SAPIENZA
UNIVERSITÀ DI ROMA

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

Automi Calcolabilità e Complessità

Autore:
Simone Lidonnici

18 dicembre 2024

Indice

1

Linguaggi regolari

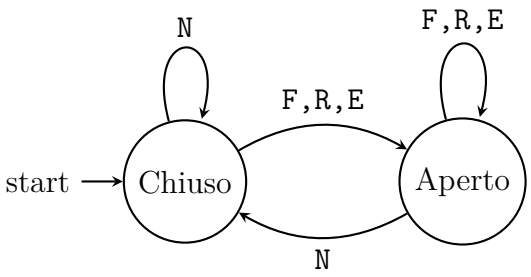
Definizione di linguaggio

Dato un **alfabeto** Σ, cioè un insieme di elementi, un **linguaggio** Σ^* è l'insieme di tutte le strighe ottenibili usando l'alfabeto Σ.

1.1 Automi Deterministici a Stati Finiti (DFA)

Il modello usato per definire i **linguaggi regolari** è l'**automa a stati finiti**, cioè una macchina che permette tramite l'input di passare da uno stato ad un altro, che ha memoria limitata e gestione dell'input limitata, ma è molto semplice.
Esempio:
Una porta che si apre tramite dei sensori può essere descritta tramite un automa con due stati (Aperta e Chiusa) e quattro input dati dai sensori (`N, F, R, E`):

- `N`: se non ci sono persone da nessun lato della porta
- `F`: se c'è una persona davanti alla porta
- `R`: se c'è una persona dietro la porta
- `E`: se ci sono persone da entrambi i lati della porta

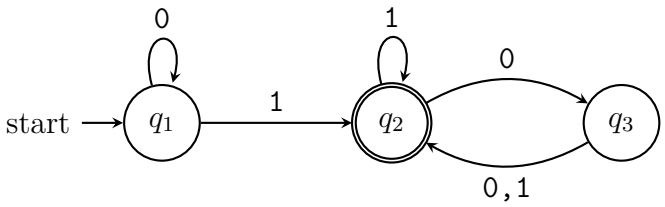


Automa Deterministico a Stati Finiti (DFA)

Un **DFA (Deterministic Finite Automaton)** è una tupla $(Q, \Sigma, \delta, q_o, F)$ in cui:

- Q è l'insieme degli stati dell'automa
- Σ è l'alfabeto dell'automa
- $\delta : Q \times \Sigma \to Q$ è la funzione di transizione degli stati
- $q_0 \in Q$ è lo stato iniziale dell'automa
- $F \subseteq Q$ è l'insieme di **stati accettanti** dell'automa

Esempio:
Preso il seguente DFA:



In questo caso:

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

δ	q_1	q_2	q_3
0	q_1	q_3	q_2
1	q_2	q_2	q_2

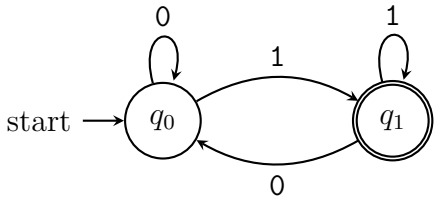
- q_1 è lo stato iniziale dell'automa
- $F = \{q_2\}$ è l'insieme degli stati accettanti

Funzione di transizione estesa

Dato un DFA D, definiamo una **funzione di transizione estesa** $\delta^* : Q \times \Sigma^* \to Q$ in modo ricorsivo:

$$\begin{cases} \delta^*(q, \varepsilon) = \delta(q, \varepsilon) = q \\ \delta^*(q, ax) = \delta^*(\delta(q, a), x) \quad a \in \Sigma, x \in \Sigma^* \end{cases}$$

Esempio:
Preso il seguente DFA:



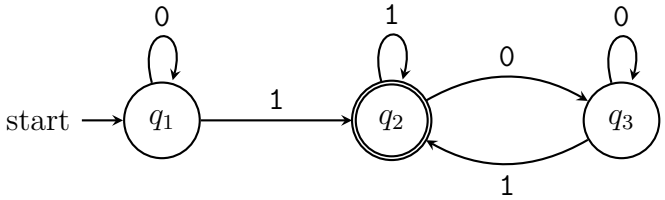
In questo DFA:

- $\delta^*(q_0, 011) = \delta^*(\delta(q_0, 0), 11) = \delta^*(q_0, 11) = \delta^*(\delta(q_0, 1), 1) = \delta^*(q_1, 1) = \delta^*(\delta(q_1, 1), \varepsilon) = \delta^*(q_1, \varepsilon) = q_1$

Linguaggio di un automa

Il **linguaggio di un DFA** D è l'insieme delle stringhe in input che l'automa accetta, cioè quelle per cui l'automa termina in uno stato accettante $q_0 \in F$.
Ogni automa ha un solo linguaggio che si scrive $L(D) = \{x \in \Sigma^* | D \text{ accetta } x\}$. Una stringa $x \in \Sigma^*$ sarà accettata da una DFA se $\delta^*(q_0, x) = q \in F$

Esempio:
Preso il seguente DFA:



Il linguaggio di questa DFA è l'insieme di tutte le stringhe che finiscono con 1:

$$L(D) = \{x \in \{0,1\}^* | x = y1 \wedge y \in \{0,1\}^*\}$$

1.1.1 Configurazione di un DFA

Configurazione di un DFA

Dato un DFA D, una **configurazione** di D è una coppia $Q \times \Sigma^*$ che indica:

- lo stato attuale dell'automa
- l'input ancora da leggere

La configurazione iniziale è sempre (q_0, x).

Passo di computazione di un DFA

Un **passo di computazione** è una relazione binaria con simbolo $\vdash_D$ per cui:

$$(q_1, ax) \vdash_D (q_2, x) \iff \delta(q_1, a) = q_2$$

La **chiusura per riflessione e transitività** di $\vdash_D$, scritta come $\vdash_D^*$, ha delle proprietà:

1. $(q_1, ax) \vdash_D (q_2, x) \implies (q_1, ax) \vdash_D^* (q_2, x)$
2. $\forall q, x \ (q, x) \vdash_D^* (q, x)$
3. $(q_1, abc) \vdash_D (q_2, bc) \vdash_D (q_3, c) \implies (q_1, abc) \vdash_D^* (q_3, c)$

Una stringa $x \in \Sigma^*$ sarà accettata da un DFA se:

$$\exists q \in F | (q_0, x) \vdash_D^* (q, \varepsilon)$$

1.2 Automi Non Deterministici a Stati Finiti (NFA)

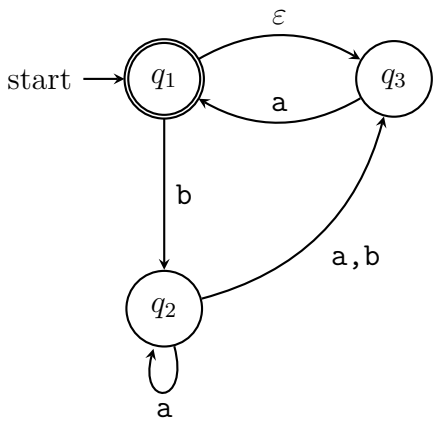
Automa Non Deterministico a Stati Finiti (NFA)

Un **NFA (Non-deterministic Finite Automaton)** è una tupla $(Q, \Sigma, \delta, q_o, F)$ in cui:

- Q è l'insieme degli stati dell'automa
- Σ è l'alfabeto dell'automa
- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ è la funzione di transizione degli stati
- $q_0 \in Q$ è lo stato iniziale dell'automa
- $F \subseteq Q$ è l'insieme di **stati accettanti** dell'automa

A differenza del DFA la funzione δ ha come uno dei parametri $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e come ritorno un insieme di stati, contenuto nell'insieme delle parti di Q, cioè $\mathcal{P}(Q)$. Inoltre per considerare una stringa accettata da un NFA basta che in uno dei rami della computazione la stringa venga accettata.

Esempio:
Preso il seguente NFA:



In questo caso:

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{\texttt{a},\texttt{b}\}$
- $\delta =$

δ	q_1	q_2	q_3
a	$\emptyset$	$\{q_2, q_3\}$	q_1
b	q_2	q_3	$\emptyset$
ε	q_3	$\emptyset$	$\emptyset$

- q_1 è lo stato iniziale dell'automa
- $F = \{q_1\}$ è l'insieme degli stati accettanti

1.2.1 Configurazione di un NFA

Configurazione di un NFA

Dato un NFA N, una **configurazione** di N è una coppia $Q \times \Sigma_\varepsilon^*$ che indica:

- lo stato attuale dell'automa
- l'input ancora da leggere

La configurazione iniziale è sempre (q_0, x).

Passo di computazione di un NFA

Un **passo di computazione** è una relazione binaria con simbolo $\vdash_N$ per cui:

$$(q_1, ax) \vdash_N (q_2, x) \iff q_2 \in \delta(q_1, a)$$

La **chiusura per simmetria e transitività** di $\vdash_N$, scritta come $\vdash_N^*$, ha delle proprietà:

1. $(q_1, ax) \vdash_N (q_2, x) \implies (q_1, ax) \vdash_N^* (q_2, x)$
2. $(q_1, abc) \vdash_N (q_2, bc) \vdash_N (q_3, c) \implies (q_1, abc) \vdash_N^* (q_3, c)$

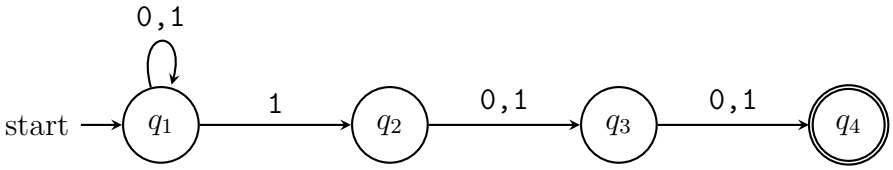
Una stringa $x \in \Sigma^*$ sarà accettata da un NFA se:

$$\exists q \in F | (q_0, x) \vdash_N^* (q, \varepsilon)$$

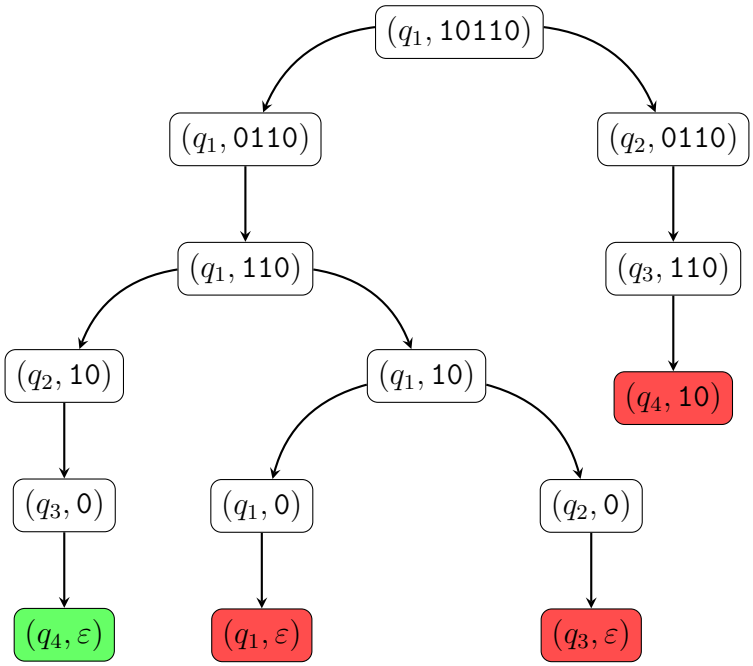
Oppure se:

$$x = y_1 \dots y_n \wedge \exists \underbrace{q_0 \dots q_n}_{\text{sequenza di stati}} | q_{i+1} = \delta(q_i, y_{i+1}) \wedge q_n \in F$$

Esempio:
Preso il seguente NFA:



Data la stringa `10110` la computazione sarà:



La stringa `10110` viene quindi accettata dal NFA.

1.3 Linguaggi regolari

Insieme dei Linguaggi regolari

Dato un alfabeto Σ, l'insieme dei **linguaggi regolari** di Σ, scritto come REG, è l'insieme dei linguaggi per cui esiste una DFA che li accetta:

$$\text{REG} = \{L \subseteq \Sigma^* | \exists D \; L(D) = L\}$$

1.3.1 Proprietà dei linguaggi regolari

I linguaggi sono insiemi di stringhe di un alfabeto Σ, quindi dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$ possiamo definire le operazioni:

- **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:**
$$\overline{L} = \{x \in \Sigma^* | x \notin L\}$$
- **Concatenazione:**
$$L_1 \circ L_2 = \{xy \in \Sigma^* | x \in L_1 \wedge y \in L_2\}$$
- **Potenza:**
$$L^n = \begin{cases} \{\varepsilon\} & n = 0 \\ L \circ L^{n-1} & n > 0 \end{cases}$$
- **Star di Kleene:**
$$L^* = \{x_1 \dots x_k \in \Sigma^* | x_i \in L\} = \bigcup_{n \geq 0} L^n$$

1.3.2 Chiusura dei linguaggi regolari

Chiusura dell'unione in REG

L'unione è chiusa in REG, cioè:

$$\forall L_1, L_2 \in \text{REG} \implies L_1 \cup L_2 \in \text{REG}$$

Dimostrazioni:

- **Tramite DFA:**
  Dati $L_1, L_2 \in$ REG, allora esistono:
  - $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) | L(D_1) = L_1$
  - $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) | L(D_2) = L_2$

  Creo il DFA $D_0 = (Q_0, \Sigma, \delta_0, q_0, F_0)$ per cui:
  - $Q_0 = Q_1 \times Q_2 = \{(q_x, q_y) | q_x \in Q_1 \wedge q_y \in Q_2\}$
  - $q_0 = (q_1, q_2)$
  - $F_0 = (F_1 \times Q_2) \cup (Q_1 \times F_2) = \{(q_x, q_y) | q_x \in F_1 \vee q_y \in F_2\}$
  - $\forall (q_x, q_y) \in Q_0, a \in \Sigma$:

    $$\delta((q_x, q_y), a) = (\delta_1(q_x, a), \delta_2(q_y, a))$$

  Quindi $x \in L(D_0) \iff x \in L_1 \vee x \in L_2$, quindi $L_1 \cup L_2 \in$ REG.

- **Tramite NFA:**
  Dati $L_1, L_2 \in$ REG, allora esistono:
  - $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) | L(N_1) = L_1$
  - $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) | L(N_2) = L_2$

  Creo il NFA $N_0 = (Q_0, \Sigma, \delta_0, q_0, F_0)$per cui:
  - $Q_0 = Q_1 \cup Q_2 \cup \{q_0\}$
  - q_0 è un nuovo stato iniziale
  - $F_0 = F_1 \cup F_2$
  - $\forall q \in Q_0, a \in \Sigma$:

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

  Quindi $x \in L(N_0) \iff x \in L_1 \vee x \in L_2$, quindi $L_1 \cup L_2 \in$ REG.

Chiusura dell'intersezione in REG

L'intersezione è chiusa in REG, cioè:

$$\forall L_1, L_2 \in \text{REG} \implies L_1 \cap L_2 \in \text{REG}$$

Dimostrazione:
Dati $L_1, L_2 \in$ REG, allora esistono:

- $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) | L(D_1) = L_1$
- $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) | L(D_2) = L_2$

Creo il DFA $D_0 = (Q_0, \Sigma, \delta_0, q_0, F_0)$ per cui:

- $Q_0 = Q_1 \times Q_2 = \{(q_x, q_y) | q_x \in Q_1 \wedge q_y \in Q_2\}$
- $q_0 = (q_1, q_2)$
- $F_0 = F_1 \times F_2$
- $\forall (q_x, q_y) \in Q_0, a \in \Sigma$:
$$\delta((q_x, q_y), a) = (\delta_1(q_x, a), \delta_2(q_y, a))$$

Quindi $x \in L(D_0) \iff x \in L_1 \wedge x \in L_2$, quindi $L_1 \cap L_2 \in$ REG.

Chiusura del complemento in REG

Il complemento è chiuso in REG, cioè:

$$\forall L \in \text{REG} \implies \overline{L} \in \text{REG}$$

Dimostrazione:
Dato $L \in$ REG, esiste $D = (Q, \Sigma, \delta, q_0, F) | L(D) = L$.
Creo il DFA $D^* = (Q, \Sigma, \delta, q_0, Q - F)$, uguale a D ma con gli stati accettanti invertiti, quindi $x \in L(D^*) \iff x \notin L(D)$, quindi $\overline{L} \in$ REG.

Chiusura della concatenazione in REG

La concatenazione è chiusa in REG, cioè:

$$\forall L_1, L_2 \in \text{REG} \implies L_1 \circ L_2 \in \text{REG}$$

Dimostrazione:
Dati $L_1, L_2 \in \text{REG}$, allora esistono:

- $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) | L(N_1) = L_1$
- $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) | L(N_2) = L_2$

Creo il NFA $N_0 = (Q_0, \Sigma, \delta_0, q_0, F_0)$ per cui:

- $Q_0 = Q_1 \cup Q_2$
- $q_0 = q_1$
- $F_0 = F_2$
- $\forall q \in Q_0, a \in \Sigma$:

$$\delta_0(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 - F_1 \\ \delta_1(q,a) & q \in F_1 \wedge a \neq \varepsilon \\ \delta_1(q,a) \cup q_2 & q \in F_1 \wedge a = \varepsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

Quindi $x \in L(D_0) \iff x \in L_1 \circ L_2$, quindi $L_1 \cap L_2 \in \text{REG}$.

Chiusura di star in REG

Star è chiusa in REG, cioè:

$$\forall L \in \text{REG} \implies L^* \in \text{REG}$$

Dato $L \in \text{REG}$, esiste $N = (Q, \Sigma, \delta, q_0, F) | L(D) = L$.
Creo il NFA $N^* = (Q^*, \Sigma, \delta^*, q_0^*, F^*)$ in cui:

- q_0^* è un nuovo stato iniziale
- $Q^* = Q \cup q_0^*$
- $F^* = F \cup q_0^*$
- $\forall q \in Q^*, a \in \Sigma$:

$$\delta^*(q,a) = \begin{cases} \delta(q,a) & q \in Q - F \\ \delta(q,a) & q \in F \wedge a \neq \varepsilon \\ \delta(q,a) \cup q_0 & q \in F \wedge a = \varepsilon \\ q_0 & q = q_0^* \wedge a = \varepsilon \\ \emptyset & q = q_0^* \wedge a \neq \varepsilon \end{cases}$$

Quindi $x \in L(N^*) \iff x \in L^*$, quindi $L^* \in \text{REG}$.

