, \Sigma^*$ esiste $x_{\text{yes}} \in A$ e $x_{\text{no}} \notin A$. Vediamo la riduzione $R_{x_{\text{yes}}, x_{\text{no}}}$:

1. Dato x, siccome $L \in$ NP=P esegui una TM per $x \in L$
2. Se la TM accetta, restituisci x_{yes}, altrimenti x_{no}

Analisi: $x \in L \Leftrightarrow R(x) \in A$

12.15 coNP

Idea: NP certifica che una data stringa appartiene ad un linguaggio, coNP il contrario, ovvero:

- NP certifica $x \in L$
- coNP certifica $x \notin L$

Domanda: se UNSAT=$\overline{\text{SAT}}$, allora UNSAT$\in$ NP? $\overline{\text{3COL}}$ $\in$NP?
Sappiamo che 3COL e SAT sono in NP, quindi esiste una NTM M che in tempo polinomiale li decide:

1. Uso M per dimostrare che UNSAT, $\overline{\text{3COL}} \in$ NP
2. Se M accetta **rifiuto** e se rifiuta **accetto**

12.15.1 Definizione - coNP

$$\text{coNP} = \{L : \overline{L} \in \text{NP}\}$$

Attenzione: coNP $\neq \overline{\text{NP}}$. In coNP sono presenti quei linguaggi la quale negazione è in NP. Ad esempio UNSAT $\in$ coNP.

12.15.2 Teorema - SAT $\in$ P $\Leftrightarrow$ UNSAT $\in$ P

SAT $\in$ P $\Leftrightarrow$ UNSAT $\in$ P.

Dimostrazione: se SAT $\in$ P, allora esiste una TM che decide SAT. La TM che decide UNSAT è uguale a quella che decide SAT ma semplicemente scambia l'**accept** con la **reject**, cosa che non si può fare usando il non determinismo.
Diverso però da UNSAT $\leq$ SAT.
Esempio: la riduzione che nega ϕ non va bene:

$$\phi = x \vee y \in SAT$$
$$\neg\phi = \neg(x \vee y) \in SAT$$

la stessa ϕ negata è comunque in SAT.
Esempio: $A \leq_m^p B \Leftrightarrow \overline{A} \leq_m^p \overline{B}$

12.15.3 Teorema - coP=P

coP = P, ovvero $L \in$ P $\Leftrightarrow \overline{L} \in$ P.
Anche EXP=coEXP. Queste due affermazioni sono vere in quanto sia D il decisore per un linguaggio $L \in$ P (o anche $L \in$ EXP). Per decidere $\overline{L}$ basta solamente invertire la ***accept*** con la ***reject*** in D.

12.15.3.1 Corollario - coNP $\subseteq$ EXP

coNP $\subseteq$ EXP.

Dimostrazione: sia $L \in$ coNP, allora $\overline{L} \in$ NP $\subseteq$ EXP. Quindi $\overline{L} \in$ EXP, allora $L \in$ coEXP=EXP.

12.15.4 Teorema - P $\subseteq$ coNP

P $\subseteq$ coNP.

Dimostrazione: se $L \in$ P, allora $\overline{L} \in$ coP=P$\subseteq$NP. Quindi $\overline{L} \in$ NP e $L \in$ coNP.

12.15.5 Teorema - P=NP $\Rightarrow$ P=coNP

P=NP $\Rightarrow$ P=coNP.

Dimostrazione: se $L \in$ coNP, allora $\overline{L} \in$ NP=P che implica $L \in$ P.

12.15.5.1 Corollario - P=NP $\Rightarrow$ NP=coNP

P=NP $\Rightarrow$ NP=coNP.

12.15.5.2 Corollario - coNP$\neq$NP $\Rightarrow$ P$\neq$NP

coNP$\neq$NP $\Rightarrow$ P$\neq$NP.

coNP $\neq$ NP è un assunzione più forte di P $\neq$ NP.

12.15.6 Rappresentazione insiemistica delle classi di linguaggi

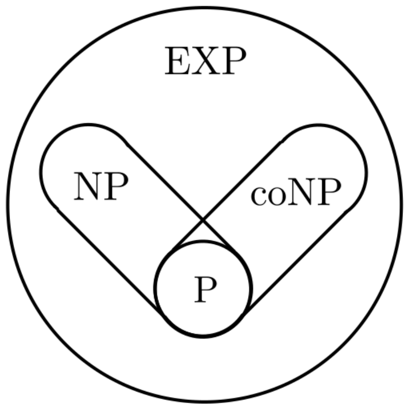


Figure 67: Rappresentazione insiemistica delle classi di linguaggi

12.15.7 Teorema - NP=coNP $\Leftrightarrow$ UNSAT $\in$ NP

NP=coNP $\Leftrightarrow$ UNSAT $\in$ NP.