1.4 Equivalenza tra DFA e NFA

Linguaggi accettati da DFA e NFA

Sia $\mathcal{L}(\text{NFA})$ l'insieme dei linguaggi per cui esiste un NFA che li accetta e $\mathcal{L}(\text{DFA})$ l'insieme dei linguaggi per cui esiste un DFA che li accetta, allora:

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

Dimostrazione per doppia inclusione:

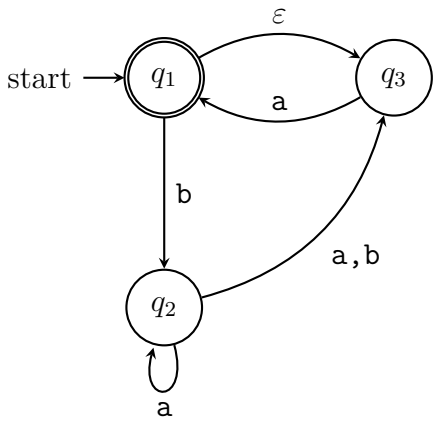
1. $\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{NFA})$:
   Dato un $L \in \mathcal{L}(\text{DFA})$ allora esiste $D = (Q, \Sigma, \delta, q_0, F)$ tale che $L(D) = L$.
   Visto che NFA è una generalizzazione di DFA, allora è ovvio che:
   $$\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{NFA})$$

2. $\mathcal{L}(\text{NFA}) \subseteq \mathcal{L}(\text{DFA})$:
   Dato un $L \in \mathcal{L}(\text{NFA})$ allora esiste $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$ tale che $L(N) = L$.
   Considero un DFA $D = (Q_D, \Sigma, \delta_D, q_{0_D}, F_D)$, costrito partendo da N in cui:

   - $Q_D = \mathcal{P}(Q_N)$
   - Dato $R \in Q_D$, definiamo l'**estensione** di R:
     $$E(R) = \left\{ q \in Q_N \,\middle|\, \begin{array}{c} q \text{ può essere raggiunto da uno stato } r \in R \\ \text{tramite solamente } \varepsilon\text{-archi} \end{array} \right\}$$
   - $q_{0_D} = E(\{q_{0_N}\})$
   - $F_D = \{R \in Q_D | R \cap F_N \neq \emptyset\}$
   - Dati $R \in Q_D$ e $a \in \Sigma$, δ_D è definita:
     $$\delta_D(R, a) = \bigcup_{r \in R} E(\delta_N(r, a))$$

   Detto questo allora abbiamo che:
   $$w \in L(N) \iff w \in L(D)$$
   quindi:
   $$\mathcal{L}(\text{NFA}) \subseteq \mathcal{L}(\text{DFA})$$

Esempio:
Preso il seguente NFA:



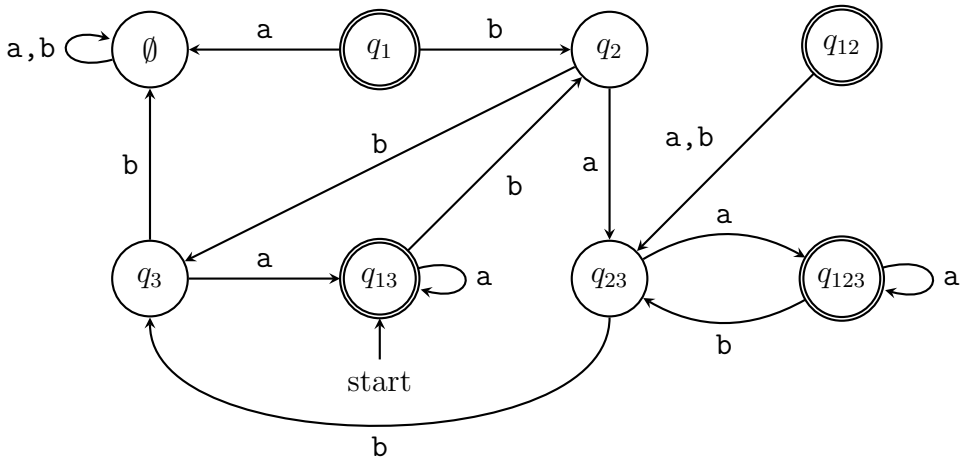
Per costruire il DFA equivalente, che sarà definito:

- $Q_D = \{\emptyset, q_1, q_2, q_3, \{q_1, q_2\}, \{q_1, q_3\}, \{q_2, q_3\}, \{q_1, q_2, q_3\}\}$
  che scriviamo in notazione semplificata:

$$Q_D = \{\emptyset, q_1, q_2, q_3, q_{12}, q_{13}, q_{23}, q_{123}\}$$

- $q_{0_D} = E(\{q_{0_N}\}) = E(q_1) = \{q_1, q_3\} = q_{13}$
- $F_D = \{q_1, q_{12}, q_{13}, q_{123}\}$
- δ_D viene definita come scritto prima, per esempio:
  - $\delta_D(q_1, a) = E(\delta_N(q_1, a)) = \emptyset$
  - $\delta_D(q_1, b) = E(\delta_N(q_1, b)) = q_2$
  - $\delta_D(q_2, a) = E(\delta_N(q_2, a)) = \{q_2, q_3\} = q_{23}$
  - $\delta_D(q_2, b) = E(\delta_N(q_2, b)) = q_3$
  - $\delta_D(q_{12}, b) = E(\delta_N(q_1, b)) \cup E(\delta_N(q_2, b)) = \{q_2, q_3\} = q_{23}$
  - $\delta_D(q_{13}, a) = E(\delta_N(q_1, a)) \cup E(\delta_N(q_3, a)) = \{\emptyset, q_1, q_3\} = q_{13}$

Il DFA equivalente quindi è:



1.5 Espressioni regolari

Definizione di espressione regolare

Dato un alfabeto Σ, un'**espressione regolare** di Σ è una stringa r che rappresenta un linguaggio $L(r) \subseteq \Sigma^*$. L'insieme delle espressioni regolari di un alfabeto Σ, scritte come $\text{re}(\Sigma)$, è definito:

- $\emptyset \in \text{re}(\Sigma)$
- $\varepsilon \in \text{re}(\Sigma)$
- $a \in \text{re}(\Sigma)\ \forall a \in \Sigma$
- $r_1, r_2 \in \text{re}(\Sigma) \implies r_1 \cup r_2 \in \text{re}(\Sigma)$
- $r_1, r_2 \in \text{re}(\Sigma) \implies r_1 \circ r_2 \in \text{re}(\Sigma)$
- $r \in \text{re}(\Sigma) \implies r^* \in \text{re}(\Sigma)$

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

- $0 \cup 1 = \{0\} \cup \{1\} = \{0, 1\}$
- $0^*10^* = \{0\}^* \circ \{1\} \circ \{0\}^* = \{x1y | x, y \in \{0\}^*\}$
- $\Sigma^*1\Sigma^* = \Sigma^* \circ \{1\} \circ \Sigma^* = \{x1y | x, y \in \Sigma^*\}$
- $1^*\emptyset = \{1\}^* \circ \emptyset = \emptyset$
- $\emptyset^* = \varepsilon$

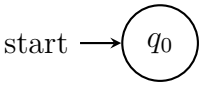
Conversione da espressione regolare a DFA

Date la classe dei linguaggi descritti da un'espressione regolare $\mathcal{L}(\text{re})$ e $\mathcal{L}(\text{DFA})$:

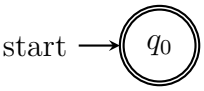
$$\mathcal{L}(\text{re}) \subseteq \mathcal{L}(\text{DFA})$$

Dimostrazione per induzione:

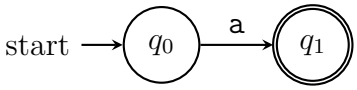
- Caso base:
  - $r = \emptyset \in \text{re}(\Sigma)$, definiamo il DFA $D_\emptyset$:

    in cui $x \in L(r) \iff x \in L(D_\emptyset)$ quindi $L(r) \in \mathcal{L}(\text{DFA})$
  - $r = \varepsilon \in \text{re}(\Sigma)$, definiamo il DFA D_ε:

    in cui $x \in L(r) \iff x \in L(D_\varepsilon)$ quindi $L(r) \in \mathcal{L}(\text{DFA})$
  - $r = a \in \text{re}(\Sigma)$, definiamo il DFA D_a:

    in cui $x \in L(r) \iff x \in L(D_a)$ quindi $L(r) \in \mathcal{L}(\text{DFA})$
- Passo induttivo:
  - $r = r_1 \cup r_2$, allora abbiamo che:

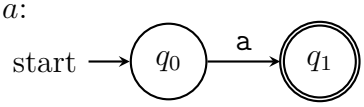
    $$L(r) = L(r_1) \cup L(r_2) = L(D_1) \cup L(D_2) \in \mathcal{L}(\text{DFA})$$
  - $r = r_1 \circ r_2$, allora abbiamo che:

    $$L(r) = L(r_1) \circ L(r_2) = L(D_1) \circ L(D_2) \in \mathcal{L}(\text{DFA})$$
  - $r = r_1^*$, allora abbiamo che:

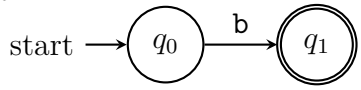
    $$L(r) = L(r_1)^* = L(D_1)^* \in \mathcal{L}(\text{DFA})$$

Esempio:
Data l'espressione regolare $(a \cup ab)^*$, il NFA corrispondente a tale espressione creata partendo dai sotto-componenti:

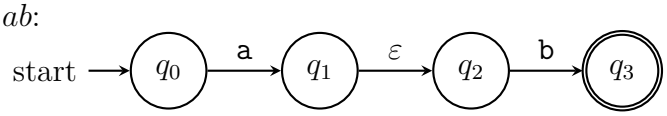
- a:



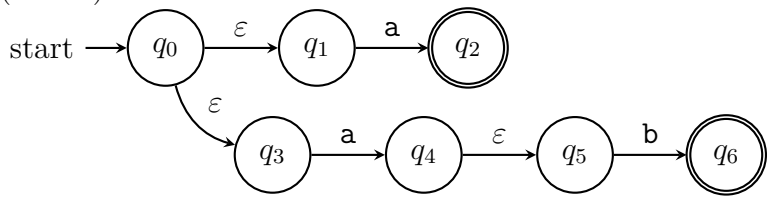
- b:



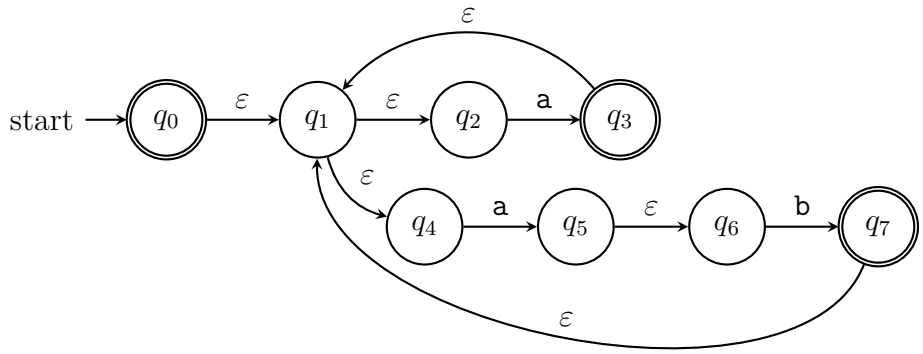
- ab:



- $(a \cup ab)$:



- $(a \cup ab)^*$:



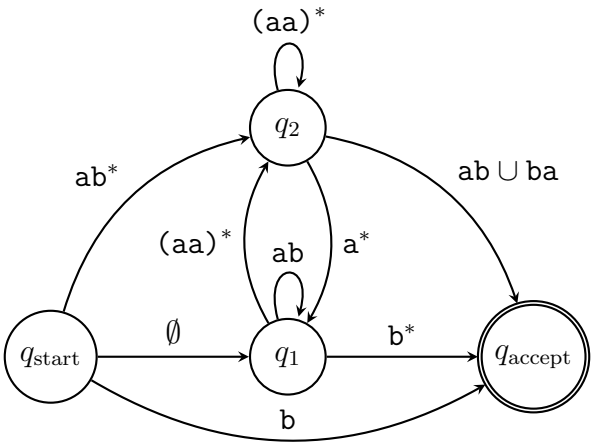
1.5.1 NFA generalizzati (GNFA)

NFA generalizzato (GNFA)

Un **GNFA (Generalized NFA)** è una tupla $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ in cui:

- $|Q| \geq 2$ è l'insieme degli stati dell'automa
- Σ è l'alfabeto dell'automa
- $q_{\text{start}} \in Q$ è lo stato iniziale dell'automa
- $q_{\text{accept}} \in Q$ è l'unico stato accettante dell'automa
- $\delta : (Q - q_{\text{accept}}) \times (Q - q_{\text{start}}) \to \text{re}(\Sigma)$ è la funzione di transizione degli stati in cui però:
  - q_{start} ha solo archi uscenti
  - q_{accept} ha solo archi entranti
  - Per ogni coppia di stati (q_1, q_2), c'è esattamente un arco $q_1 \to q_2$ e un arco $q_2 \to q_1$, incluse le coppie (q, q)
  - Le etichette degli archi sono espressioni regolari

Esempio:



Conversione da DFA a GNFA

Date $\mathcal{L}(\text{DFA})$ e $\mathcal{L}(\text{GNFA})$ si ha che:

$$\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{GNFA})$$

Dimostrazione:
Dato $L \in \mathcal{L}(\text{DFA})$, allora esiste $D = (Q, \Sigma, \delta, q_0, F)$ tale che $L(D) = L$.
Costruisco un GNFA $G = (Q_G, \Sigma, \delta_G, q_{\text{start}}, q_{\text{accept}})$ costruito da D in cui:

- $Q_G = Q \cup \{q_{\text{start}}, q_{\text{accept}}\}$
- $\delta_G(q_{\text{start}}, q_0) = \varepsilon$
- $\forall q \in F \ \delta_G(q, q_{\text{accept}}) = \varepsilon$
- Ogni transizione con etichette multiple in D viene trasformata in un'etichetta con l'unione delle etichette multiple
- Per ogni coppia per cui non ci sia un arco in un verso o nell'altro in D viene aggiunto un arco in G con etichetta $\emptyset$

Quindi $x \in L(D) \implies x \in L(G)$, quindi $\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{GNFA})$.

1.5.2 Riduzione minimale di un GNFA

Dato un GNFA $G = (Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, possiamo usare un'algoritmo per ottenere un GNFA G' con solo due stati (equivalente quindi ad un'espressione regolare) e per cui $L(G) = L(G')$:

Algoritmo: Riduzione minimale di un GNFA

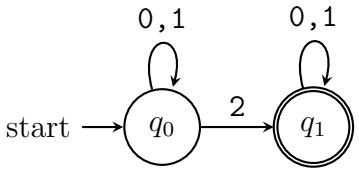
```
def ReduceGNFA(G):
    if |Q|==2 :
        return G
    q = q ∈ Q − {q_start, q_accept}// scelto in modo casuale
    Q' = Q − q
    for q_i in Q' − {q_accept} :
        for q_j in Q' − {q_start} :
            δ'(q_i, q_j) = δ(q_i, q)δ(q, q)*δ(q, q_j) ∪ δ(q_i, q_j)
    G' = (Q', Σ, δ', q_start, q_accept)
    return ReduceGNFA(G')
```

Dimostrazione per induzione:
Induzione in base al numero degli stati k

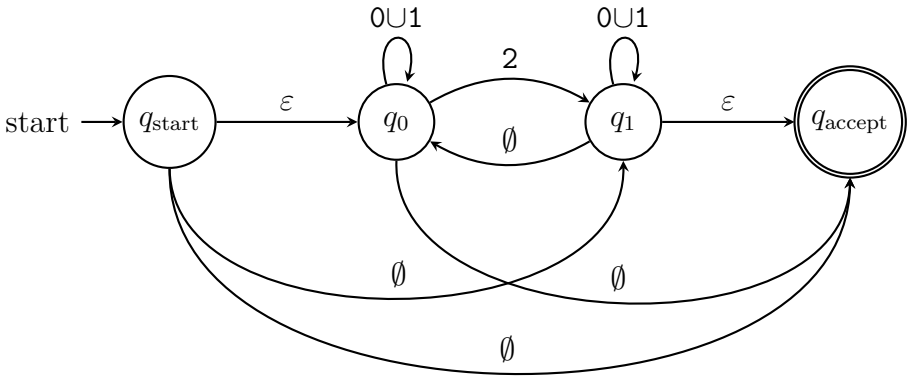
- Caso base:
  $k = 2 \implies G' = G \implies L(G') = L(G)$
- Caso induttivo:
  Assumo che per un G_k sia vero che $L(G_k) = L(G)$
- Passo induttivo:
  Dato $G_{k-1} = G_k - \{q_r\}$ si ha che se G_k accetta una stringa x, allora esiste una serie di stati $q_{\text{start}} q_1 \dots q_{\text{accept}}$ in cui abbiamo due casi:
  - $q_r \notin q_{\text{start}} q_1 \dots q_{\text{accept}}$ allora se $x \in L(G_k) \implies x \in L(G_{k-1})$
  - $q_r \in q_{\text{start}} q_1 \dots q_{\text{accept}}$ allora gli stati q_i e q_j vicini allo stato rimosso saranno collegati da una nuova espressione regolare che per cui $\delta'(q_i, q_j) = \delta(q_i, q_r)\delta(q_r, q_r)^*\delta(q_r, q_j)$ per cui $x \in L(G_k) \implies x \in L(G_{k-1})$

  Allora se $x \in L(G_{k+1})$ allora per ogni coppia (q_i, q_j) l'etichetta rappresenterà tutti i cammini tra q_i e q_j anche in G_k quindi $x \in L(G_{k-1}) \implies x \in L(G_k)$. Quindi alla fine si ha che $L(G) = L(G_k) = L(G_{k-1})$

Esempio:
Dato il seguente DFA:

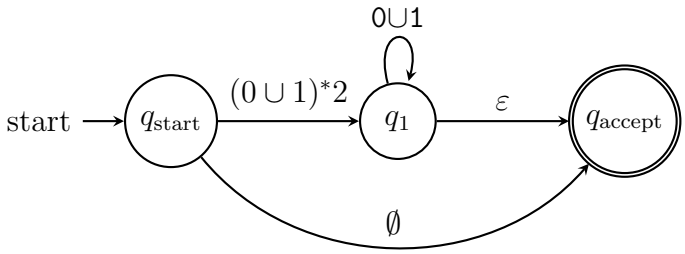


Il GNFA equivalente sarà:



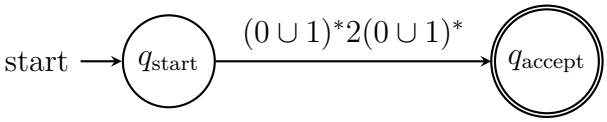
Rimuovendo lo stato q_0 e ricalcolando le transizioni:

- $\delta'(q_{\text{start}}, q_1) = \delta(q_{\text{start}}, q_0)\delta(q_0, q_0)^*\delta(q_0, q_1) \cup \delta(q_{\text{start}}, q_1) = \varepsilon(0 \cup 1) * 2 \cup \emptyset = (0 \cup 1)^*2$
- $\delta'(q_{\text{start}}, q_{\text{accept}}) = \delta(q_{\text{start}}, q_0)\delta(q_0, q_0)^*\delta(q_0, q_{\text{accept}}) \cup \delta(q_{\text{start}}, q_{\text{accept}}) = \varepsilon(0 \cup 1)^*\emptyset \cup \emptyset = \emptyset$
- $\delta'(q_1, q_1) = \delta(q_1, q_0)\delta(q_0, q_0)^*\delta(q_0, q_1) \cup \delta(q_1, q_1) = \emptyset(0 \cup 1)^*2 \cup (0 \cup 1) = 0 \cup 1$
- $\delta(q_1, q_{\text{accept}}) = \delta(q_1, q_0)\delta(q_0, q_0)^*\delta(q_0, q_{\text{accept}}) \cup \delta(q_1, q_{\text{accept}}) = \emptyset(0 \cup 1)^*\emptyset \cup \varepsilon = \varepsilon$



Rimuovendo anche lo stato q_1 e ricalcolando l'ultima transizione:

$$\delta'(q_{\text{start}}, q_{\text{accept}}) = \delta(q_{\text{start}}, q_1)\delta(q_1, q_1)^*\delta(q_1, q_{\text{accept}}) \cup \delta(q_{\text{start}}, q_{\text{accept}}) = (0 \cup 1)^*2(0 \cup 1)^*\varepsilon \cup \emptyset = (0 \cup 1)^*2(0 \cup 1)^*$$



Conversione da GNFA a espressione regolare

Dati $\mathcal{L}(\text{GNFA})$ e $\mathcal{L}(\text{re})$ si ha quindi che:

$$\mathcal{L}(\text{GNFA}) \subseteq \mathcal{L}(\text{re})$$

Equivalenza nella classe dei linguaggi regolari

Per un alfabeto Σ, dai vari lemmi precedenti di conversione di ha che:

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

1.6 Pumping Lemma per linguaggi regolari

Preso un linguaggio L, ad esempio:

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

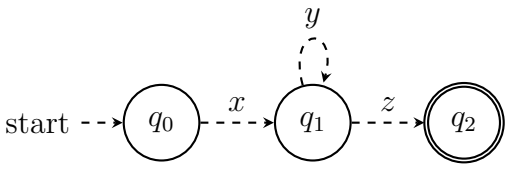
Questo linguaggio avrebbe bisogno di un automa ad infiniti stati per poter sapere il numero di 0 e 1, quindi impossibile con un DFA, NFA o GNFA.

Pumping Lemma per linguaggi regolari

Dato un linguaggio $L \in$ REG, allora esiste un DFA D con p stati tale che $L(D) = L$.
$\forall w \in L$, con $|w| \geq p$, è ovvio che alcuni stati devono ripetersi e dividendo $w = xyz$ con x, y, z sottostringhe di w si ha che:

1. $xy^k z \in L \; \forall k \geq 0$
2. $|y| > 0$
3. $|xy| < p$

Graficamente, con le freccie tratteggiate che indicano che in mezzo possono esserci altri stati:



Dimostrazione:
Dato $L \in$ REG e il DFA D per cui $L(D) = L$, consideriamo $p = |Q|$ e $w = w_1 \dots w_n$, con $n \geq p$, allora esiste una sequenza di stati $q_1 \dots q_{n+1}$ in cui:

$$\delta(q_k, w_k) = q_{k+1} \quad \forall k$$

La sequenza è lunga $n+1 \geq p+1$ perciò tra i primi $p+1$ stati ci deve essere almeno uno stato ripetuto. Siano q_i e q_j le due occorrenze di questo stato ripetuto, con $i < j \leq p+1$. Dividiamo la stringa in:

- $x = w_1 \dots w_{i-1} \implies \delta^*(q_1, x) = q_i$
- $y = w_i \dots w_{j-1} \implies \delta^*(q_i, y) = q_j = q_i$
- $z = w_j \dots w_n \implies \delta^*(q_j, z) = q_{n+1} \in F$

Quindi $\delta(q_1, xy^k z) = q_{n+1} \in F \implies xy^k z \in L$.
Inoltre sapendo che $i \neq j \implies |y| > 0$ e $j \leq p+1 \implies |xy| \leq p$.

1.6.1 Dimostrazione di non regolarità tramite pumping lemma

Il pumping lemma viene usato per dimostrare che una linguaggio non è regolare, per farlo basta trovare una stringa che non possa essere scomposta secondo il pumping lemma.
Esempio:
Dato il linguaggio:

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

Preso un valore p del pumping lemma, scelgo la stringa $w = 0^p 1^p$ con quindi $|w| = 2p > p$, visto che $|xy| < p \implies y$ è composta da solo 0, questo vuol dire che con indice $i = 2$:

$$xy^2z = 0^q 1^p \; q > p \implies xy^2z \notin L$$

Quindi il linguaggio non è regolare.

2

Linguaggi acontestuali

2.1 Grammatiche acontestuali

Grammatica acontestuale (CFG)

Una **CFG (Context-free Grammar)** è una tupla (V, Σ, R, S) in cui:

- V è l'insieme delle **variabili**
- Σ è l'insieme dei **terminali** per cui $V \cap \Sigma = \emptyset$
- R è l'insieme delle **regole** nella forma $R : V \to (V \cup \Sigma)^*$
- S è la variabile iniziale della grammatica

Esempio:
Data la grammatica $G = (\{A, B\}, \{0, 1, \#\}, R, A)$ in cui R ha le seguenti regole:

- $A \to 0A1$
- $A \to B$
- $B \to \#$

Le regole possono essere scritte in modo contratto nel caso in cui una variabile A abbia diverse regole $A \to X_1, A \to X_2, \ldots, A \to X_n$, scrivendo $A \to X_1|X_2|\ldots|X_n$. Nel caso sopra per esempio avremmo potuto scrivere $A \to 0A1|B$.

Produzione e derivazione

Data una grammatica $G = (V, \Sigma, R, S)$, u, v, w stringhe di variabili o terminali ed esiste una regola $A \to w$, allora la stringa uAv **produce** la stringa uwv, scritto come:

$$uAv \Rightarrow uwv$$

Se invece esistono un insieme di stringhe $u_1, \ldots, u_k$ tale che:

$$u \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$$

allora la stringa u **deriva** v, scritto come:

$$u \overset{*}{\Rightarrow} v$$

Esempio:
Data la grammatica con le regole:

$$E \to E + E | E \cdot E | (E) | 0 | \dots | 9$$

Possiamo derivare la stringa:

$$E \Rightarrow E \cdot E \Rightarrow (E) \cdot E \Rightarrow (E + E) \cdot E \Rightarrow (3 + 4) \cdot 5$$

Linguaggio di una grammatica

Data una grammatica $G = (V, \Sigma, R, S)$, il linguaggio generato da G è l'insieme delle stringhe che si possono derivare da S:

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

Esempio:
Data la CFG $G = (\{S\}, \{a, b\}, R, S)$ con le regole:

$$S \to \varepsilon | aSb | SS$$

allora:

- $S \Rightarrow aSb \Rightarrow a\varepsilon b = ab \implies ab \in L(G)$
- $S \Rightarrow SS \Rightarrow aSbaSb \Rightarrow a\varepsilon ba\varepsilon b = abab \implies abab \in L(G)$

Classe dei linguaggi acontestuali

Dato un alfabeto Σ, la **classe dei linguaggi acontestuali** di Σ, scritta come CFL, è l'insieme dei linguaggi per cui esiste una grammatica che li accetta:

$$\text{CFL} = \{L \subseteq \Sigma^* | \exists CFGG\ L(G) = L\}$$

2.1.1 Grammatiche ambigue

Data una CFG G e una stringa w è possibile che esistano più derivazioni di w, cioè passaggi diversi che portano alla stringa w.
Esempio:
Data la CFG con regole:

$$E \to E + E | E \cdot E | a$$

la stringa $a + a \cdot a$ può essere derivata in due modi:

1. $E \Rightarrow E + E \Rightarrow E + E \cdot E \Rightarrow a + a \cdot a$
2. $E \Rightarrow E \cdot E \Rightarrow E + E \cdot E \Rightarrow a + a \cdot a$

Derivazione a sinistra e grammatica ambigua

Data una CFG G e una stringa w, una **derivazione a sinistra** è una derivazione $S \overset{*}{\Rightarrow} w$ in cui ad ogni passo viene sostituita la variabile più a sinistra. Questo tipo di derivazione diminuisce le derivazioni multiple, ma ci sono comunque alcuni casi in cui può esistere più di una derivazione per una stringa.
Se una grammatica ha almeno una stringa con derivazioni a sinistra multiple allora la grammatica si dice **ambigua**.

2.2 Forma normale di una grammatica

Forma normale di una grammatica

Una CFG G è in **forma normale (CNF)** se tutte le regole sono della forma:

$$A \to BC$$
$$A \to a$$
$$S \to \varepsilon$$

in cui $A \in V, a \in \Sigma$ e $B, C \in V - \{S\}$

Trasformazione di una CFG in forma normale

Data una CFG G è sempre possibile trasformarla in una CFG G' in forma normale per cui:

$$L(G) = L(G')$$

Dimostrazione:
Data una CFG $G = (V, \Sigma, R, S)$, la CFG G' in forma normale si ottiene con i passaggi:

1. Aggiungo una variabile iniziale S_0 e una regola $S_0 \to S$
2. Elimino le ε-regole, cioè quelle nella forma $A \to \varepsilon$ con $A \neq S_0$ e per ogni regola contente A nella parte destra aggiungo una regola con quella occorrenza eliminata. Se più di una A è presente nella parte destra allora aggiungo una regola per ogni combinazione di occorrenze eliminate.
   Per esempio se elimino la regola $A \to \varepsilon$ e esiste la regola $B \to uAvAw$ allora aggiungerò le regole $B \to uvAw|uAvw|avw$
3. Elimino le regole unitarie, cioè nella forma $A \to B$ e per ogni regola della forma $B \to u$ in cui u è una stringa di variabili e terminali, aggiungo una regola $A \to u$
4. Per ogni regola $A \to v_1 v_2 \dots v_k$ in cui v è una variabile o terminale singolo e $k \geq 3$, creo una serie di variabili $A_1, \dots, A_{k-2}$ per cui:

   $$\begin{aligned} A &\to v_1 A_1 \\ A_1 &\to v_2 A_2 \\ &\dots \\ A_{k-1} &\to v_{k-1} v_k \end{aligned}$$

   ed elimino la regola $A \to v_1 v_2 \dots v_k$
5. Per ogni regola nella forma $A \to aB$ oppure $A \to Ba$, si crea una nuova variabile U e la regola $U \to a$ e si sostituisce la regola $A \to aB$ con $A \to UB$ oppure la regola $A \to Ba$ con $A \to BU$

Esempio:
Data la CFG con stato iniziale S e regole:

$$\begin{aligned} S &\to ASA|aB \\ A &\to B|S \\ B &\to b|\varepsilon \end{aligned}$$

Per ceare la CFG in forma normale corrispondente seguo i passaggi:

1. Aggiungo la variabile iniziale S_0 e la regola $S_0 \to S$:

   $$\begin{aligned} \boldsymbol{S_0} &\boldsymbol{\to S} \\ S &\to ASA|aB \\ A &\to B|S \\ B &\to b|\varepsilon \end{aligned}$$

2. Elimino le ε-regole:

   2.1 Elimino $B \to \varepsilon$:

$$
\begin{aligned}
&S_0 \to S\\
&S \to ASA|aB|\boldsymbol{a}\\
&A \to B|S|\boldsymbol{\varepsilon}\\
&B \to b|\not{\varepsilon}
\end{aligned}
$$

   2.2 Elimino $A \to \varepsilon$:

$$
\begin{aligned}
&S_0 \to S\\
&S \to ASA|aB|a|\boldsymbol{AS}|\boldsymbol{SA}|\boldsymbol{S}\\
&A \to B|S|\not{\varepsilon}\\
&B \to b
\end{aligned}
$$

3. Elimino le regole unitarie:

   3.1 Elimino $S \to S$ e $S_0 \to S$:

$$
\begin{aligned}
&S_0 \to \not{S}|\boldsymbol{ASA}|\boldsymbol{aB}|\boldsymbol{a}|\boldsymbol{AS}|\boldsymbol{SA}\\
&S \to ASA|aB|a|AS|SA|\not{S}\\
&A \to B|S\\
&B \to b
\end{aligned}
$$

   3.2 Elimino $A \to B$ e $A \to S$:

$$
\begin{aligned}
&S_0 \to ASA|aB|a|AS|SA\\
&S \to ASA|aB|a|AS|SA\\
&A \to \not{B}|\not{S}|\boldsymbol{b}|\boldsymbol{ASA}|\boldsymbol{aB}|\boldsymbol{a}|\boldsymbol{AS}|\boldsymbol{SA}\\
&B \to b
\end{aligned}
$$

4. Separo le regole con tre o più elementi a destra e le divido in regole con due elementi a destra:

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

5. Converto le regole con due elementi a destra di cui almeno uno un terminale:

$$
\begin{aligned}
&S_0 \to AC|\cancel{aB}|\boldsymbol{UB}|a|AS|SA\\
&S \to AC|\cancel{aB}|\boldsymbol{UB}|a|AS|SA\\
&A \to b|AC|\cancel{aB}|\boldsymbol{UB}|a|AS|SA\\
&C \to SA\\
&\boldsymbol{U \to a}\\
&B \to b
\end{aligned}
$$

2.3 Automi a Pila

Automa a Pila (PDA)

Un **PDA (Push-Down Automata)** è una tupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$ in cui:

- Q è l'insieme degli stati dell'automa
- Σ è l'alfabeto dell'automa
- Γ è l'alfabeto della pila (stack)
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ è la funzione di transizione per cui se $(q_2, c) \in \delta(q_1, a, b)$ allora:
  l'automa legge un simbolo $a \in \Sigma_\varepsilon$ e se in cima allo stack c'è $b \in \Gamma_\varepsilon$ passa dallo stato q_1 a q_2 e sostituisce b con c. Viene scritto come $a; b \to c$
- q_0 è lo stato iniziale
- F è l'insieme degli stati accettanti

Un PDA tramite la funzione δ, se $(q_2, c) \in \delta(q_1, a, b)$ ci sono diversi casi:

- $a, b \neq \varepsilon, c = \varepsilon$ $(a; b \to \varepsilon)$: l'automa legge a e se in cima allo stack c'è b passa dallo stato q_1 allo stato q_2 e rimuove b dallo stack (pop)
- $a, c \neq \varepsilon, b = \varepsilon$ $(a; \varepsilon \to c)$: l'automa legge a e passa dallo stato q_1 allo stato q_2 e aggiunge c allo stack (push)
- $a \neq \varepsilon, b, c \neq \varepsilon$ $(a; \varepsilon \to \varepsilon)$: l'automa legge a e passa dallo stato q_1 allo stato q_2 senza modificare lo stack

Una stringa $w = w_1 \dots w_k$ è accettata da un PDA se esiste una serie di stati $q_0 \dots q_{k+1}$ e una serie di stringhe $s_1 \dots s_n$ per cui:

- $s_0 = \varepsilon$ (pila vuota)
- $\forall i (q_{i+1}, a) \in \delta(q_i, w_i, b)$ con $s_i = bt$ e $s_{i+1} = at$
- $q_{k+1} \in F$

Passo di computazione di un PDA

Un **passo di computazione** è una relazione binaria con simbolo $\vdash_P$ per cui:

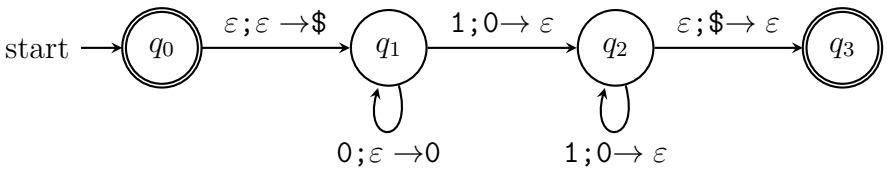
$$(q_1, ax, by) \vdash_P (q_2, x, cy) \iff (q_2, c) \in \delta(q_1, a, b)$$

La **chiusura transitiva** di $\vdash_P$, scritta come $\vdash_P^*$ definisce la transizione estesa per cui:

$$L(P) = \{w \in \Sigma^* | (q_0, w, \varepsilon) \vdash_P^* (q, \varepsilon, y) \land q \in F\}$$

Questa definizione sarebbe equivalente se sostituissimo (q, ε, y) con $(q, \varepsilon, \varepsilon)$, perchè è sempre possibile svuotare lo stack senza cambiare il linguaggio.