<u>Dimostrazione</u>: due versi da dimostrare:

- $\Rightarrow$ Sia NP=coNP, allora UNSAT $\in$ coNP=NP e quindi UNSAT $\in$ NP
- $\Leftarrow$ Se UNSAT$\in$NP, dimostro che coNP$\subseteq$NP e NP$\subseteq$coNP.
  Sia $L \in$ coNP, quindi per definizione $\overline{L} \in$ NP e quindi per il teorema di Cook-Levin (vedere 12.9 pag. 105) $\Rightarrow \overline{L} \leq_m^p$ SAT allora $L \leq_m^p$ UNSAT e UNSAT $\in$ NP e dunque $L \in$ NP.

12.15.8 Definizione - coNP-completezza

Un linguaggio L è **coNP-completo** se:

1. $L \in$ coNP
2. L è **coNP-Hard**, ovvero per ogni linguaggio $A \in$ coNP si ha $A \leq_m^p$ L

12.15.9 Teorema - UNSAT è coNP-completo

UNSAT è coNP-completo.

<u>Dimostrazione</u>: ovviamente UNSAT $\in$ coNP. Sia un linguaggio $A \in$ coNP, devo mostrare che $A \leq_m^p$ UNSAT. Però $A \leq_m^p$ UNSAT $\Leftrightarrow \overline{A} \leq_m^p$ SAT. Ora, $\overline{A} \in$ NP e quindi segue il teorema.
Questo vale in generale. Se un linguaggio L è NP-completo, allora $\overline{L}$ è coNP-completo.

13 Complessità di spazio

Lo studio della complessità dello spazio si occupa di studiare la quantità di spazio, ovvero memoria, che un problema richiede. La differenza fondamentale con il tempo è che lo **spazio può essere riutilizzato**.

13.1 Definizione - Complessità di spazio

La **complessità di spazio** di una TM decisore è una funzione $S : \mathbb{N} \to \mathbb{N}$ tale che:

$$S(n) = \max_{x:|x|=n} \{\# \text{ celle di nastro distinte a cui } M(x) \text{ ha accesso}\}$$

13.1.1 Modifica del modello di TM

Per non pagare spazio occupato dall'input, modifichiamo il modello di TM che utilizzeremo nel seguente modo.
La TM avrà due nastri e due testine:

- Il **primo nastro** viene detto **input-tape** ed è di sola lettura
- Il **secondo nastro** viene detto **work-tape**

Ricordiamo che una TM a k-nastri con tempo $T(n)$ può essere simulata con un nastro a tempo $O(T^2(n))$.
Esempio: una TM a $k = O(1)$-nastri con spazio $S(n)$ può essere simulata con un nastro a spazio $O(S(n))$.

13.2 Definizione - SPACE

Definiamo ora le seguente classe di spazio:

$$\text{SPACE}(f(n)) = \{A : \exists \text{TM } M \text{ che decide il linguaggio } A \text{ in spazio } O(f(n))\}$$

13.3 Definizione - L

$$\text{L} = \text{SPACE}(\log(n))$$

Ovvero in L sono presenti tutti i linguaggi decisi da una TM con spazio $\log(n)$.

13.3.1 Esempi

Alcuni esempi di analisi di problemi in L:

- Si consideri il linguaggio $A = \{0^n 1^n : n \geq 0\} \in \text{L}$. Sia M la seguente TM che lo decide:

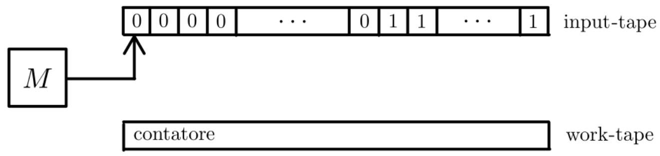


Figure 68: TM M per il linguaggio A

<u>Analisi</u>:

- Finché viene letto il carattere "0", viene incrementato il contatore. Spazio $\underline{O(\log(n)}$
- Appena viene letto il primo carattere "1", viene decrementato il contatore per ogni "1" che viene letto
- Se alla fine il contatore è pari a zero, accetto

Spazio totale: $\underline{O(\log(n)}$

- PALINDROMES $\in$ L. Ecco la TM con 6 nastri che lo decide:
  - Su input x, determino $n = |x|$
  - For $i = 1, \ldots, n$ rifiuto se $x_i \neq x_{n+1-i}$
  - Accetto

  <u>Descrizione più dettagliata della TM</u>:

  - Il primo passo richiede spazio $\underline{O(\log(n))}$ sul *work-tape* 1
  - Su *work-tape* 2 viene messo il contatore i che viene incrementato. Spazio $\underline{O(\log(n))}$
  - Su *work-tape* 3 viene calcolato $n + 1 - i$. Spazio $\underline{O(\log(n))}$
  - Per leggere x_i, viene copiato i sul *work-tape* 4 e si decrementa muovendo a destra la testina dell'input. Si smette di muovere a destra quando $i = 0$
  - Su *work-tape* 5 viene calcolato $n + 1 - i$, viene prelevato x_{n+1-i} e viene fatto il confronto

La classe di spazio L può essere visto nel seguente modo:

- Si conosce $n = |x|$
- Possono essere usati $O(1)$ contatori "i", "j" con range poly(n)
- Vengono fatte operazioni di *input lookup* e operazioni *aritmetiche*

Ovvero, più semplicemente, tutto quello che si riesce a fare utilizzando solo **contatori e puntatori**.

13.4 Definizione - PSPACE

Analogo della classe P ma stavolta per lo spazio:

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k)$$

13.5 Teorema - DTIME$(f(n)) \subseteq$ SPACE$(f(n))$

DTIME$(f(n)) \subseteq$ SPACE$(f(n))$.

<u>Dimostrazione</u>: una TM con tempo di esecuzione $O(f(n))$ può utilizzare al più $O(f(n))$ celle di nastro. Da questo possiamo ricavare anche che P $\subseteq$ PSPACE e che NP $\subseteq$ NPSPACE che è la classe di linguaggi decidibili da NTM con spazio $O(n^k)$.

13.6 Teorema - Per ogni $f(n) \geq \log n$, SPACE$(f(n)) \subseteq$ TIME$(2^{O(f(n))})$

Per ogni $f(n) \geq \log n$, SPACE$(f(n)) \subseteq$ TIME$(2^{O(f(n))})$.

<u>Idea</u>: una macchina che usa $f(n)$ celle di nastro non può usare più di $2^{O(f(n))}$ tempo in quanto $2^{O(f(n))}$ è il **numero di configurazioni** della macchina e queste non si possono ripetere perché le TM che consideriamo sono tutte decisori. Ad esempio, se la macchina usa 3 celle, il massimo numero di configurazioni senza ripetizioni è 2^3, che equivale proprio al tempo massimo di esecuzione della macchina.
<u>Dimostrazione</u>: sia M una TM che decide un linguaggio A in $O(f(n))$ spazio. Mostriamo che A si può decidere in tempo $2^{O(f(n))}$. Per semplicità, M ha un solo *work-tape*:

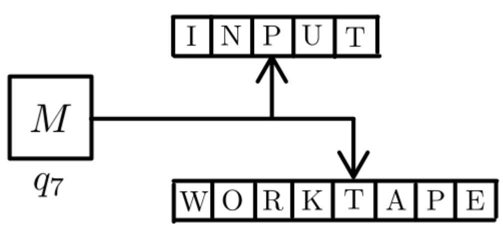


Figure 69: TM M

Configurazione di M:

$$\underbrace{\text{WORK}}_{(1)}\underbrace{q_7}_{(2)}\underbrace{\text{TAPE}}_{(3)}\underbrace{3}_{(4)}$$

Dove:

- $(1), (2)$ e (3) indicano il contenuto del *work-tape*, lo stato attuale e la posizione della testina, ovvero il contenuto del *work-tape* è WORKTAPE, lo stato attuale è q_7 e la testina si trova sulla T
- (4) indica il numero della cella indicata dalla testina del nastro di input, ovvero indica la P

Siccome M è un decisore, le configurazioni **non si ripetono**. Il tempo di M è $\leq$ al numero di configurazioni che a loro volta sono $\leq |\Gamma|^{f(n)} \cdot |Q| \cdot f(n) \cdot n$. Siccome $n \leq 2^{f(n)}$ e $|\Gamma|, |Q|$ sono costanti, segue dal teorema.

13.6.1 Corollario - P $\subseteq$ PSPACE

Dal teorema precedente segue che P $\subseteq$ PSPACE.

13.6.2 Rappresentazione insiemistica delle classi di linguaggio

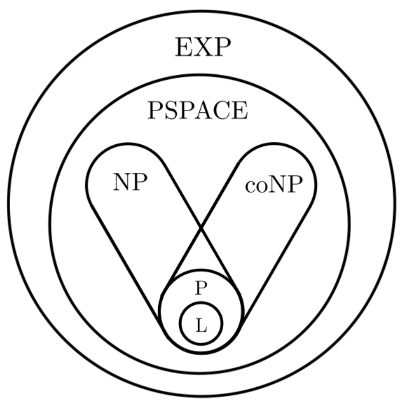


Figure 70: Rappresentazione insiemistica delle classi di linguaggo

L $\subseteq$ P $\subseteq$ PSPACE $\subseteq$ EXP e P $\neq$ EXP.
Quindi P $\neq$ PSPACE oppure PSPACE $\neq$ EXP (o entrambe). Vale il teorema di gerarchia.