Esempio:
Dato il linguaggio $L = \{0^n 1^n | n \geq 0\}$, il PDA equivalente è:



Linguaggi accettati da PDA

Dato un alfabeto Σ, definiamo l'insieme dei linguaggi accettati da un PDA come:

$$\mathcal{L}(PDA) = \{L \subseteq \Sigma^* | \exists PDA P \; L(P) = L\}$$

2.4 Equivalenza tra CFG e PDA

Conversione da CFG a PDA

Date le due classi di linguaggi CFL e $\mathcal{L}$(PDA), si ha che:

$$\text{CFL} \subseteq \mathcal{L}(\text{PDA})$$

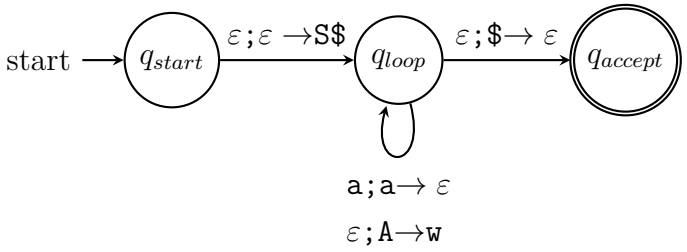
Dimostrazione:
Dato un linguaggio $L \in$ CFL, esiste una grammatica $G = (V, \Sigma, R, S) | L(G) = L$.
Costruiamo un PDA $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$ tale che:

- $Q = (q_{start}, q_{loop}, q_{accept}) \cup E$, per cui E sono gli stati aggiunti per far si che δ esegua un push o un pop di un solo simbolo alla volta
- $\Gamma = V \cup \Sigma$
- $F = q_{accept}$
- δ è una funzione per cui:
  - $\delta(q_{start}, \varepsilon, \varepsilon) = (q_{loop}, S\$)$
  - $\forall A \in V \; \delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w) | (A \to w) \in R \wedge u \in \Gamma^*\}$
  - $\forall a \in \Sigma \; \delta(q_{loop}, a, a) = (q_{loop}, \varepsilon)$
  - $\delta(q_{lopp}, \varepsilon, \$) = (q_{accept}, \varepsilon)$

Quindi si ha che:

$$w \in L(G) \iff w \in L(P)$$

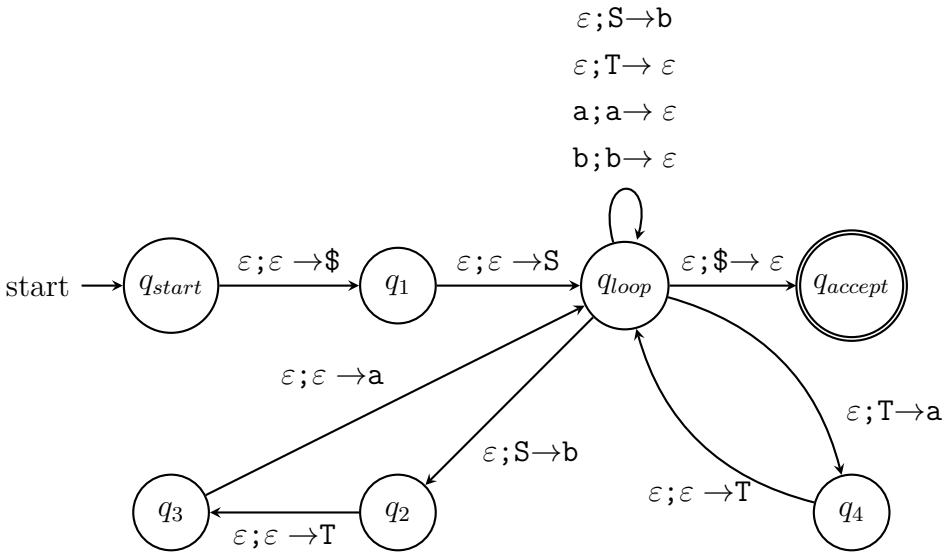
Graficamente un PDA schematizzato secondo queste regole:



Esempio:
Data la grammatica G con regole:

$$S \to aTb|b$$
$$T \to Ta|\varepsilon$$

Il PDA associato sarà:



Conversione da PDA a CFG

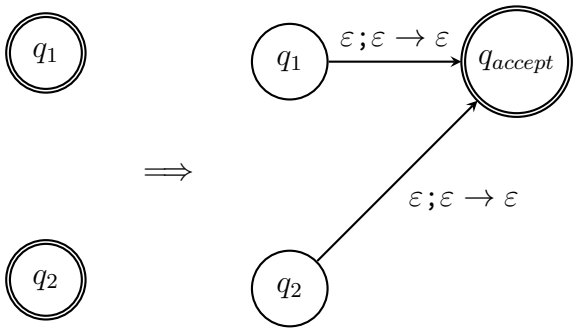
Date le due classi di linguaggi CFL e $\mathcal{L}$(PDA), si ha che:

$$\mathcal{L}(\text{PDA}) \subseteq \text{CFL}$$

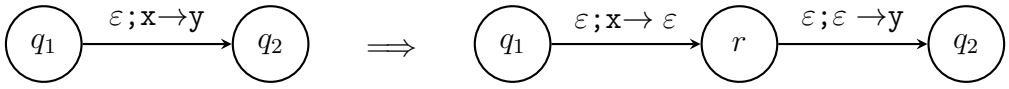
Dimostrazione:
Dato un linguaggio $L \in \mathcal{L}$(PDA), esiste un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)|L(P) = L$.
Trasformiamo P scrivendolo in una forma "canonica" per cui:

- Il PDA inizia e finisce la computazione con la pila vuota, quindi con:

- Ha un solo stato accettante, quindi:



- Il PDA esegue solo un push o pop alla volta (non contemporaneamente), quindi:



Creiamo una CFG $G = (V, \Sigma, R, S)$ tale che:

- $V = \{A_{p,q} | p, q \in Q\}$
- $S = A_{q_0, q_{accept}}$

In cui la variabile $A_{p,q}$ è variabile che deriva tutte le stringe generabili passando dallo stato p allo stato q con stack vuoto, per cui:

- $\forall p \in Q$:
$$(A_{p,p} \to \varepsilon) \in R$$
- $\forall p, q, r \in Q$, nel caso in cui lo stack si svuoti in mezzo nello stato r:
$$(A_{p,q} \to A_{p,r} A_{r,q}) \in R$$
- $\forall p, q, r, s \in Q, u \in \Gamma, a, b \in \Sigma_\varepsilon$, nel caso in cui lo stack non si svuoti in mezzo:
$$(A_{p,q} \to a A_{r,s} b) \in R \iff (r, u) \in \delta'(p, a, \varepsilon) \land (q, \varepsilon) \in \delta(s, b, u)$$

Derivazione con stack vuoto

Dati $p, q \in Q, x \in \Sigma^*$, allora:

$$A_{p,q} \overset{*}{\Rightarrow} x \iff x \text{ porta } P \text{ dallo stato } p \text{ allo stato } q \text{ con stack vuoto}$$

Dimostrazione per doppia implicazione:

1. $A_{p,q} \overset{*}{\Rightarrow} x \implies x$ porta P dallo stato p allo stato q con stack vuoto:
Dimostriamo per induzione sul numero di passi della produzione di x in G:
   - Caso base:
   Con 1 passo l'unica regola possibile per cui $A_{p,q} \overset{*}{\Rightarrow} x$ è la regola $A_{p,q} \to \varepsilon$, quindi è vero che x porta P dallo stato p allo stato $q = p$ con stack vuoto

- Ipotesi induttiva:
  Supponiamo che per ogni stringa x con computazione di passi$\leq k$ per cui $A_{p,q} \overset{*}{\Rightarrow} x$, x porti P da p a q con stack vuoto
- Passo induttivo:
  Con una stringa x con computazione di $k+1$ passi ha due opzioni in base alle due regole possibili:
  1. $A_{p,q} \to aA_{r,s}b$:
     Sia $x = ayb$, per cui $A_{r,s} \overset{*}{\Rightarrow} y$ con un numero di passi $k-1$, per cui per l'ipotesi induttiva y porta P da r a s con stack vuoto. Per costruzione di G:

     $$(A_{p,q} \to aA_{r,s}b) \in R \iff (r,u) \in \delta'(p,a,\varepsilon) \wedge (q,\varepsilon) \in \delta(s,b,u)$$

     Quindi $ayb = x$ porta P da p a q con stack vuota.
  2. $A_{p,q} \to A_{p,r}A_{r,q}$:
     Sia $x = yz$, per cui $A_{p,r} \overset{*}{\Rightarrow} y$ e $A_{r,q} \overset{*}{\Rightarrow} z$ con numero di passi $\leq k$, per ipotesi induttiva y porta P da p a r con stack vuoto e z porta P da r a q con stack vuoto. Quindi $yz = x$ porta P da p a q con stack vuoto.

2. $A_{p,q} \overset{*}{\Rightarrow} x \impliedby x$ porta P dallo stato p allo stato q con stack vuoto:
   Dimostriamo per induzione sul numero di transizioni di P mentre legge x:

   - Caso base:
     Con 0 transizioni, l'unico input è $x = \varepsilon$ che porta P da p a p. Quindi per la regola $A_{p,p} \to \varepsilon$ si ha che $A_{p,p} \overset{*}{\Rightarrow} x$.
   - Ipotesi induttiva:
     Supponiamo che per ogni stringa che porta P da p a q con stack vuoto con numero di transizioni $\leq k$, sia vero che $A_{p,q} \overset{*}{\Rightarrow} x$
   - Passo induttivo:
     Con una stringa x che porta P da p a q con stack vuoto con $k+1$ transizioni ci sono due opzioni possibili in base a se lo stack si svuota o no in mezzo:
     1. Lo stack si suota solo in p e q:
        Allora il simbolo $u \in \Gamma$ inserito nella prima transizione è lo stesso che viene tolto dallo stack nell'ultima. Siano quindi $a, b \in \Sigma_\varepsilon$ i simboli letti nella prima e ultima transizione allora ci sono due stati $r, a \in Q$ per cui vale:

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

        Sia $x = ayb$ per cui x porta P da p a q con stack vuoto, per far si che ciò avvenga allora y deve portare P da r a s con stack vuoto. La computazione di y ha $k-1 < k$ transizioni quindi per ipotesi induttiva $A_{r,s} \overset{*}{\Rightarrow} y$ e allora $A_{p,q} \Rightarrow aA_{r,s}b \overset{*}{\Rightarrow} x$.
     2. Lo stack si svuota in uno stato intermedio r:
        Sia $x = yz$ per cui y porta P da p a r con stack vuoto e z porta P da r a q con stack vuoto, percorrendo entrambe un numero di transizioni minore di k, quindi per ipotesi induttiva $A_{p,r} \overset{*}{\Rightarrow} y$ e $A_{r,q} \overset{*}{\Rightarrow} z$ e allora $A_{p,q} \Rightarrow A_{p,r}A_{r,q} \overset{*}{\Rightarrow} x$.

Concludiamo quindi che:

$$x \in L(G) \iff A_{q_0,q_{accept}} \overset{*}{\Rightarrow} x \iff x \in L(P)$$

Equivalenza tra CFG e PDA

Date i due lemmi precedenti abbiamo quindi che:

$$\mathcal{L}(\text{PDA}) = \text{CFL}$$

2.5 Pumping lemma per i linguaggi acontestuali

Albero di derivazione di una CFG

Data una CFG $G = (V, \Sigma, R, S)$ in forma normale, una stringa $x \in L(G)$ con l'albero di derivazione con altezza h (altezza del ramo di computazione più lungo), si ha che:

$$|x| \leq 2^{h+1}$$

Dimostrazione per induzione:
Dimostriamo per induzione sull'altezza dell'albero h:

- Caso base:
  Con $h = 1$, l'unica regola possibile in forma normale è $S \to a$, per cui $x = a$ e $|x| \leq 2^{1-1} = 1$
- Ipotesi induttiva:
  Suppongo per ogni stringa x con albero di derivazione di altezza k che $|x| \leq 2^{k-1}$
- Passo induttivo:
  Con una stringa x con albero di derivazione di altezza $k+1$, la prima produzione deve essere di tipo $S \to AB$. I due sottoalberi generati da B e da C hanno altezza k per cui per ipotesi induttiva le stinghe generate hanno lunghezza $le2^{k-1}$, quindi:

$$|x| \leq 2^{k-1} \cdot 2 = 2^h$$

Pumping lemma per linguaggi acontestuali

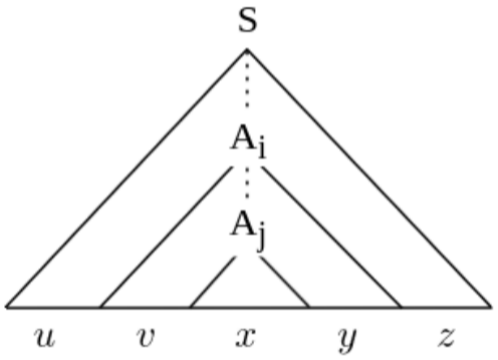
Dato un linguaggio $L \in \text{CFL}$, allora esiste p per cui per ogni w con $|w| \geq p$, scomponendo $w = uvxyz$ si ha che:

1. $uv^kxy^kz \in L \; \forall k \geq 0$
2. $|vy| > 0$
3. $|yxy| \leq p$

Dimostrazione:
Dato un linguaggio $L \in$ CFL, esiste una CFG in forma normale $G = (V, \Sigma, R, S) | L(G) = L$.
Sia $p = 2^m$ con $m = |V|$ e w una stringa con $|w| \geq p$, l'albero di derivazione di w avrà altezza $h \geq m+1$. Il numero di nodi nel ramo di computazione di altezza $m+1$ passa per $m+2$ nodi di cui $m+1$ variabili e 1 teminale. Essendo $|V| = m$, esiste almeno una variabile che si ripete, che denominiamo $A_i = A_j$.
La sottostringhe di w saranno quindi derivate:

- $S \overset{*}{\Rightarrow} uA_iz$
- $A_i \overset{*}{\Rightarrow} vA_jy$
- $A_j \overset{*}{\Rightarrow} x$

Graficamente:



Se nelle formule precendi sostituiamo A_j con A_i possiamo dire che:

$$S \overset{*}{\Rightarrow} uA_iz \overset{*}{\Rightarrow} uvA_iyz \overset{*}{\Rightarrow} uv^kA_iy^kz \overset{*}{\Rightarrow} uv^kxy^kz$$

quindi $uv^kxy^kz \in L(G)\ \forall k \geq 0$.
Visto che G è in forma normale il sottoalbero di A_i deve essere stato creato tramite una regola del tipo $A_i \to BC$ in cui ci sono due casi:

- $B \overset{*}{\Rightarrow} vA_i \wedge C \overset{*}{\Rightarrow} y$
- $B \overset{*}{\Rightarrow} v\wedge \overset{*}{\Rightarrow} A_iy$

Non esistendo ε-regole, si ha che $|vy| > 0$.
Prendendo il ramo di computazione con altezza $m+1$ che produce stringhe di lunghezza massima $2^{m+1-1} = 2^m = p$ allora:

$$|vxy| \leq 2^m = p$$

2.5.1 Dimostrazione di non acontestualità tramite pumping lemma

Il pumping lemma viene usato per dimostrare che un linguaggio non è acontestuale, per farlo basta trovare una stringa che non possa essere scomposta tramite il pumping lemma.
Esempio:

1. Dimostrare che il linguaggio:
$$L = \{0^n 1^n 2^n | n \geq 0\}$$
non è acontestuale.
Suppongo che il linguaggio sia acontestuale, quindi esiste p per cui $\forall w \in L$ con $|w| \geq p$, vale il pumping lemma.
Sia $w = 0^p 1^p 2^p$ con $|w| = 3p$, scompongo $w = uvxyz \in L$ in cui vxy possono essere in 2 modi:

   - vxy contiene un solo simbolo: allora uv^0xy^0z ha meno elementi di quel determinato simbolo rispetto agli altri, quindi $vxy \notin L$
   - vxy contiene due simboli: allora uv^0xy^0z ha meno elementi di quei due simboli rispetto agli altri, quindi $vxy \notin L$

   Non esistendo nessuna scomposizione di w per cui vale il pumping lemma allora $L \notin$ CFL.

2. Dimostrare che il linguaggio
$$L = \{ww | w \in \{0,1\}^*\}$$
non è acontestuale:
Suppongo che il linguaggio sia acontestuale, quindi esiste p per cui $\forall w \in L$ con $|w| \geq p$, vale il pumping lemma.
Sia $x = 0^p 1^p 0^p 1^p$ con $|w| = 4p$, scompongo $x = uvxyz \in L$ in cui vxy possono essere in 3 modi:

   - vxy contiene solo un simbolo: allora uv^0xy^0z ha una metà con meno elementi di quel simbolo rispetto all'altra metà, quindi $uv^0xy^0z \notin L$
   - vxy contiene sia 0 che 1 di una delle due metà: allora uv^0xy^0z ha il centro spostato e le due metà saranno diverese, quindi $uv^0xy^0z \notin L$
   - vxy contiene gli 1 della prima metà e gli 0 della seconda: allora uv^0xy^0z ha la prima metà con meno 1 della seconda e la seconda metà con meno 0 della prima, quindi $uv^0xy^0z \notin L$

   Non esistendo nessuna scomposizione di w per cui vale il pumping lemma allora $L \notin$ CFL.

2.6 Chiusura dei linguaggi acontestuali

Chiusura dell'unione in CFL

L'unione è chiusa in CFL, cioè:

$$\forall L_1, \dots, L_k \in \text{CFL} \implies \bigcup_{i=1}^{k} L_i \in \text{CFL}$$

Dimostrazione:
Dati $L_1, \dots, L_k \in \text{CFL}$, allora esistono le CFG $G_1, \dots, G_k$ per cui $G_i = (V_i, \Sigma_i, R_i, S_i) | L(G_i) = L_i$.
Creo la CFG $G = (V, \Sigma, R, S)$ in cui:

- S è una nuova variabile iniziale
- $V = \bigcup_{i=0}^{k} V_i \cup \{S\}$
- $\Sigma = \bigcup_{i=0}^{k} \Sigma_i$
- $R = \bigcup_{i=0}^{k} R_i \cup \{S \to S_j | j \in [1, k]\}$

Presa una strina $w \in \bigcup_{i=1}^{k} L_i$, allora $\exists j | w \in L_j$, ma visto che esiste $(S \to S_j) \in R$ allora:

$$w \in L_j \implies S_j \overset{*}{\Rightarrow} w \implies S \Rightarrow S_j \overset{*}{\Rightarrow} w \implies w \in L(G)$$

Data invece $w \in L(G)$, le uniche regole applicabili su S sono $\{S \to S_j | j \in [1, k]\}$, allora:

$$w \in L(G) \implies \exists j | S \Rightarrow S_j \overset{*}{\Rightarrow} w \implies w \in L_j \subseteq \bigcup_{i=1}^{k} L_i$$

Quindi:

$$L_1 \cup \dots \cup L_k = L(G_1) \cup \dots \cup L(G_k) = L(G) \in \text{CFL}$$

Chiusura della concatenazione in CFL

La concatenazione è chiusa in CFL, cioè:

$$\forall L_1, \dots, L_k \in \text{CFL} \implies L_1 \circ \dots \circ L_k \in \text{CFL}$$

Dimostrazione:
Dati $L_1, \dots, L_k \in$ CFL, allora esistono le CFG $G_1, \dots, G_k$ per cui $G_i = (V_i, \Sigma_i, R_i, S_i) | L(G_i) = L_i$.
Creo la CFG $G = (V, \Sigma, R, S)$ in cui:

- S è una nuova variabile iniziale
- $V = \bigcup_{i=0}^{k} V_i \cup \{S\}$
- $\Sigma = \bigcup_{i=0}^{k} \Sigma_i$
- $R = \bigcup_{i=0}^{k} R_i \cup \{S \to S_1 \dots S_k\}$

Presa una strina $w = w_1 \dots w_k \in L_1 \circ \dots \circ L_k$, allora $\forall j \; w_j \in L_j$, ma visto che esiste $(S \to S_1 \dots S_k) \in R$ allora:

$$\forall j \; w_j \in L_j \iff S_j \overset{*}{\Rightarrow} w_j \implies S \Rightarrow S_1 \dots S_k \overset{*}{\Rightarrow} w \implies w \in L(G)$$

Data invece $w \in L(G)$, l'unica regola applicabile su S è $(S \to S_1 \dots S_k)$, allora:

$$w \in L(G) \implies S \overset{*}{\Rightarrow} w = w_1 \dots w_k \implies w \in L_1 \circ \dots \circ L_k$$

Quindi:

$$L_1 \circ \dots \circ L_k = L(G_1) \circ \dots \circ L(G_k) = L(G) \in \text{CFL}$$

Non chiusura dell'intersezione in CFL

L'intersezione **non** è chiusa in CFL, cioè:

$$\exists L_1, L_2 \in \text{CFL} \, | L_1 \cap L_k \notin \text{CFL}$$

Dimostrazione:
Dati i linguaggi:

$$L_1 = \{0^n 1^n 2^i | n, i \geq 0\}$$
$$L_2 = \{0^i 1^n 2^n | n, i \geq 0\}$$

Questi linguaggi sono rappresentati dalle grammatiche:

G_1 :
$S \to AB$
$A \to 0A1 | \varepsilon$
$B \to 2B | \varepsilon$

G_2 :
$S \to AB$
$A \to 0A | \varepsilon$
$B \to 1B2 | \varepsilon$

L'intersezione di questi linguaggi è:

$$L_1 \cap L_2 = \{0^n 1^n 2^n | n \geq 0\}$$

che abbiamo dimostrato precedentemente non essere acontestuale.

Non chiusura del complemento in CFL

Il complemento **non** è chiusa in CFL, cioè:

$$\exists L \in \text{CFL} \, | \overline{L} \notin \text{CFL}$$

Dimostrazione:
Dato il linguaggio:

$$L = \{a, b\}^* - \{ww | w \in \{a, b\}^*\}$$

e la grammatica associata:

$$\begin{aligned} S &\to A|B|AB|BA \\ A &\to a|aAb|bAa|aAa|bAb \\ B &\to b|aBb|bBa|aBa|bBb \end{aligned}$$

Sia $x \in L$ una stringa per cui $|x|$ è dispari, allora $x \in L(G)$.
Sia x una stringa per cui $|x|$ è pari allora dobbiamo dimostrare che:

- $x \in L \implies S \overset{*}{\Rightarrow} x$:
  Scrivo $x = x_1 \dots x_n$, per cui:

  $$x \in L \implies \exists i | x_i \neq x_{i+\frac{n}{2}}$$

  Dividiamo $x = uv$ con $|u|, |v|$ dispari per cui $u = x_1 \dots x_{2i-1}$ e $v = x_{2i} \dots x_n$. Il simbolo centrale di u è x_i mentre quello di v è $x_{i+\frac{n}{2}}$, per cui:

  $$x_i \neq x_{i+\frac{n}{2}} \implies u \neq v$$

  Visto che $|u|, |v|$ sono dispari allora $u, v \in L$, quindi:

  $$S \overset{*}{\Rightarrow} uv = x \implies x \in L(G)$$

- $S \overset{*}{\Rightarrow} x \implies x \in L$:
  Essendo $|x|$ pari, deve essere stata generata dalla regola $S \to AB$ oppure $S \to BA$. Consideriamo il caso in cui $S \to AB$.
  Scriviamo $x = uv$ in cui $u = x_i \dots x_k$ e $v = x_{k+1} \dots x_n$, allora $S \Rightarrow A \overset{*}{\Rightarrow} u$ e $S \Rightarrow B \overset{*}{\Rightarrow} v$ per cui abbiamo $|u|, |v|$ dispari. Il simbolo centrale di u è $x_{\frac{k+1}{2}} = a$ e quello di v è $x_{\frac{k+n+1}{2}} = b$.
  Dividiamo $x = w'w''$ tali che $|w'| = |w''| = \frac{n}{2}$, allora:

  $$w'_{\frac{k+1}{2}} = x_{\frac{k+1}{2}} \neq x_{\frac{k+l+1}{2}} = w''_{\frac{k+1}{2}} \implies w' \neq w'' \implies x = w'w'' \in L$$

Quindi $L = L(G) \in \text{CFL}$.
Il complemento di L è $\overline{L} = \{ww | w \in \{a, b\}^*\}$, che abbiamo dimostrato precedentemente non essere acontestuale.

3

Calcolabilità

3.1 Macchina di Turing (TM)

La **macchina di Turing** è un'estensione del modello di calcolo visto fin'ora, ed è un'astrazione di un computer.
Ha diverse caratteristiche:

- Utilizza un nastro di lavoro (memoria) di lunghezza infinita a destra, dentro cui viene copiato l'input all'inizio della computazione
- Ha una testina di lettura che si sposta a sinistra e destra
- Ha uno stato di acettazione e rifiuto, che però a differenza degli automi sono immediati

Macchina di Turing (TM)

Una **Turing Machine (TM)** è una tupla $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$ in cui:

- Q è l'insieme degli stati della macchina
- Σ è l'alfabeto in input, per cui $\sqcup \notin \Sigma$
- Γ è l'alfabeto del nastro, per cui $\sqcup \in \Gamma$ e $\Sigma \subseteq \Gamma$
- q_{start} è lo stato iniziale
- q_{accept} è lo stato accettante
- q_{reject} è lo stato rifiutante
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ la funzione di transizione per cui se $\delta(q_1, a) = (q_2, b, X)$:
  L'automa legge il simbolo a dal nastro, lo sostituisce con b passando dallo stato q_1 a q_2 e sposta il nastro a destra se $X = R$ e a sinistra se $X = L$. Viene scritto $a \to b; X$.
  Il simbolo $\sqcup$ indica uno spazio vuoto nel nastro.

Configurazione di una TM

Data una TM $M = (Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$ definiamo la stringa

$$uqav$$

come la **configurazione** di M, per cui:

- $u, v \in \Gamma^*$ sono i simboli prima e dopo a nel nastro
- $a \in \Gamma$ è il simbolo in cui sta la testina
- $q \in Q$ è lo stato attuale della macchina

La configurazione iniziale è sempre $q_{start}w$ in cui w è la stringa in input.

Produzione in una TM

Data una TM $M = (Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$, si dice che:

$$uaq_ibv \text{ produce } uq_jacv \iff \delta(q_i, b) = (Q_j, c, L)$$
$$uaq_ibv \text{ produce } uacq_jv \iff \delta(q_i, b) = (Q_j, c, R)$$

Quindi data una stringa w, sarà accettata da M se esiste una serie di configurazioni $c_1, \ldots, c_k$, tali che:

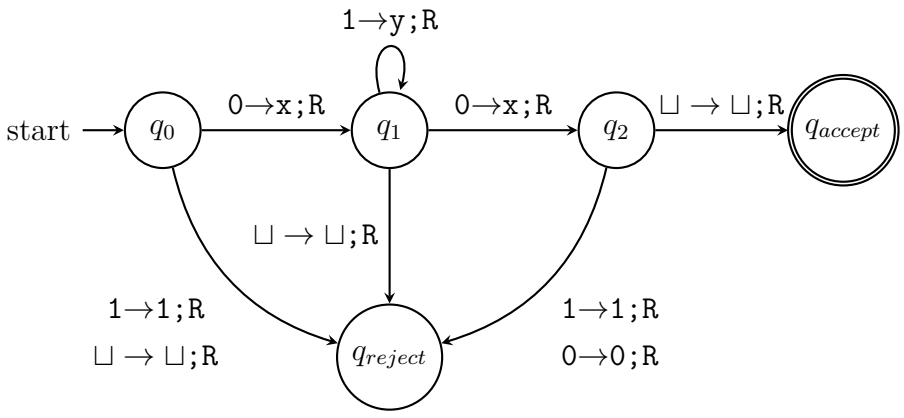
- $c_1 = q_0w$
- c_i produce c_k $\forall i$
- c_k contiene uno stato accettante (q_{accept})

Possiamo inoltre definire il linguaggio:

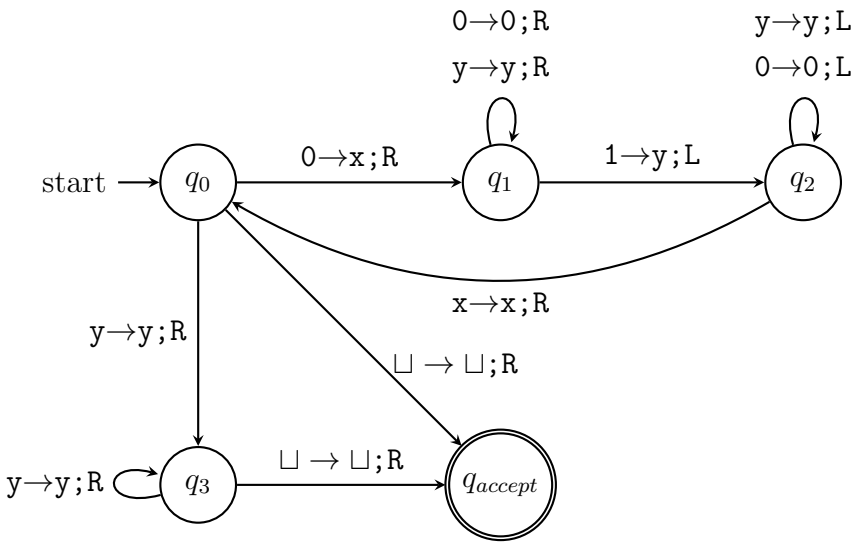
$$L(M) = \{w \in \Sigma^* | M \text{ accetta } w\}$$

Esempi:

1. Dato il linguaggio $L = \{01^*0\}$, la TM che lo riconosce è:



2. Dato il linguaggio $L = \{0^n 1^n | n \geq 0\}$, la TM che lo riconosce è:



Tutte le transizioni omesse vengono considerate come se vanno allo stato q_{reject}.

Classe dei linguaggi Turing-riconoscibili

Dato un alfabeto Σ, la **classe dei linguaggi Turing-riconoscibili** di Σ, scritta come REC, è l'insieme dei linguaggi per cui esiste una macchina di Turing che li accetta:

$$\text{REC} = \{L \subseteq \Sigma^* | \exists \, \text{TM} \, M \; L(M) = L\}$$

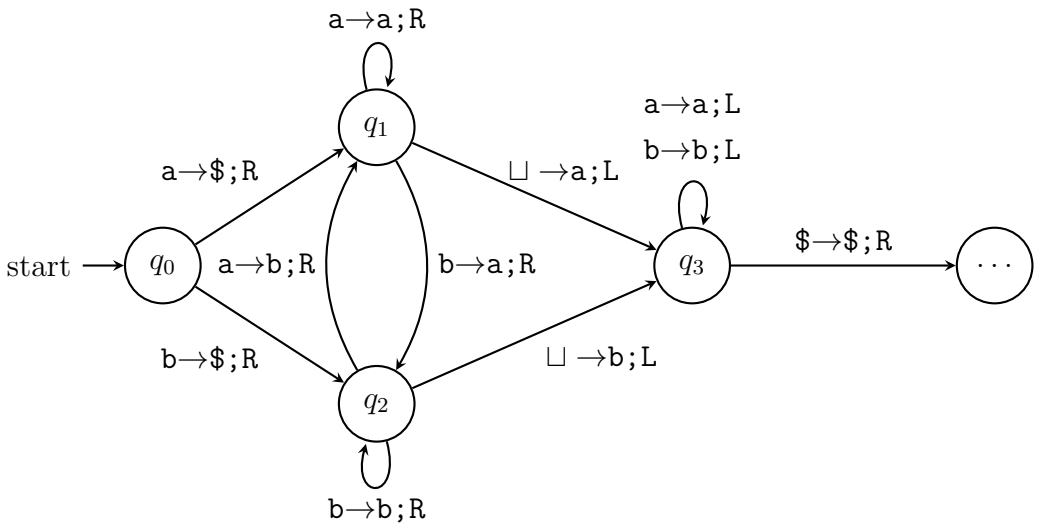
Classe dei linguaggi Turing-decidibili

Una TM M viene definita **decisore** se termina sempre la sua esecuzione, non andando mai in loop.
Dato un alfabeto Σ, la **classe dei linguaggi Turing-decidibili** di Σ, scritta come DEC, è l'insieme dei linguaggi per cui esiste una macchina di Turing che li decide:

$$\text{REC} = \{L \subseteq \Sigma^* | \exists \, \text{TM} \;\; \text{decisore} \; M \; L(M) = L\}$$

Le macchine di Turing hanno un nastro illimitato a destra, quindi è facile capire dove finisce la stringa in input (è semplicemente seguita da infiniti $\sqcup$), ma se si volesse capire la fine del nastro a sinistra bisognerebbe marcarlo in qualche modo.

L'automa che permette di marcare l'inizio del nastro con un simbolo $ spostando tutta la stringa in input di uno spazio più a destra cambia in base all'alfabeto Σ, nel caso l'alfabeto sia $\Sigma = \{a, b\}$ è:



Lo stato indicato dai puntini di sospensione (...), indica un possibile inizio di un automa che necessita di marcare il limite sinistro del nastro.
Se si volesse invece marcare un simbolo sul nastro, senza però modificarne il significato (potrebbe servire per calcoli successivi), basta aggiungere all'alfabeto Γ una versione marcata di ogni simbolo in Σ da sostituire per marcarli. Per esempio se l'alfabeto è $\Sigma = \{a, b, c\}$ aggiungiamo a Γ la versione marcata di ogni simbolo facendolo diventare $\Gamma = \{a, b, c, \sqcup, \dot{a}, \dot{b}, \dot{c}\}$.

3.1.1 Varianti di TM

TM stazionaria

Una **TM stazionaria** è una TM che permette alla testina di rimenere ferma dopo una transizione, la funzione di transizione δ sarà quindi definita come:

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

in sui S sta appunto per la possibilità della testina di restare ferma.
Per ogni TM stazionaria M' esiste una TM M equivalente.

Dimostrazione:
Qualsiasi transizione in M' con spostamento R o L è presente anche in M.
Le transizioni in M' con spostamento S, possono essere simulate creando una serie di regole che permette di muoversi in una direzione senza modificare il nastro. Una transizione in M' del tipo:

$$\delta(q, a) = (p, b, S)$$

è equivalente alle transizioni in M:

$$\delta(q, a) = (r, b, L)$$
$$\delta(r, x) = (p, x, R)\ \forall x \in \Gamma$$

TM multinastro

Una **TM multinastro** è una TM con k nastri, ognuno con la propria testina, e con la possibilità ad ogni transizione di aggiornare tutti i nastri, la funzione di transizione δ sarà quindi definita come:

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

Per ogni TM multinastro M' esiste una TM M equivalente.

Dimostrazione:
Definisco in M un simbolo $\#\notin \Gamma'$ che utilizzo per separare i nastri, in modo da poterli rappresentare su un solo nastro. Per segnare la posizione di tutte le testine invece aggiungo una versione marcata per ogni simbolo in Γ'. La configurazione iniziale di M, considerando che l'input in M' viene posizionato sul primo nastro, sarà:

$$\#\dot{w}_1 \dots w_n \# \dot{\sqcup} \# \dots \# \dot{\sqcup} \#$$

Un passo di computazione invece scorrerà tutto il nastro e controllerà tutti i simboli marcati e aggiornerà i nastri in base a δ'. Se una testina dovesse spostarsi su un $\#$, allora M sposterà prima tutti gli elementi successivi a destra e poi eseguirà lo spostamento.

TM non deterministica

Una **TM non deterministica (NTM)** è una TM in cui ogni transizione può finire su più di uno stato, la funzione di transizione δ sarà quindi definita come:

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

in cui una stringa w viene accettata se almeno un ramo della computazione accetta.
Per ogni TM non deterministica N esiste una TM M equivalente.

Dimostrazione:
Dato l'albero di computazione di N e considerando la radice dell'albero come ε, possiamo identificare ogni nodo dell'albero con un indirizzo che rappresenta le scelte non deterministiche che sono state eseguite per raggiungere quel nodo. M controllerà questo albero per livello (per evitare di entrare in un ramo di lunghezza infinita) e accetterà non appena troverà un nodo che accetta la stringa in input. M usa 3 nastri:

1. Nastro con l'input w, che non viene mai cambiato
2. Nastro di lavoro, che esplora un determinato cammino dell'albero
3. Nastro che contine l'indirizzo i del nodo corrente dell'albero di computazione

Ad ogni step M simulerà un ramo di computazione dalla radice fino all'indirizzo i, usando il secondo nastro come nastro di lavoro. Nel caso questo ramo accettasse, M accetterebbe, sennò va al prossimo indirizzo.

Enumeratore

Un **enumeratore** è una TM E che produce stringhe di un linguaggio in ordine casuale, con possibili ripetizioni. Può essere considerato come un generatore di linguaggi.
Dato un linguaggio L, allora:

$$L \in \text{REC} \iff \exists \text{enumeratore} E | L(E) = L$$

Dimostrazione per doppia implicazione:

1. $L \in \text{REC} \implies \exists E | L(E) = L$:
   Definisco la TM M per cui $L(M) = L$, l'enumeratore E di tutte le stringhe $s_1, s_2, \ldots, s_k \in \Sigma^*$ deve stampare solo quelle in L.
   Faccio eseguire ad E l'algoritmo:

Algoritmo: Enumeratore simula TM

```
for i in range(0,∞) :
    for j in range(0,i) :
        accept=M(s_j,i)
        if accept :
            return s_j
```

   In cui `M(`s_j`,i)` simula la TM M con input s_j per i passi di computazione e ritorna `True` se la stringa s_j viene accettata.

2. $L \in \text{REC} \impliedby \exists E | L(E) = L$:
   Definiamo una TM M che simula E e confronta l'output con w, accettandolo quando è uguale (in un tempo infinito succederà).

3.2 Linguaggi decidibili

Codifica binaria di un oggetto

Dato un **oggetto** O, definiamo con $< O >$ la sua **codifica binaria**, cioè una stringa binaria che ne descrive le caratteristiche.

Accettazione di DFA

Il linguaggio:

$$A_{\text{DFA}} = \{< D, w > | D \in \text{DFA} \land w \in L(D)\}$$

è decidibile.

Dimostrazione:
La TM M con input $< D, w >$:

1. Interpreta la codifica di D, se è sbagliata rifiuta
2. Simula D con input w
3. Se D termina con uno stato accettante, M accetta, sennò rifiuta

Visto che i DFA terminano sempre, il linguaggio $A_{\text{DFA}} \in \text{DEC}$.
Analogamente anche i linguaggi:

$$A_{\text{NFA}} = \{< N, w > | N \in \text{DFA} \land w \in L(N)\}$$
$$A_{\text{REX}} = \{< R, w > | R \in \text{re} \land w \in L(R)\}$$

sono decidibili.

Linguaggi vuoti di DFA

Il linguaggio:

$$E_{\text{DFA}} = \{< D > | D \in \text{DFA} \land L(D) = \emptyset\}$$

è decidibile.

Dimostrazione:
La TM M con input $< D, w >$:

1. Interpreta la codifica di D, se è sbagliata rifiuta
2. Marca lo stato iniziale di D
3. Continua a marcare tutti gli stati che hanno uno transazione entrante da uno stato marcato
4. Se viene marcato uno degli stati accettanti di D, M rifiuta, sennò accetta

Essendo il numero di stati di D finito, il linguaggio $E_{\text{DFA}} \in \text{DEC}$.