13.6.3 Teorema di gerarchia di spazio

Sia $f(n)$ una funzione *spazio-costruttibile* ($f(n) \geq \log n$ è calcolabile in spazio $O(f(n))$) allora esiste un linguaggio L tale che $L \in \text{SPACE}(f(n))$ ma allo stesso tempo $L \notin \text{SPACE}(g(n))$ per $g(n) = o(f(n))$.

13.6.4 Teorema di gerarchia di tempo 2

Sia $f(n)$ una funzione *tempo-costruttibile*, allora esiste un linguaggio L tale che $L \in \text{DTIME}(f(n))$ ma allo stesso tempo $L \notin \text{DTIME}(g(n))$ per $g(n) = o(\frac{f(n)}{\log n})$.

13.7 Teorema - PATH $\in$ SPACE($\log^2 n$)

PATH $\in$ SPACE($\log^2 n$).

Dimostrazione: dato G grafo con n nodi, possiamo calcolare n in $O(\log n)$ spazio. La TM conosce anche i nodi $s, t \in V$. Consideriamo la funzione Path?(x, y, k) che restituisce *sì/no* se esiste un cammino da x a y con lunghezza $\leq 2^k$. La TM che consideriamo eseguirà Path?$(s, t, \lceil \log n \rceil)$:
Path?(x, y, k):

- Se $k = 0$, **accetta** se e solo se $(x, y) \in E$ oppure $x = y$
- Altrimenti:
    - Per ogni $w \in V$
      Se Path?$(x, w, k-1) \wedge$ Path?$(w, y, k-1)$ **accetta**
    - Altrimenti **rifiuta**

Analisi spazio: dobbiamo memorizzare i valori di n, s, t, x, y e k, con il riutilizzo dello spazio, quindi lo spazio occupato da questi valori è $O(\log n)$. Inoltre la profondità della ricorsione è $O(\log n)$. Quindi la complessità di spazio totale è $O(\log n) \cdot O(\log n) = O(\log^2 n)$.

13.8 Definizione - NSPACE

$$\text{NSPACE} = \{A : \exists \text{ NTM } N \text{ che decide } A \text{ con spazio } O(f(n))\}$$

13.9 Definizione - NL

$$\text{NL} = \text{NSPACE}(\log n)$$

13.10 Teorema - Teorema di Savitch

NL $\subseteq$ P, SPACE($\log^2 n$).
(Sul libro: NSPACE$(f(n)) \subseteq$ SPACE$(f^2(n))$ per $f(n) > n$)

Idea: dobbiamo far vedere che PATH $\in$ NL indovinando **non deterministicamente** un cammino $s \rightsquigarrow t$ di lunghezza n. Per usare $\log n$ spazio, il cammino deve essere indovinato **gradualmente**.
Su input G, s, t dove $G = (V, E)$ è un grafo e $s, t \in V$ sono nodi del grafo

- Se $s = t$ **accetto**
- Calcolo deterministicamente $n = |V|$
- Imposto la variabile *curNode*$= s$ e *for* $i = 1, \ldots, n$:
    - Scelgo non deterministicamente un nodo $u \in V$

- ◦ Se $(curNode, u) \in E$, imposto $curNode = u$
  - ◦ Se $curNode = t$, **accetto**
  - ◦ Altrimenti se $(curNode, u) \notin E$, **rifiuto**

- **Rifiuto**

Ciascun ramo usa $O(1)$ variabili e $O(\log n)$ spazio.
Correttezza: se esiste un **ramo accettante** allora esiste un cammino $s \rightsquigarrow t$. Viceversa, se esiste un cammino $s \rightsquigarrow t$ allora esiste un ramo accettante.

Per provare il teorema abbiamo bisogno del concetto di **configurazione**. Un esempio di configurazione è: $C = \text{WORK}q_7\text{TAPE}; 2$.
Il numero di configurazioni totali è $\leq 2^{O(\log n)} = \text{poly}(n)$ per una TM con $\log n$ spazio.
Nel caso non deterministico avremo *Next-config*$_0$ e *Next-config*$_1$ in corrispondenza di ciascuna diramazione non deterministica.