Equivalenza tra DFA

Il linguaggio:

$$EQ_{\text{DFA}} = \{< D_1, D_2 > | D_1, D_2 \in \text{DFA} \land L(D_1) = L(D_2)\}$$

è decidibile.

Dimostrazione:
Considero la **differenza simmetrica** (Δ) tra $L(D_1) = L_1$ e $L(D_2) = L_2$ come:

$$L_1 \Delta L_2 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$$

cioè tutti gli elementi solo in L_1 o L_2.
Essendo le operazioni di complemento, intersezione e unione chiuse nei linguaggi regolari, ne deriva che:

$$L_1 \Delta L_2 \in \text{REG}$$

Inoltre è vero che:

$$L_1 \Delta L_2 = \emptyset \iff L_1 = L_2$$

La TM M con input $< D_1, D_2 >$:

1. Interpreta la codifica di D_1 e D_2, se è sbagliata rifiuta
2. Costruisce un DFA D per cui $L(D) = L_1 \Delta L_2$
3. Esegue gli stessi passaggi del linguaggio E_{DFA} con input D
4. Ritorna lo stesso risultato

Essendo il linguaggio $E_{\text{DFA}} \in \text{DEC}$, anche il linguaggio $EQ_{\text{DFA}} \in \text{DEC}$.

Accettazione di CFG

Il linguaggio:

$$A_{\text{CFG}} = \{< G, w > | G \in \text{CFG} \land w \in L(G)\}$$

è decidibile.

Dimostrazione:
Assumiamo G sia in CNF, per ogni stringa $w \in L(G)$ con $|w| = n$, i passi necessari per generarla sono $2n - 1$.
La TM M con input $< G, w >$:

1. Interpreta la codifica di G, se è sbagliata rifiuta
2. Controlla tutte le produzioni di lunghezza $2n - 1$, con $|w| = n$
3. Se una delle produzioni è uguale a w accetta, sennò rifiuta

Visto che il numero di produzioni di una lunghezza specificata è finita, il linguaggio $A_{\text{CFG}} \in \text{DEC}$.

Linguaggi vuoti di CFG

Il linguaggio:

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

è decidibile.

Dimostrazione:
La TM M con input $< G, w >$:

1. Interpreta la codifica di G, se è sbagliata rifiuta
2. Marca tutti i terminali di G
3. Continua a marcare tutte le variabili A per cui esiste in G una regola $A \to u_1 \dots u_k$ tale che $u_1 \dots u_k$ sono già marcati
4. Se viene marcata la variabile iniziale di G, M rifiuta, sennò accetta

Essendo il numero di variabili di G finito, il linguaggio $E_{\text{CFG}} \in \text{DEC}$.

Equivalenza tra DFA

Il linguaggio:

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

non è decidibile.

3.3 Linguaggi non decidibili

Accettazione di TM

Il linguaggio:

$$A_{\text{TM}} = \{< M, w > | M \in \text{TM} \land w \in L(M)\}$$

è riconoscibile ma non decidibile.

Dimostrazione di riconoscibilità:
Definiamo U come la TM universale, con due nastri, uno per l'input e uno con la configurazione attuale di M.
La TM U con input $< M, w >$:

1. Interpreta la codifica di M, se è sbagliata rifiuta
2. Scrive sul nastro 2 la codifica iniziale $< (q_{start}, w) >$
3. Ad ogni passo, con stringa attuale sul nastro 2 uguale a $< (a, q, b) >$:
   - 3.1 Scansiona il nastro 1 in cerca di una transizione della forma $< (q, x), (r, y, z) >$, che indica $\delta(q, x) = (r, y, z)$ con $z \in \{L, R\}$
   - 3.2 Se $x \neq b[1]$, cerca la prossima regola
   - 3.3 Se $x = b[1]$, aggiorna la configurazione sul nastro 2 secondo la transizione
   - 3.4 Se sul nastro 2 è scritto q_{accept} o q_{reject}, accetta o rifiuta di conseguenza

Quindi U accetterà solo se $w \in L(M)$, quindi $A_{\text{TM}} \in$ REC.
Dimostrazione di indecidibilità:
Supponiamo che A_{TM} sia decidibile, allora esiste una TM H decisore per cui $L(H) = A_{\text{TM}}$.

Definisco una TM D che su input $< M >$:

1. Esegue H con input $< M, < M >>$, cioè la stringa data come input ad H è la codifica della TM M stessa
2. Se H accetta, D rifiuta a viceversa

Si ha quindi che per una generica TM in input M:

$$D(< M >) = \begin{cases} \text{accetta} & M \text{ rifiuta } < M > \implies H \text{ rifiuta } < M, < M >> \\ \text{rifiuta} & M \text{ accetta } < M > \implies H \text{ accetta } < M, < M >> \end{cases}$$

Essendo H un decisore ne deriva che anche D sia un decisore.
Ma allora se l'input di D fosse D stesso:

$$D(< D >) = \begin{cases} \text{accetta} & D \text{ rifiuta } < D > \implies H \text{ rifiuta } < D, < D >> \\ \text{rifiuta} & D \text{ accetta } < D > \implies H \text{ accetta } < D, < D >> \end{cases}$$

Questo causa una contraddizione in quando D non può accettare se lui stesso rifiuta, quindi H non è decisore e $A_{\text{TM}} \notin$ DEC.

Classe dei linguaggi coTuring-riconoscibili

Dato un alfabeto Σ, la **classe dei linguaggi coTuring-riconoscibili** di Σ, scritta come coREC, è l'insieme dei linguaggi per cui il loro complemento è riconoscibile:

$$\text{coREC} = \{L \subseteq \Sigma^* | \overline{L} \in \text{REC}\}$$

Decidibilità in base al complemento

Un linguaggio L è decidibile se è sia Turing-riconoscibile che coTuring-riconoscibile.
Quindi se L non è decidibile allora uno tra L e $\overline{L}$ non è riconoscibile.
Potremmo anche dire che:

$$\text{DEC} = \text{REC} \cap \text{coREC}$$

Dimostrazione per doppia implicazione:

1. L decidibile $\implies$ $L, \overline{L}$ riconoscibili:
   Visto che L è decidibile, esiste la TM M decisore per cui $L(M) = L$, allora possiamo costruire la TM $\overline{M}$ inversa che accetta quando M rifiuta e viceversa. Per costruzione $\overline{M}$ riconosce il linguaggio $L(\overline{M}) = \overline{L}$ e quindi sia L che $\overline{L}$ sono riconoscibili.

2. L decidibile $\impliedby$ $L, \overline{L}$ riconoscibili:
   Essendo $L, \overline{L}$ riconoscibili, esistono le TM $M, \overline{M}$ per cui $L(M) = L$ e $L(\overline{M}) = \overline{L}$.
   Costruisco uno TM D che con input w:

   1. Esegue in parallelo sia M che $\overline{M}$ con input w
   2. Se M accetta, D accetta. Se $\overline{M}$ accetta, D rifiuta.

   Per costruzione D è una TM decisore per L, quindi L è decidibile.

3.3.1 Linguaggi non riconoscibili

Diagonalizzazione

La **diagonalizzazione** è un metodo di dimostrazione per dimostrare l'esistenza di una funzione biettiva tra due insiemi.

Esempi:

1. L'insieme $\mathbb{R}$ non è numerabile.
   Per dimostrarlo supponiamo che esista una biezione:

   $$f : \mathbb{N} \to [0, 1]$$

   Definiamo un numero d per cui la i-esima cifra è diversa dalla i-esima cifra di $f(i)$. Per costruzione $\nexists n | f(n) = d$, quindi la funzione f non è biettiva, dimostrando che $\mathbb{R}$ non sia numerabile.

2. L'insieme $\mathcal{B}$ di tutte le stringhe binarie di lunghezza infinita non è numerabile.
   Per dimostrarlo supponiamo che esista una biezione:

   $$f : \mathbb{N} \to \mathcal{B}$$

   Definiamo una sequenza x per cui la i-esima cifra è diversa dalla i-esima cifra di $f(i)$. Per costruzione $\nexists n | f(n) = x$, quindi la funzione f non è biettiva, dimostrando che $\mathcal{B}$ non sia numerabile.

Esistenza di linguaggi non riconoscibili

Dato un alfabeto Σ allora:

$$\exists L \subseteq \Sigma^* | L \notin \text{REC}$$

Dimostrazione:
Possiamo dire che $\mathcal{M} = \{< M > | M \in \text{TM}\} \subseteq \Sigma^*$. Definiamo la relazione "$<_l$" che genera un ordine lessicografico tra i caratteri di Σ, allora:

$$x < y \iff \begin{cases} |x| < |y| \\ |x| = |y| \wedge x <_l y \end{cases}$$

La relazione è di ordine totale, quindi posso definire una funzione biettiva:

$$f : \mathbb{N} \to \Sigma^*$$

per cui:

$$f(i) = i\text{-esima stringa di } \Sigma^* \text{ secondo l'ordine lessicografico}$$

Quindi Σ^* è numerabile e di conseguenza $\mathcal{M}$ è numerabile.
Prendiamo $\mathcal{P}(\Sigma^*)$, l'insieme dei linguaggi esistenti su Σ^*. Associo poi ad ogni linguaggio una stringa binaria X_L di lughezza infinita per cui:

$$X_L[i] = \begin{cases} 0 & s_i \notin L \\ 1 & s_i \in L \end{cases}$$

in cui s_i è la i-esima stringa di Σ^* ordinata secondo l'ordine lessicografico.
Esiste quindi una biezione:

$$f : \mathcal{L} \to \mathcal{B}$$

perciò essendo $\mathcal{B}$ non numerabile, di conseguenza $\mathcal{L}$ è non numerabile.
Essendo quindi $\mathcal{M}$ numerabile e $\mathcal{L}$ non numerabile, deve esistere almeno un linguaggio che non è riconosciuto da nessuna TM e quindi non riconoscibile.

3.4 Riduzione

Riduzione

La **riduzione** è un metodo di dimostrazione per cui dati due problemi Ae B, sapendo la soluzione di B è possibile risolvere A. Viene scritto come:

$$A \leq B$$

Terminazione di TM

Il linguaggio:

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

è indecidibile.

Dimostrazione:
Supponiamo che $HALT_{\text{TM}}$ sia decidibile e H sia la TM decisore che lo decide.
Definisco una TM S che con input $< M, w >$:

1. Esegue H con input $< M, w >$
2. Se H rifiuta, anche S rifiuta
3. Se H accetta, allora S esegue M con input w essendo sicuro che la TM termini
4. Se M accetta o rifiuta, S fa lo stesso

Essendo H decisore, anche S è una TM decisore per A_{TM}, cosa non possibile per la dimostrazione precedente.

Linguaggi vuoti di TM

Il linguaggio:

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

è indecidibile.

Dimostrazione:
Supponiamo che E_{TM} sia decidibile e R sia la TM decisore che lo decide.
Definisco una TM S che con input $< M, w >$:

1. Costruisce una TM M' che con input x:
   1.1 Se $x \neq w$, rifiuta
   1.2 Se $x = w$, esegue M con input x
   1.3 Se M accetta, anche M' accetta
2. Esegue R con input $< M' >$
3. Se R accetta, S rifiuta e viceversa

Essendo R decisore, anche S è una TM decisore per A_{TM}, cosa non possibile per la dimostrazione precedente.

3.4.1 Riduzione tramite mappatura

Funzione calcolabile

Una funzione $f : \Sigma^* \to \Sigma^*$ è **calcolabile** se esiste una TM M che la calcola, cioè che:

$$\forall w \in \Sigma^* \; M(w) \text{ termina con } f(w) \text{ nel nastro}$$

Riduzione tramite mappatura

Un linguaggio A è riducibile ad un'altro linguaggio B, scritto $A \leq_m B$, se:

$$\exists f : \Sigma^* \to \Sigma^* | \forall w \in \Sigma^* \; w \in A \iff f(w) \in B$$

La funzione f deve essere calcolabile.

Decidibilità tramite riduzione

Dati due linguaggi A, B per cui $A \leq_m B$, allora:

$$B \in \text{DEC} \implies A \in \text{DEC}$$

Da questo possiamo dire anche che:

$$A \notin \text{DEC} \implies B \notin \text{DEC}$$

Dimostrazione:
Se $B \in$ DEC esiste una TM decisore M_B tale che $L(M_B) = B$.
Definiamo una TM M_A che con input $w \in \Sigma^*$:

1. Calcola $f(w)$
2. Esegue M_B con input $f(w)$
3. Se M_B accetta, anche M_A accetta, sennò rifiuta

M_A è quindi una TM decisore per il linguaggio A.
Esempi:

1. Il linguaggio:
$$HALT_{\text{TM}} = \{< M, w > | M \in \text{TM} \wedge M(w) \text{ termina}\}$$
è indecidibile.
Per dimostarlo lo riduciamo al linguaggio A_{TM}, trovando una funzione:
$$f : \Sigma^* \to \Sigma^* | < M, w > \in A_{\text{TM}} \iff f(< M, w >) \in HALT_{\text{TM}}$$

La funzione f sarà calcolata da una TM che su input $< M, w >$:

1. Costruisce la TM M' che con input x:
   1.1 Esegue $M(x)$
   1.2 Se M accetta, M' accetta
   1.3 Se M rifiuta, M' va in loop
2. Restituisce in output $< M', w >$

Dimostrazione di correttezza:

1. $< M, w > \in A_{\text{TM}} \implies < M', w > \in HALT_{\text{TM}}$:
   Se $< M, w > \in A_{\text{TM}}$ allora M accetta w, quindi per costruzione M' termina e $< M', w > \in HALT_{\text{TM}}$
2. $< M, w > \notin A_{\text{TM}} \implies < M', w > \notin HALT_{\text{TM}}$:
   Se $< M, w > \notin A_{\text{TM}}$ allora M rifiuta w o va in loop, quindi per costruzione M' va in loop e $< M', w > \notin HALT_{\text{TM}}$

Da questo segue che $A_{\text{TM}} \leq_m HALT_{\text{TM}}$, ma poiché $A_{\text{TM}} \notin \text{DEC} \implies HALT_{\text{TM}} \notin \text{DEC}$.

2. Il linguaggio:

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

è indecidibile.
Per dimostarlo lo riduciamo al linguaggio E_{TM}, trovando una funzione:

$$f : \Sigma^* \to \Sigma^* | < M > \in E_{\text{TM}} \iff f(< M_1, M_2 >) \in EQ_{\text{TM}}$$

La funzione f sarà calcolata da una TM che su input $< M >$:

1. Costruisce la TM M' che con input x:
   1.1 Rifiuta sempre
2. Restituisce in output $< M, M' >$

Dimostrazione di correttezza:

1. $< M > \in E_{\text{TM}} \implies < M, M' > \in EQ_{\text{TM}}$:
   Se $< M > \in E_{\text{TM}}$ allora $L(M) = \emptyset$, per costruzione $L(M') = \emptyset$ e quindi $< M, M' > \in EQ_{\text{TM}}$
2. $< M > \notin E_{\text{TM}} \implies < M, M' > \notin EQ_{\text{TM}}$:
   Se $< M > \notin E_{\text{TM}}$ allora $L(M) \neq \emptyset$, per costruzione $L(M') = \emptyset$ e quindi $< M, M' > \notin EQ_{\text{TM}}$

Da questo segue che $E_{\text{TM}} \leq_m EQ_{\text{TM}}$, ma poiché $E_{\text{TM}} \notin \text{DEC} \implies EQ_{\text{TM}} \notin \text{DEC}$.

Riconoscibilità tramite riduzione

Dati due linguaggi A, B per cui $A \leq_m B$, allora:

$$B \in \text{REC} \implies A \in \text{REC}$$

Da questo possiamo dire anche che:

$$A \notin \text{REC} \implies B \notin \text{REC}$$

Riduzione dei complementi

Dati due linguaggi A, B, si ha che:

$$A \leq_m B \iff \overline{A} \leq_m \overline{B}$$

Dimostrazione:
Presa la funzione f per cui $A \leq_m B$, si ha che:

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

Esempio:
Il linguaggio:

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

non è riconoscibile.
Per dimostarlo mostriamo che $A_{\text{TM}} \leq_m \overline{EQ_{\text{TM}}}$, trovando una funzione:

$$f : \Sigma^* \to \Sigma^* | < M, w > \in A_{\text{TM}} \iff f(< M_1, M_2 >) \in \overline{EQ_{\text{TM}}}$$

La funzione f sarà calcolata da una TM che su input $< M, w >$:

1. Costruisce le TM M_1, M_2 che con input x:
   1.1 M_1 rifiuta sempre
   1.2 M_2 esegue $M(w)$ e accetta se M accetta
2. Restituisce in output $< M_1, M_2 >$

Dimostrazione di correttezza:

1. $< M, w > \in A_{\text{TM}} \implies < M_1, M_2 > \in \overline{EQ_{\text{TM}}}$:
   Se $< M, w > \in A_{\text{TM}}$ allora M accetta w, quindi $L(M_2) = \Sigma^*$ e essendo per costruzione $L(M_1) = \emptyset$ si ha che $< M_1, M_2 > \in \overline{EQ_{\text{TM}}}$
2. $< M, w > \notin A_{\text{TM}} \implies < M_1, M_2 > \notin \overline{EQ_{\text{TM}}}$:
   Se $<, w > \notin A_{\text{TM}}$ allora M rifiuta w o va in loop, quindi $L(M_2) = \emptyset$ e essendo per costruzione $L(M_1) = \emptyset$ si ha che $< M_1, M_2 > \notin \overline{EQ_{\text{TM}}}$

Da questo segue che $A_{\text{TM}} \leq_m \overline{EQ_{\text{TM}}}$ e quindi anche $\overline{A_{\text{TM}}} \leq_m EQ_{\text{TM}}$, ma poiché $\overline{A_{\text{TM}}} \notin \text{REC} \implies EQ_{\text{TM}} \notin \text{REC}$.

3.5 Teoremi di incompletezza di Gödel

Rappresentazione di dimostrazioni

Dato un sistema Π, basato su assiomi, per ogni affermazione o dimostrazione, esiste una rappresentazione tramite una stringa di lunghezza finita.

Esiste quindi una TM decisore V per cui $V(< x, w >)$ accetta se w è una dimostrazione dell'affermazione x.

Affermazione dimostrabile

Data un'affermazione x, questa è **dimostrabile** se esiste una stringa w per cui $V(< x, w >)$ accetta.
L'affermazione x viene detta **indipendente** se né x, né $\overline{x}$ sono dimostrabili.

Un sistema Π può essere:

- **Consistente**: se $\forall x$ al massimo uno tra x e $\overline{x}$ è dimostrabile
- **Valido**: se $\forall x$, se x è dimostrabile allora è vero
- **Completo**: se $\forall x$ almeno uno tra x e $\overline{x}$ è dimostrabile

Da questo possiamo derivare che:

- Π valido $\implies$ Π consistente
- Π completo e consistente $\implies$ $\forall x$ esattamente uno tra x e $\overline{x}$ è dimostrabile
- Π completo e valido $\implies$ tutto e solo ciò che è vero è dimostrabile

Validità e completezza dei sistemi

Se un sistema Π è valido, allora non è completo.