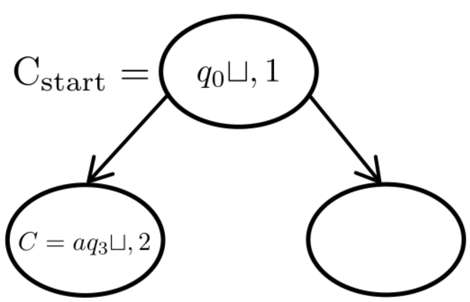


Figure 71: Esempio di configurazioni non deterministiche

Ogni configurazione non deterministica corrisponde ad un grafo $G_{N,x} = (V, E)$, chiamato **grafo delle configurazioni di** N **su input** x, con:

- $|V| = \text{poly}(n)$, corrispondono a tutte le configurazioni di N su x
- $(C, C') \in E$ se e solo se $C' = \textit{Next-config}_0(C)$ oppure $C' = \textit{Next-config}_1(C)$, ovvero se C' segue da C dopo un singolo step.

Affermazione: la NTM N con input x accetta se e solo se esiste un cammino $C_{\text{start}} \rightsquigarrow C_{\text{acc}}$ nel grafo delle configurazioni $G_{N,x}$.

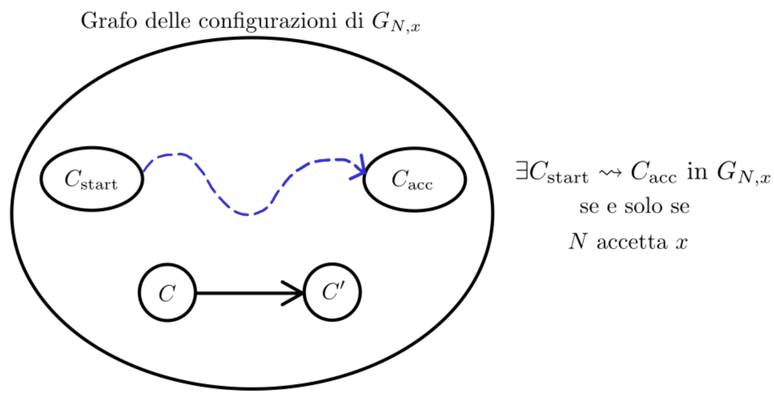


Figure 72: Grafo delle configurazioni

Definiamo la configurazione C_{acc} in **maniera univoca**, dicendo che N prima di accettare va nella configurazione $C_{\text{acc}} = q_{\text{acc}} \sqcup ; 1$, ovvero pulisce il contenuto del nastro e muove la testina completamente a sinistra.

Da una parte abbiamo NL $\subseteq$ P. Sia un linguaggio $A \in$ NL, ovvero esiste una NTM N che decide A in $\log n$ spazio. Allora esiste una TM M per decidere $x \in A$:

- Scrive $G_{N,x}$ (ovvero la lista dei nodi e degli archi, ha dimensione polinomiale)
- Usa una *BFS* per stabilire se esiste un cammino $C_{\text{start}} \rightsquigarrow C_{\text{acc}}$. Se tale cammino esiste **accetta**, altrimenti **rifiuta**

D'altra parte abbiamo NL $\subseteq$ SPACE($\log^2 n$), questo perché stabilire il cammino $C_{\text{start}} \rightsquigarrow C_{\text{acc}}$ è possibile deterministicamente in $O(\log^2 n)$ spazio.

Savitch si può generalizzare NSPACE($f(n)$) per $f(n) \geq \log n$.
Ora $|V| = 2^{O(f(n))} \Rightarrow$ NSPACE($f(n)$) $\subseteq$ DTIME($2^{O(f(n))}$), SPACE($f^2(n)$).
Se $f(n) = n^k,\ k > 0$

$$\bigcup_k \text{NSPACE}(n^k) \subseteq \bigcup_k \text{TIME}(2^{O(n^k)}), \bigcup_k \text{SPACE}(n^{2k})$$

NPSPACE $\subseteq$ EXP, PSPACE $\Rightarrow$ NPSPACE=PSPACE.
Quello che ci dice savitch è che il costo di spazio per passare da una NTM a una TM è quadratico.

13.11 Definizione - NL-completezza

Un linguaggio B è **NL-completo** se:

1. $B \in$ NL

2. B è **NL-Hard**, ovvero per ogni altro linguaggio $A \in$ NL, si ha che $A \leq B$

Quale riduzione usare però? Se usassimo $\leq_m^p$ andremmo a distruggere l'efficienza di spazio.
Quello che vogliamo è: $A \in$ NL e $A \leq B$, allora

$$B \in \text{L} \Rightarrow A \in \text{L}$$
$$B \in \text{NL} \Rightarrow A \in \text{NL}$$

Inoltre vale la **transitività**: $A \leq B, B \leq C \Rightarrow A \leq C$.

13.11.1 Definizione - $\leq_m^L$ log-space mapping reduction

Siano A e B due linguaggi. Diciamo che $A \leq_m^L B$ se esiste una funzione $R : \{0,1\}^* \to \{0,1\}^*$ calcolabile in $O(\log n)$ spazio tale che

$$\forall x, x \in A \Leftrightarrow R(x) \in B$$

Problema: la dimensione di $R(x)$ potrebbe essere n, ma deve essere calcolabile in $\log n$ spazio.
Soluzione: R può scrivere l'output su un tape apposito di tipo **write-once**.

13.12 Teorema - PATH è NL-completo

PATH è NL-completo.

Dimostrazione: abbiamo già visto che PATH $\in$ NL, dobbiamo quindi far vedere che per ogni linguaggio $A \in$ NL, $A \leq_m^L$ PATH.
Esiste una $O(\log n)$-spazio riduzione $R : \{0,1\}^* \to \{0,1\}^*$ che dato input $x \in \{0,1\}^*$ da in output un grafo $G_{N,x}$ e le configurazioni C_{start} e C_{acc}. Così:

$$x \in A \Leftrightarrow N(x) \text{ accetta} \Leftrightarrow \exists C_{\text{start}} \rightsquigarrow C_{\text{acc}} \text{ in } G_{N,x} \Leftrightarrow R(x) \in \text{PATH}$$

Questo è chiaro dalla dimostrazione precedente.

PATH è NL-completo, questo significa che PATH è il **problema più difficile in NL**, infatti PATH $\in$ L se e solo se tutti gli altri problemi in NL sono in L.

13.13 Teorema - Siano P, Q funzioni. Se queste sono calcolabili in log-space allora lo è anche $R(x) = Q(P(x))$

Siano P, Q funzioni. Se queste sono calcolabili in log-space allora lo è anche $R(x) = Q(P(x))$.

13.13.1 Corollario - Se $A \leq_m^L B$ allora $B \in \mathbf{L} \Rightarrow A \in \mathbf{L}$

Se $A \leq_m^L B$ allora $B \in \text{L} \Rightarrow A \in \text{L}$.

Dimostrazione: P è la riduzione da A a B e sia $Q(x)$ tale che restituisce 1 se e solo se la TM che decide il linguaggio B accetta. Quindi $Q(P(x))$ può essere utilizzata per decidere $A \in \text{L}$ in log-space.

13.13.2 Corollario - $A \leq_m^L B$, allora $B \in \mathbf{NL} \Rightarrow A \in \mathbf{NL}$

$A \leq_m^L B$, allora $B \in \text{NL} \Rightarrow A \in \text{NL}$.

13.13.3 Corollario - $A \leq_m^L B, B \leq_m^L C \Rightarrow A \leq_m^L C$

$A \leq_m^L B, B \leq_m^L C \Rightarrow A \leq_m^L C$.

13.14 Dimostrazione teorema 13.13

Diciamo che P ha tempo n^p, mentre Q ha tempo n^q. Dobbiamo trovare una TM M che prende come input x con $n = |x|$ e calcola $R(x)$.
Sia $y = P(x)$ con $|y| \leq n^p$ e M deve calcolare $Q(y)$, quindi M deve simulare Q come se avesse un input tape **read-only** per y. Come fa? Tiene traccia della posizione della testina sull'input tape di Q ed è per questo che usa $\log n$ spazio. Poi preleva l'i-esimo carattere di y e simula Q. Per fare ciò, calcola ripetutamente $P(x) = y$ fino ad ottenere $y[i]$, buttando via tutti gli altri caratteri di y che non servono.

13.15 Definizione - P-completezza

Un problema C è **P-completo** se:

1. $C \in \text{P}$
2. C è **P-Hard**, ovvero per ogni linguaggio $A \in \text{P}$, si ha che $A \leq_m^p \text{C}$

Questo è utile perché $C \in \text{L}$ se e solo se $\text{P} \subseteq \text{L}$. Ciò però **non è noto**, dunque se vogliamo dimostrare che non è vero e quindi trovare un problema in P che non può essere risolto in un numero costante di variabili, allora devo usare C.
Esempi:

$$\text{CIRCUIT-EVAL} = \{\langle C, x\rangle : C \text{ è un circuito e } C(x) = 1\}$$

Fatto: CIRCUIT-EVAL $\leq_m^L$ 3-SAT. Inoltre il teorema di Cook-Levin (teorema 12.9 pag. 105) vale anche sotto log-space reductions.

13.15.1 Corollario - CIRCUIT-EVAL è P-completo

CIRCUIT-EVAL è **P-completo**.

Dimostrazione: CICRUIT-EVAL $\in$ P. Dobbiamo far vedere che è P-hard, ovvero:

$$\forall A \in \text{P}, \quad A \leq_m^L \text{CIRCUIT-EVAL}$$

Il circuito per A è quello di Cook-Levin.