Dimostrazione tramite TM:
Consideriamo la TM P, che su input $< x >$:

1. Per ogni $k \in [1, \infty)$
2. Per ogni w con $|w| = k$
3. Se $V(< x, w >)$ accetta, ritorna w

Questa TM decide il linguaggio $L_{\text{provable}} = \{< x > | x \text{ è dimostrabile}\}$

Consideriamo anche la TM R, che su input $< x >$:

1. Per ogni $k \in [1, \infty)$
2. Per ogni w con $|w| = k$
3. Se $V(< x, w >)$ accetta, R accetta
4. Se $V(< \overline{x}, w >)$ accetta, R rifiuta

Se π fosse valido e completo R sarebbe un decisore per cui $\forall x$ vera, allora $R(< x >)$ accetta. Partendo da R creiamo una TM D che su input $< M, w >$:

1. Ritorna $R(<$ "$M(w)$ termina" $>)$

La TM D è un decisore per $HALT_{\text{TM}} = \{< M, w > | M(w) \text{ termina}\}$, cosa però impossibile perchè $HALT_{\text{TM}}$ è indecidibile, quindi Π non può essere valido e completo.

Primo teorema di incompletezza

Dato un sistema Π e l'implicazione:

$$\Pi \text{ valido} \implies \Pi \text{ consistente}$$

Possiamo estendere l'affermazione precedente dicendo anche che:
Se un sistema Π è consistente, allora non è completo.

Dimostrazione tramite TM:
Se Π fosse consistente, allora usando il lemma:

- Esistono affermazioni di cui si può essere convinti che siano vere, controllando con "brute force" l'affermazione.

Come:

- Per una TM con una traccia di esecuzione con $t \geq 1$ passi, esiste una prova dell'affermazione in Π. Per dimostrarlo che è vero basta controllare l'esecuzione della TM.

Quindi se M è una TM e $M(w)$ termina, allora esiste in Π una prova dell'affermazione: "$M(w)$ termina". Considerando la TM D, che su input $< M >$:

1. Per ogni $k \in [1, \infty)$
2. Per ogni w con $|w| = k$
3. Se w è una prova per l'affermazione "$M(w)$ termina", D va in loop
4. Se w è una prova per l'affermazione "$M(w)$ va in loop", D termina

Siccome Π è consistente e valido, al massimo una tra le affermazioni "$D(< D >)$ termina" e "$D(< D >)$ va in loop" è dimostrabile e vera:

- Suppongo che l'affermazione "$D(< D >)$ va in loop" sia dimostrabile, allora $D(< D >)$ trova la dimostrazione e termina. Ma $D(< D >)$ terminando implica che esista anche una dimostrazione per l'affermazione "$D(< D >)$ termina".
- Suppongo che l'affermazione "$D(< D >)$ termina" sia dimostrabile, allora $D(< D >)$ trova la dimostrazione e va in loop. Ma $D(< D >)$ andando in loop implica abbia una traccia di esecuzione e quindi esista anche una dimostrazione per l'affermazione "$D(< D >)$ va in loop".

Quindi entrambe le affermazioni sono dimostrabili e quindi Π non può essere consistente e completo.

Secondo teorema di incompletezza

Dato un sitema Π, l'affermazione:

$$\Pi \text{ è consistente}$$

non è dimostrabile in Π stesso.

Dimostrazione:
Usiamo la stessa TM D definita precedentemente.
La TM non trova dimostrazione né per l'affermazione "$D(< D >)$ termina" né per "$D(< D >)$ va in loop", allora $D(< D >)$ va in loop. Quindi l'affermazione "$D(< D >)$ va in loop" è codificabile e questo implica che:

$$\Pi \text{ consistente} \implies D(< D >) \text{ va in loop}$$

L'unico modo per evitare una contraddizione è che l'affermazione "Π è consistente" non sia dimostrabile in Π.

4

Complessità

Oltre che concentrarsi sul sapere se un problema sia risolvibile o no, si vogliono anche mettere in gioco le risorse, in particolare:

- Tempo, considerato come numero di passi
- Spazio, considerato come la memoria necessaria

4.1 Complessità temporale

Tempo di esecuzione di una TM

Data una TM D decisore, il **tempo di esecuzione** di D è una funzione $f : \mathbb{N} \to \mathbb{N}$ che con input x per cui $|x| = n$:

$$t(n) = \max_{x \in \Sigma^*}(\text{n° passi di } M(x))$$

Tempo di una TM multinastro

Data una TM multinastro M con tempo $t(n)$ esiste una TM M' che la simula con tempo $t^2(n)$

Dimostrazione:
Consideriamo la TM M che simula una TM multinastro, come visto nella dimostrazione di equivalenza.
La TM M ha bisogno di $k \cdot O(n) = O(n)$ per preparare il nastro, poi ogni passo ha costo $k \cdot O(t(n)) = O(t(n))$, quindi facendo la TM multinastro $t(n)$ passi, il tempo di M è $O(n) + O(t(n)) \cdot O(t(n)) = O(t^2(n))$

4.2 Classe dei linguaggi P

Classe DTIME

Data la funzione $t : \mathbb{N} \to \mathbb{N}$, definiamo la classe:

$$\text{DTIME}(t(n)) = \{L \in \text{DEC} \,|\, L \text{ decidibile in tempo } O(t(n))\}$$

Per questa definizione vengono contate solo le TM a singolo nastro.

Classe P

Definiamo la classe:

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

cioè l'insieme di tutti i linguaggi decidibili in tempo polinomiale.

Esempi:

1. $PATH = \{< G, s, t > | G$ è un grafo ed esiste un cammino $s \to t\}$
   Definita una TM M che su input $< G, s, t >$:

   1. Marca s
   2. Per ogni arco (u, v):
      2.1 Se u è marcato e v no marca anche v
   3. Quando non ci sono più vertici marcabili termina
   4. Se t è marcato accetta

   Il costo di questa TM è $O(nm) = O(n^3)$, quindi $PATH \in \mathrm{P}$

2. $2COL = \{< G > | G$ è 2 colorabile$\}$ in cui 2 colorabile indica che si possono colorare i vertici con 2 colori in modo tale che ogni arco colleghi nodi di colore diverso (per esempio verranno usati il rosso e il blu).
   Definita una TM M che su input $< G >$:

   1. Colora un nodo a caso di rosso
   2. Per ogni arco (u, v):
      2.1 Se u è colorato di rosso e v non è colorato, colora v di blu
      2.2 Se u è colorato di blu e v non è colorato, colora v di rosso
      2.3 Se u e v sono colorati dello stesso colore rifiuta
   3. Quando non ci sono più vertici colorabili termina
   4. Se il grafo è completamente colorato accetta

   Il costo di questa TM è polinomiale, quindi $2COL \in \mathrm{P}$

Classe EXP

Definiamo la classe:

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

cioè l'insieme di tutti i linguaggi decidibili in tempo esponenziale.

Esempio:

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

Tutte le possibili combinazioni di colori di G sono 3^n, quindi possiamo controllarle tutte in tempo $O(n \cdot 3^n)$. Questo implica che $3COL \in \mathrm{EXP}$, ma non sappiamo se $3COL \in \mathrm{P}$ perché il miglior algoritmo conosciuto ci dice che $3COL \in \mathrm{DTIME}((1.\overline{3})^n)$.

4.2.1 2CNF

Conjunctive Normal Form (CNF)

Una **Conjunctive Normal Form (CNF)** è una formula logica formata da un "$\wedge$" tra n clausole, ognuna formata da un "$\vee$" tra m letterali, cioè variabili o variabili negate.

Definiamo il linguaggio:

$$CNFSAT = \{< \phi > | \phi \in \text{CNF} \wedge \exists x \in \{0,1\}^n \ \phi(x) = true\}$$

che comprende tutte le formule in CNF soddisfacibili, $CNFSAT \in$ EXP, ma non sappiamo se $CNFSAT \in P$.

Un qualsiasi insieme del tipo $kCNF$ indica una CNF in cui ogni clausola è formata al massimo da k letterali.
Esempi:

1. Formula in 2CNF:
$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_3} \vee x_4)$$
Inoltre è soddisfacibile dall'assegnamento:
$$x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1$$

2. Formula in 3CNF:
$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$
Inoltre è soddisfacibile dall'assegnamento:
$$x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 1, x_5 = 1, x_6 = 0$$

Dagli insiemi precedenti è possibile anche definire gli insiemi:

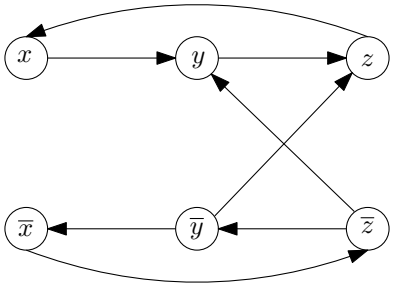
$$kSAT = \{< \phi > | \phi \in k\,\text{CNF} \wedge \exists x \in \{0,1\}^n \ \phi(x) = true\}$$

che contengono tutte le formule in kCNF soddisfacibili.

2SAT polinomialmente decidibile

Il linguaggio $2SAT \in P$

Dimostrazione:
Sia $\phi(x_2, \ldots, x_n)$ la formula con $\phi \in 2SAT$. Associamo alla formula un grafo in cui i vertici sono $\{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\}$ e per ogni clausola di ϕ del tipo $(a \vee b)$ aggiungo gli archi $(\overline{a}, b)$ e $(\overline{b}, a)$. Per esempio la formula:
$$(\overline{x} \vee y) \wedge (\overline{y} \vee z) \wedge (x \vee \overline{z}) \wedge (y \vee z)$$
avrebbe associato il grafo:



Soddisfacibilità di ϕ in base al grafo associato

Data una formula $\phi \in 2CNF$ e il grafo associato G, allora si ha che:

$$\phi \text{ soddisfacibile} \iff$$
$$\forall x\ G \text{ non contiene una componente fortemente connessa con sia } x \text{ che } \overline{x}$$

Dimostrazione:

1. ϕ soddisfacibile $\implies \forall x\ G$ non contiene una componente fortemente connessa con sia x che $\overline{x}$:
   Sia ϕ soddisfacibile, allora per un arco (a, b) si ha che $a = 1 \implies b = 1$, questo perché se esiste l'arco (a, b) allora esiste la clausola $(\overline{a} \vee b)$, che se $a = 1$ è vera solo se $b = 1$.
   Questo implica che per ogni x, se esiste un cammino $x \to \overline{x}$, allora $x = 0$ perché se $x = 1$ ci sarebbe un cammino $x \to \cdots \to \overline{x}$ con sia $\overline{x} = 1$ che $x = 1$. Inoltre se esiste un cammino $\overline{x} \to x$, allora $\overline{x} = 0$ $(x = 1)$, per lo stesso principio.
   Quindi se esistono entrambi i cammini, che implica che x e $\overline{x}$ sono nello stesso componente fortemente connesso, la formula non è soddisfacibile.

2. ϕ soddisfacibile $\impliedby \forall x\ G$ non contiene una componente fortemente connessa con sia x che $\overline{x}$:
   Supponiamo che G non abbia componenti fortemente connessi che contengono sia x che $\overline{x}$ per un qualsiasi x. Ordiniamo topologicamente le componenti e poniamo $x = 1$ se x appare dopo $\overline{x}$.

   Ordinamento topologico dei componenti

   Data una formula $\phi \in 2CNF$ e il grafo associato G con componenti fortemente connesse ordinate topologicamente, allora si ha che per nessun arco $(a, b) \in G$ ha $a = 1$ e $b = 0$.

   Questo implica che ϕ è soddisfacibile perché se ci fosse una clausola $(x \vee y)$ con $x = 0$ e $y = 0$ avremo un arco $\overline{x} \to y$ con $\overline{x} = 1$ e $y = 0$.
   Dimostrazione:
   Supponiamo che non sia vero, prendendo un arco (a, b) con $a = 1$ e $b = 0$ con $a, b \in C_i$ (componente generico), questo arco implica la clausola $(\overline{a} \vee b)$ e anche un altro arco $(\overline{b}, \overline{a})$. Siccome $a = 1$ allora $\overline{a}$ deve apparire in un componente C_j per cui $j < i$, inoltre siccome $b = 0$ allora $\overline{b}$ deve apparire in un componente C_k con $k > i$. Quindi l'arco $(\overline{b}, \overline{a})$ dovrebbe avere $\overline{a}$ in un componente prima di a, b e $\overline{b}$ in un componente dopo.

Questo dimostra che non esistono archi (a, b) con $a = 1$ e $b = 0$ e quindi nessuna componente formente connessa contiene sia x che $\overline{x}$ per un qualsiasi x e ciò dimostra che $2SAT \in P$, perché si può creare e controllare il grafo associato in tempo polinomiale.

4.3 Classe dei linguaggi NP

Classe NTIME

Data la funzione $t : \mathbb{N} \to \mathbb{N}$, definiamo la classe:

$$\text{NTIME}(t(n)) = \{L \in \text{DEC} \,|\, L \text{ decidibile non deterministicamente in tempo } O(t(n))\}$$

Classe NP

Definiamo la classe:

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

cioè l'insieme di tutti i linguaggi decidibili non deterministicamente in tempo polinomiale.

Esempio:
Dato il linguaggio:

$$SAT = \{< \phi > \,|\exists x \in \{0,1\}^n \; \phi(x) = true\}$$

Se creo una NTM N che su input $< \phi >$:

1. Inizializza $x[1, \ldots, n]$
2. Per ogni elemento con indice i crea 2 rami di computazione:
   - 2.1 Uno in cui $x[i] = 0$
   - 2.2 Uno in cui $x[i] = 1$
3. Alla fine se almeno una delle foglie dell'albero binario accetta, N accetta.

Esistendo la NTM N con tempo polinomiale, allora $SAT \in$ NP.

TM verificatore

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

- V prende in input $< x, y >$
- $\forall x \; (x \in L \iff \exists y | V(< x, y >) \text{ accetta})$

V ha tempo polinomiale se viene eseguito in $O(|x|^k)$. La stringa y viene detta **certificato di appartenenza**.

In pratica V capisce se y è un corretto ramo di computazione per la stringa x sulla TM N per cui $L(N) = L$. Tramite i verificatori si può creare una seconda definizione di NP.

Classe NP definita con i verificatori

Data la classe NP:

$$\text{NP} = \{L \in \text{DEC} \,|\exists V \text{ verificatore per } L\}$$

cioè l'insieme di tutti i linguaggi verificabili in tempo polinomiale.

Esempio:
Dato il linguaggio:

$$3COL = \{G|G \text{ è 3 colorabile}\}$$

Creiamo il verificatore V che su input $< G, y >$:

1. Interpreta y come una colorazione $(C_1, \ldots, C_n)$ con $C_i = \{R, G, B\}$
2. Per ogni arco del grafo (i, j):
   2.1 Rifiuta se $C_i = C_j$
3. Accetta alla fine

V ha tempo polinomiale rispetto ad $n = |V(G)|$, quindi $3COL \in \text{NP}$.

Equivalenza delle definizioni di NP

La due definizioni di NP date precedentemente sono equivalenti, cioè:

$$L \text{ ha un verificatore polinomiale} \iff$$
$$\exists\, \text{NTM}\, N | N \text{ decide } L \text{ in tempo polinomiale}$$

Dimostrazione:

1. L ha un verificatore polinomiale $\implies \exists\, \text{NTM}\, N|N$ decide L in tempo polinomiale:
   Creo una NTM N che prova tutti i certificati y e accetta se $V(x, y)$ accetta. Per costruzione $L(N) = L$ e N ha tempo polinomiale perché $|y|$ è polinomiale rispetto a $|x|$ e V ha tempo polinomiale.
2. L ha un verificatore polinomiale $\impliedby \exists\, \text{NTM}\, N|N$ decide L in tempo polinomiale:
   Se N accetta un input x allora il certificato y esiste e determina l'insieme delle scelte non deterministiche da eseguire nella compilazione di $N(x)$. Quindi V può usare y per decidere le scelta da fare mentre simula $N(x)$ deterministicamente. V avrà tempo polinomiale perché N ha tempo polinomiale.

Classe NEXP

Definiamo la classe:

$$\text{NEXP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(2^{n^k})$$

cioè l'insieme di tutti i linguaggi decidibili in tempo esponenziale non deterministicamente.

Gerarchia delle classi

Date le classi definite precedentemente, si ha che:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP}$$

Dimostrazione:
Sia $L \in \mathrm{P}$, allora esiste una TM M per cui $L(M) = L$ che ha tempo polinomiale.
Allora:

1. Creiamo $V(x, y)$ che accetta se $M(x)$ accetta. V ha tempo polinomiale, quindi $L \in \mathrm{NP}$.
2. Creiamo inoltre la TM M' che prova tutti i certificati possibili y con lunghezza polinomiale rispetto a $|x|$ e accetta se $V(x, y)$ accetta. M' ha tempo esponenziale quindi $L \in \mathrm{EXP}$.
3. Per lo stesso principio $L \in \mathrm{NEXP}$.

4.4 Riduzione in tempo polinomiale

Riduzione tramite mappatura in tempo polinomiale

Un linguaggio A è riducibile ad un'altro linguaggio in tempo polinomiale B, scritto $A \leq_m^P B$, se:

$$\exists f : \Sigma^* \to \Sigma^* | \forall w \in \Sigma^* \; w \in A \iff f(w) \in B$$

La funzione f deve avere tempo polinomiale.
La relazione $A \leq_m^P$ è transitiva.

Tempistiche tramite riduzione

Dati due linguaggi A, B per cui $A \leq_m^P B$, allora:

$$B \in \mathrm{P} \implies A \in \mathrm{P}$$

Inoltre:

$$B \in \mathrm{NP} \implies A \in \mathrm{NP}$$

Dimostrazione:

1. $B \in \mathrm{P} \implies A \in \mathrm{P}$:
   Data la TM M_B per cui $L(M_B) = B$ in tempo polinomiale, tramite la riduzione R si ha che $M_B(R(x))$ decide A in tempo polinomiale.
2. $B \in \mathrm{NP} \implies A \in \mathrm{NP}$:
   Data la NTM N_B per cui $L(N_B) = B$ in tempo polinomiale, tramite la riduzione R si ha che $N_B(R(x))$ decide A non deterministicamente in tempo polinomiale.