13.16 Il problema TQBF

Si consideri il seguente linguaggio chiamato *Totally Quantified Boolean Formula*:

TQBF $= \{$Affermazioni vere del tipo: $Q_1x_1, Q_2x_2, \ldots, Q_nx_n,\ \phi(x_1, \ldots, x_n) = 1\}$

dove $Q_1, Q_2, \ldots, Q_n \in \{\exists, \forall\}$.
Esempio di una TQBF:

$$\phi = \forall x \exists y[(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

Come si può vedere, **tutte le variabili** che appaiono nella formula sono **quantificate**.

13.16.1 Proposizione - TQBF $\in$ PSPACE

TQBF $\in$ PSPACE.

Dimostrazione: consideriamo la seguente funzione **ricorsiva**:
isTrue?$(Q_1x_1, \ldots, Q_nx_n, \phi(x_1, \ldots, x_n))$:

- Se $n = 0$, l'input è una **formula costante** e quindi restituisco la sua valutazione
- Altrimenti, se $Q_1 = \exists$ restituisco:

$$\text{isTrue?}(Q_2x_2, \ldots, Q_nx_n, \phi(0, x_2, \ldots, x_n)) \vee$$
$$\text{isTrue?}(Q_2x_2, \ldots, Q_nx_n, \phi(1, x_2, \ldots, x_n))$$

  mentre se $Q_1 = \forall$ restituisco:

$$\text{isTrue?}(Q_2x_2, \ldots, Q_nx_n, \phi(0, x_2, \ldots, x_n)) \wedge$$
$$\text{isTrue?}(Q_2x_2, \ldots, Q_nx_n, \phi(1, x_2, \ldots, x_n))$$

Complessità di spazio: la ricorsione è profonda n e ad ogni ricorsione possiamo riusare lo spazio utilizzato tra chiamate diverse. Per ogni chiamata lo spazio è **lineare**, quindi lo spazio utilizzato è: $O(n^2) \in \text{poly}(n)$.

13.17 Teorema - TQBF è PSPACE-completo

TQBF è **PSPACE-completo**.

Dimostrazione: siccome TQBF $\in$ PSPACE (proposizione 13.16.1) dobbiamo mostrare che per ogni linguaggio $A \in$ PSPACE, $A \leq_m^p$ TQBF. Faremo la seguente semplificazione:

- Insistere che $\exists, \forall$ si alternano
- ϕ ammette anche "$\Rightarrow$" e "$\Leftrightarrow$" o una CNF
- I quantificatori non sono all'inizio:

$$\exists x_1 \forall x_2 (x_1 \Rightarrow x_2) \wedge \exists x_3 \ldots$$

Queste semplificazioni non alterano in nessun modo una TQBF.
Sia $A \in$ PSPACE, questo vuol dire che esiste una TM M che decide A in spazio $O(n^a)$ per qualche a. Dobbiamo costruire una riduzione R tale che:

$$x \in A \Leftrightarrow R(x) = \phi \text{ è vera}$$

Abbiamo che: $x \in A \Leftrightarrow \exists$ un cammino $C_{\text{start}} \rightsquigarrow C_{\text{acc}}$ nel grafo i cui nodi sono le configurazioni di M su x, ovvero $G_{M,x}$ e gli archi corrispondono a transizioni. Il numero di nodi è $\leq 2^{O(n^a)}$.
R produce ϕ tale che $\phi(x) = 1$ se e solo se $\exists C_{\text{start}} \rightsquigarrow C_{\text{acc}}$. La formula ϕ deve essere tale che $|\langle \phi \rangle| = \text{poly}(n)$ e deve essere $\text{poly}(n)$ computabile:

$$\phi_{M,n} = \forall C_1 \exists x_2 \ldots \phi(C_1, \ldots) = 1$$

le variabili di ϕ sono le configurazioni della macchina, che utilizzano $O(n^a)$ bit.
Vediamo alcune idee:

- **Idea 0**:

  $$\phi_{M,n} = \exists C_1 \exists C_2 \ldots \exists C_l$$
  $$\phi_{\text{yelds}}(C_{\text{start}}, C_1) \wedge \phi_{\text{yelds}}(C_1, C_2) \wedge \cdots \wedge \phi_{\text{yelds}}(C_l, C_{\text{acc}})$$

  Dove $\phi_{\text{yelds}}(C_i, C_j) = 1$ se la configurazione C_j segue dalla configurazione C_i.

  Però $l \leq 2^{O(n^a)}$ che è **esponenziale**.

- **Idea 1**: utilizziamo il teorema di Savitch. Costruiamo $\phi_k(C_0, C_1)$ che è vera se e solo se esiste un cammino $C_0 \rightsquigarrow C_1$ in $\leq 2^k$ passi. La riduzione R restituirà $\phi_{O(n^a)}(C_{\text{start}}, C_{\text{acc}})$:

  - **Caso base**:

    $$\phi_0(C_0, C_1) = \phi_{\text{yields}}(C_0, C_1) \vee (C_0 = C_1)$$

- **In generale**:

$$\phi_k(C_0, C_1) = \exists C_{\text{mid}} : \phi_{k-1}(C_0, C_{\text{mid}}) \wedge \phi_{k-1}(C_{\text{mid}}, C_1)$$

Ricorsione k: $\text{size}(\phi_n) = O(n^a) + 2 \cdot \text{size}(\phi_{k-1}) \Rightarrow \text{size}(\phi_k) = O(2^k \cdot n^a)$, dove $k = O(n^a)$. Quindi è ancora **esponenziale**.

- **Idea finale**: $\phi_k(C_0, C_1) = \exists C_{\text{mid}} \forall D \forall D'$ tale che:

$$\begin{pmatrix} (D, D') = (C_0, C_{\text{mid}}) \\ \vee \\ (D, D') = (C_{\text{mid}}, C_1) \end{pmatrix} \Rightarrow \phi_{k-1}(D, D')$$

Quindi: $\text{size}(\phi_k) = O(n^a) + \text{size}(\phi_{k-1}) = O(k \cdot n^a) = O(n^2 a)$, che è **polinomiale**.

13.18 Teorema - Immurmen-Szelepcsényi

NL=coNL.

La classe NL è **chiusa rispetto al complemento**. Un'affermazione equivalente è dire PATH $\in$ coNL, ovvero $\overline{\text{PATH}} \in$ NL. Dobbiamo quindi trovare un certificato in log-space che certifichi che non esiste il cammino $s \rightsquigarrow t$ nel grafo G, ovvero esiste un **verificatore** in log-space per $\langle G, s, t \rangle \notin$ PATH. Il verificatore avrà 3 nastri con i seguenti dati:

- **Nastro 1 (input-tape)**: l'input $\langle G, s, t \rangle$
- **Nastro 2 (read-once)**: c'è il certificato di non appartenenza
- **Nastro 3 (work-tape)**: è il work-tape di spazio log-space

Idea: sia R_l l'**insieme di nodi** raggiungibili da s in l passi, e sia $r_l = |R_l|$. Ad esempio se $R_0 = \{s\}$, avremo $r_0 = 1$. Inoltre ogni $R_l \subseteq R_{l+1}$.
Certificato: il certificato sarà della seguente forma, formato da tanti pezzi:

$$\text{cert. per } r_0, \text{cert. per } r_1, \ldots, \text{cert. per } r_n, \text{cert. per } s \not\rightsquigarrow t$$

Ogni "pezzo" indica il numero di nodi raggiungibili con una distanza da s pari o minore al numero che si trova come pedice di r.
Osservazione chiave: quando controlliamo r_{l+1}, abbiamo bisogno solamente di l e r_l.
Funzionamento del certificato: iniziamo dalla fine. Supponiamo che il verificatore V conosce r_n, il certificato sarà $t \notin R_n$:

$$\xrightarrow{\text{ord. cresc.}}$$
$$s \rightsquigarrow v_2, s \rightsquigarrow v_5, s \rightsquigarrow v_6, \ldots, s \rightsquigarrow v_{n-1}$$

Controllo: i path (cammini distinti) sono in G, ci sono r_n path e t non è mai alla fine.

Infine: certificato di r_{l+1} assumendo di avere il certificato di r_l: $v_1 \in R_{l+1}$ perché $\ldots$, $v_2 \in R_{l+1}$ perché, $\ldots v_n \in R_{l+1}$ perché $\ldots \Rightarrow$ Se contiamo abbiamo $|R_{l+1}| = r_{l+1}$.
Esempio:

- $v_8 \in R_{l+1}$. Path di lunghezza $l + 1$
- $v_9 \notin R_{l+1}$. Come prima, ovvero

$$s \overset{\leq l}{\rightsquigarrow} v_1, s \overset{\leq l}{\rightsquigarrow} v_2, \ldots$$

14 Conclusioni

Questi sono tutti gli argomenti trattati a lezione dal prof. Daniele Venturi nel corso tenuto nell'anno accademico 2022/23. Non è detto che le dispense siano prive di errori, se dovreste trovarne alcuni potete segnalarmeli contattandomi su Facebook.

Spero che queste dispense vi siano utili per il superamento dell'esame, in caso affermativo, questo è il mio account PayPal nel caso vogliate esprimermi la vostra gratitudine. Ve ne sarei molto riconoscente! Anche il semplice gesto di donarmi quanto basta per un buon caffè :)

Detto questo, *buona fortuna per l'esame!*