Esempio:

$$SAT \in \mathrm{P} \implies 4COL \in \mathrm{P}$$

Data una TM M con $L(M) = SAT$, costruisco una TM M' che trasforma l'affermazione "G è 4 colorabile" in "ϕ_G è soddisfacibile". La formula ϕ_G avrà $2n$ variabili del tipo $\{x_1, x'_1, \ldots, x_n, x'_n\}$

in cui ogni coppia (x_i, x'_i) rappresenta il colore dell'i-esimo nodo (2 variabili = 4 colori).
Per ogni arco (i, j) adeve essere vero che $(x_i, x'_i) \neq (x_j, x'_j)$, che logicamente diventa:

$$\phi_G = \bigwedge_{(i,j)\in E(G)} \overline{((x_i \wedge x_j) \vee (\overline{x_i} \wedge \overline{x_j}) \wedge (x'_i \wedge x'_j) \vee (\overline{x'_i} \wedge \overline{x'_j}))}$$

Abbiamo quindi ridotto $4COL$ a SAT polinomialmente, quindi è vero che $SAT \in \text{P} \implies 4COL \in \text{P}$.

4.5 NP-Completezza

Linguaggio NP-Completo

Un linguaggio A si dice **NP-difficile** se:

$$\forall L \in \text{NP} \;\; L \leq_m^P A$$

Se inoltre $A \in \text{NP}$, allora A è **NP-Completo**.

Teorema di Cook-Levin

Dato il linguaggo SAT, si ha che:

$$SAT \in \text{NP-Completo}$$

Dimostrazione:
Sappiamo già che $SAT \in \text{NP}$, dimostrato precedentemente.
Dato un linguaggio $A \in \text{NP}$, sia N la NTM per cui $L(N) = A$ in tempo polinomiale.
Definiamo una tabella di computazione di N con input $x = w_1 \dots w_n$, di grandezza $n^k \times n^k$, in cui ogni riga i rappresenta l'i-esima configurazione di $N(x)$ (essendo N non deterministica esiste una tabella per ogni ramo di computazione):

#	q_0	w_1	w_2	...	w_n	⊔	...	⊔	#
#				...					#
#				...					#
				⋮					
#				...					#

Una tabella è accettante se al suo interno c'è una riga contenente q_{accept} e tutte le configurazioni sono valide, cioè possibili secondo al funzione δ di N.
Data una tabella T definiamo un insieme di variabili:

$$x_{i,j,s} = \begin{cases} 1 & T[i,j] = s \\ 0 & \text{altrimenti} \end{cases}$$

In cui $i, j \in [1, n^k]$ rappresentano la riga e la colonna della tabella e $s \in S = Q \cup \Gamma \cup \{\#\}$ rappresenta un simbolo che può essere nella tabella. Da questa tabella possiamo creare una formula ϕ:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{acc}}$$

In cui:

- ϕ_{start} descrive la prima confugurazione di $N(x)$:

$$\phi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \left(\bigwedge_{k \in [1,n]} x_{1,k+2,w_k} \right) \wedge \left(\bigwedge_{k \in [n+3,n^k-1]} x_{1,k,\sqcup} \right) \wedge x_{1,n^k,\#}$$

- ϕ_{acc} è vera se nella tabella c'è una cella contenente q_{accept}:

$$\phi_{\text{acc}} = \bigvee_{i,j \in [1,n^k]} x_{i,j,q_{accept}}$$

- ϕ_{cell} controlla che le celle abbiano solo un simbolo, in particolare l'or controlla che ogni cella abbia almeno un simbolo e l'and che ogni cella non abbia 2 simboli:

$$\phi_{\text{cell}} = \bigwedge_{i,j \in [1,n^k]} \left(\left(\bigvee_{s \in S} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in S \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right)$$

- ϕ_{move} controlla che tutte le configurazioni siano valide, per farlo definiamo una finestra nella tabella, cioè un insieme di celle 2×3, per esempio:

#	q_0	w_1	w_2	...	w_n	$\sqcup$	...	$\sqcup$	#
#				...					#
#				...					#
	a_1	a_2	a_3						
	a_4	a_5	a_6						
#				...					#

  Una finestra è lecita se le celle seguono la funzione δ, cioè se è possibile che le celle nella $i+1$-esima configurazione abbiano quei simboli partendo dalla i-esima, ad esempio se $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$, entrambe le seguenti finestre sono lecite:

a	q_1	b
q_2	a	c

a	q_1	b
a	a	q_2

  Quindi la formula ϕ_{move} che controlla che tutte le configurazioni siano valide controllerà che tutte le finestre siano lecite:

$$\phi_{\text{move}} = \bigwedge_{i,j \in [1,n^k]} \left(\bigvee_{\substack{(a_1,\ldots,a_6) \\ \text{è una finestra lecita}}} x_{i,j,a_1} \wedge \cdots \wedge x_{i,j,a_6} \right)$$

La funzione ϕ sarà vera solo se la tabella è una tabella accettante per $N(x)$ e quindi:

$$\phi \text{ soddisfacibile} \iff \exists \text{tabella accettante per } N(x)$$

Inoltre le variabili di una tabella sono $n^{2k} \cdot |S| = O(n^k)$, quindi la riduzione di A in SAT è eseguibile in tempo polinomiale e questo implica che:

$$A \in \text{NP} \implies A \leq_m^P SAT \implies SAT \in \text{NP-Completo}$$

Linguaggio NP-Completo in P

Dato un linguaggio $S \in$ NP-Completo, allora:

$$S \in \text{P} \iff \text{P} = \text{NP}$$

Dimostrazione:

1. $S \in \text{P} \implies \text{P} = \text{NP}$:
   Se $S \in$ NP-Completo allora $\forall L \in \text{NP}\ \ L \leq_m^P S$, quindi se $S \in$ P allora anche $L \in$ P. Questo è valido per ogni L, quindi P = NP.

2. $S \in \text{P} \impliedby \text{P} = \text{NP}$:
   Se P = NP allora $S \in \text{NP} = \text{P}$.

Equivalenza polinomiale e esponenziale

Date le classi di linguaggi si ha che:

$$\text{P} = \text{NP} \implies \text{EXP} = \text{NEXP}$$

Dimostrazione:
Dato un linguaggio $L \in$ NEXP, allora esiste una NTM N per cui $L(N) = L$ in tempo esponenziale ($O(2^{n^k})$). Considero un linguaggio derivato da L:

$$L' = \{< x, 1^{2^{|x|^k}} > | x \in L\}$$

dove 1 è un carattere per cui $1 \notin \Sigma$.
La cardinalità dell'input di L' è 2^{n^k}, creando la NTM N' che su input $< x, 1^{2^{|x|^k}} >$:

1. Controlla che l'input sia nella forma giusta in tempo $O(|x'|) = O(2^{n^k})$, cioè in tempo polinomiale

2. Lancia $N(x)$

3. Accetta o rifiuta di conseguenza

Questa NTM decide L' in tempo polinomiale rispetto all'input, quindi $L' \in \text{NP} = \text{P}$. Esiste quindi anche una TM M' che decide L' in tempo polinomiale.
Creiamo un'altra TM M che su input x:

1. Crea $x' =< x, 1^{2^{|x|^k}} >$ in tempo esponenziale

2. Lancia $M'(x')$ che richiede tempo polinomiale

3. Accetta o rifiuta di conseguenza

Questa TM ha tempo esponenziale rispetto all'input x, quindi $L \in \text{EXP}$. Essendo questo vero per ogni linguaggio L, si ha che:

$$\text{EXP} = \text{NEXP}$$

Teorema di Ladner

Date le classi di linguaggi si ha che:

$$\text{P} \neq \text{NP} \implies \exists L \in \text{NP} \,|\, L \notin \text{NP-Completo}$$

4.6 coNP e coP

Classi di linguaggi complementari

Possiamo definire le classi di linguaggi:

$$\text{coP} = \{L | \overline{L} \in \text{P}\}$$
$$\text{coNP} = \{L | \overline{L} \in \text{NP}\}$$
$$\text{coEXP} = \{L | \overline{L} \in \text{EXP}\}$$

Chiusura del complemento in P

La classe P è chiusa rispetto al complemento, cioè:

$$\text{P} = \text{coP}$$

P $\subseteq$ coNP $\subseteq$ EXP

Date le classi dei linguaggi, si ha che:

$$\text{P} \subseteq \text{coNP} \subseteq \text{EXP}$$

Dimostrazione:
Se $L \in \text{P}$, allora $\overline{L} \in \text{P} \subseteq \text{NP}$ e quindi $L \in \text{coNP}$.
Se $L \in \text{coNP}$, allora $\overline{L} \in \text{NP} \subseteq \text{EXP}$ e quindi $L \in \text{coEXP}$.
Essendo vero per ogni L si ha che EXP = coEXP.

coNP-Completezza

Un linguaggio L è **coNP-Completo** se:

- $L \in \text{coNP}$
- $\forall A \in \text{NP} \;\; A \leq_m^P L$ (coNP-Hard)

4.7 Complessità di spazio

Complessità spaziale di una TM

La complessità spaziale di una TM è una funzione:

$$s : \mathbb{N} \to \mathbb{N}$$

per cui:

$$s(n) = \max_{|x|=n} (\text{n° celle di memoria necessaria per } M(x))$$

La differenza principale tra la complessità spaziale e temporale è il fatto che lo spazio si può riutilizzare quindi potremmo considerare la complessità spaziale come il numero massimo di celle utilizzate contemporaneamente durante la computazione di $M(x)$. Nella complessità spaziale non viene considerato il nastro di input, che è read-only.

Classe DSPACE

Definiamo la classe dei linguaggi:

$$\text{DSPACE}(s(n)) = \{L \in \text{DEC} \,|\, L \text{ decidibile con spazio } O(s(n))\}$$

Vengono consierate solo le TM deterministiche.

Classe L

Definiamo la classe:

$$\text{L} = \bigcup_{k \in \mathbb{N}} \text{DSPACE}(\log n)$$

cioè l'insieme di tutti i linguaggi decidibili con spazio logaritmico.

Esempi:

1. Dato il linguaggio:
$$A = \{0^n 1^n | n \in \mathbb{N}\}$$
In questo caso posso utilizzare un contatore (che richiede $\log n$ celle per poter contare fino ad n) per contare gli 0 e poi lo decremento quando conto gli 1. Usando spazio $\log n$ allora $A \in \text{L}$.

2. Dato il linguaggio:
$$A = \{ww^R | w \in \{0,1\}^*\}$$
Data una TM M che su input x:
   1. Determina $n = |x|$ (spazio $O(\log n)$)
   2. Per ogni indice $i \in \left[1, \left\lceil \frac{n}{2} \right\rceil\right]$ (spazio $O(\log n)$)
      2.1 Rifiuta se $x_i \neq x_{n-i+1}$ (spazio $O(\log n)$)
   3. Accetta

Il costo spaziale di questa TM è $O(\log n) + O(\log n) + O(\log n)$ perché i costi di spazio non si moltiplicano nel caso di cicli in cui le iterazioni non dipendono una dall'altra o non si vuole salvare ogni iterazione. Quindi $A \in \mathrm{L}$.

Classe PSPACE

Definiamo la classe:

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

cioè l'insieme di tutti i linguaggi decidibili con spazio polinomiale.

Classe EXPSPACE

Definiamo la classe:

$$\mathrm{EXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{DSPACE}(2^{n^k})$$

cioè l'insieme di tutti i linguaggi decidibili con tempo esponenziale.

$PATH$ decidibile con $\log^2 n$ spazio

Dato il linguaggio:

$$PATH = \{< G, s, t > | G \text{ è un grafo ed esiste un cammino } s \to t\}$$

si ha che:

$$PATH \in \mathrm{DSPACE}(\log^2 n)$$

Dimostrazione:
Definiamo la seguente funzione ricorsiva $PATH?(x, y, k)$:

1. Se $k = 0$:
   1.1 Se $(x, y) \in E$ o $x = y$ accetta, sennò rifiuta
2. Se $k > 0$ per ogni nodo $v \in V$:
   2.1 Esegue $PATH?(x, v, k - 1)$
   2.2 Esegue $PATH?(v, y, k - 1)$
   2.3 Se entrambe accettano, accetta, sennò rifiuta

Una TM M che esegue $PATH?(x, y, \lceil \log n \rceil)$ accetterà solo se in G esiste un cammino $x \to y$ che passa per massimo n nodi. La procedura ha bisogno di salvare i nodi in input, che richiede $O(\log n)$ spazio e poi ha un albero di ricorsione di altezza $\log n$, quindi il costo spaziale sarà:

$$O(\log n) \cdot O(\log n) = O(\log^2 n) \implies PATH \in \mathrm{DSPACE}(\log^2 n)$$

4.7.1 Relazione tra spazio e tempo

Tempo limita spazio

Data una funzione $f(n) \geq n$, di ha che:

$$\text{DTIME}(f(n)) \subseteq \text{DSPACE}(f(n))$$

Dimostrazione:
In tempo $O(f(n))$ una TM può eseguire al massimo $O(f(n))$ passi, quindi il massimo numero di celle utilizzabili è $O(f(n))$.

Spazio limita tempo

Data una funzione $f(n) \geq \log n$, di ha che:

$$\text{DSPACE}(f(n)) \subseteq \text{DTIME}(2^{O(f(n))})$$

Dimostrazione:
In una TM deterministica una configurazione non può ripetersi, sennò la TM andrebbe in loop, ma quindi il numero massimo di passi che può fare una TM che usa spazio $O(f(n))$ è uguale al numero massimo di configurazioni possibili, cioè:

$$|\Gamma|^{f(n)} \cdot |Q| \cdot n$$

Essendo $f(n) \geq \log n$ si ha che:

$$\text{n° conf.} = |\Gamma|^{f(n)} \cdot |Q| \cdot n \leq |\Gamma|^{f(n)} \cdot |Q| \cdot 2^{f(n)} = 2^{o(f(n))}$$

4.7.2 Spazio per TM non deterministiche

Classe NSPACE

Definiamo la classe dei linguaggi:

$$\text{NSPACE}(s(n)) = \{L \in \text{DEC} \mid L \text{ decidibile con spazio } O(s(n))\}$$

Vengono considerate solo le TM non deterministiche e viene considerato il massimo tra tutti i rami di computazione.

Classe NL

Definiamo la classe:

$$\text{NL} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}(\log n)$$

cioè l'insieme di tutti i linguaggi decidibili con spazio logaritmico da una TM non deterministica.

Classe NPSPACE

Definiamo la classe:

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

cioè l'insieme di tutti i linguaggi decidibili con spazio polinomiale da una TM non deterministica.

Classe NEXPSPACE

Definiamo la classe:

$$\text{NEXPSPACE} = \bigcup_{k\in\mathbb{N}} \text{NSPACE}(2^{n^k})$$

cioè l'insieme di tutti i linguaggi decidibili con tempo esponenziale da una TM non deterministica.

Teorema di Savich

Data una funzione $f(n) \geq \log n$, si ha che:

$$\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f^2(n))$$
$$\text{NSPACE}(f(n)) \subseteq \text{DTIME}(2^{O(f(n))})$$

Dimostrazione:
Dato un linguaggio A per cui esiste una NTM N tale che $L(N) = A$ con spazio $O(f(n))$.
Questa NTM ha al massimo $2^{O(f(n))}$ configurazioni possibili durante la sua computazione, facciamo inoltre si che abbia una sola configurazione accettante a cui tutte le altre si riconducono, cioè:

$$C_{acc} = q_{accept}\sqcup; 1$$

Trasformiamo la computazione di $N(x)$ in un grafo $G_{N,x}$ in cui ogni nodo rappresenta una possibile configurazione di $N(x)$ e un arco collega due configurazioni (C, C') se la configurazione C' può essere raggiunta dalla configurazione C tramite δ.
A questo punto possiamo dire che:

$$N(x)\text{accetta} \iff \exists C_{start} \to C_{acc} \text{ in } G_{N,x}$$

Questo è controllabile tramite la funzione $PATH?(C_{start}, C_{acc}, \lceil \log m \rceil)$, eseguibile con spazio $O(\log^2 m)$, con m che rappresenta la grandezza del grafo, cioè $2^{O(f(n))}$. Quindi il costo di spazio sarà:

$$O(\log^2 m) = O(\log^2(2^{O(f(n))})) = O(f^2(n))$$

che implica che $A \in \text{DSPACE}(f^2(n))$.
Creando poi una TM M che su input x:

1. Crea il grafo $G_{N,x}$ enumerando tutti gli archi e i nodi
2. Usa l'algoritmo di marcatura per determinare se esiste un cammino $C_{start} \to C_{accept}$

Questa TM ha tempo pari alla grandezza del grafo, cioè $2^{O(f(n))}$, quindi $A \in \text{DTIME}(2^{O(f(n))})$.

4.7.3 Equivalenza tra PSPACE e NPSPACE

$PATH$ decidibile con $\log n$ spazio

Dato il linguaggio:

$$PATH = \{< G, s, t > | G \text{ è un grafo ed esiste un cammino } s \to t\}$$

si ha che:

$$PATH \in \text{NL}$$

Dimostrazione:
Definisco la NTM N che su input $< G, s, t >$:

1. Calcola $n = |V|$ con spazio $O(\log n)$
2. Salva `currentnode = s`
3. Per ogni nodo $v \in V$:
   - 3.1 Prende non deterministicamente un nodo $u \in V$:
   - 3.2 Se $(currentnode, u) \in E$:
     - 3.2.1 Imposta `currentnode = u`
     - 3.2.2 Se currentnode è uguale a t accetta
   - 3.3 Se $(currentnode, u) \notin E$ rifiuta
4. Rifiuta

Questa NTM decide $PATH$ in spazio $\log n$, quindi $PATH \in \text{NL}$.

Equivalenza tra PSPACE e NPSPACE

Date le classi PSPACE e NPSPACE si ha che:

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

Relazione totale tra tempo e spazio

Dati tutte le classi di linguaggi precedenti, si ha che:

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$
$$\text{NPSPACE} \subseteq \text{EXP} \subseteq \text{NEXP} \subseteq \text{EXPSPACE} \subseteq \text{NEXPSPACE}$$

Sapendo che $\text{P} \neq \text{EXP}$ e $\text{PSPACE} = \text{NPSPACE}$, questo implica che:

$$\text{P} \neq \text{PSPACE} \vee \text{PSPACE} \neq \text{EXP}$$

4.7.4 Riducibilità in spazio logaritmico

Riduzione tramite mappatura in spazio logaritmico

Un linguaggio A è riducibile ad un'altro linguaggio in spazio logaritmico B, scritto $A \leq_m^L B$, se:

$$\exists f : \Sigma^* \to \Sigma^* | \forall w \in \Sigma^* \; w \in A \iff f(w) \in B$$

La funzione f deve avere utilizzare spazio logaritmico.
Si avrà inoltre che:

$$B \in \text{L} \implies A \in \text{L}$$
$$B \in \text{NL} \implies A \in \text{NL}$$

La relazione $\leq_m^L$ è inoltre transitiva.

Visto che l'output $f(w)$ potrebbe avere lunghezza polinomiale rispetto a x, viene considerato un nastro di output in cui si può scrivere solo una volta e non viene contato nel calcolo dello spazio della riduzione.

Linguaggio NL-Completo

Un linguaggio A si dice **NL-completo** se:

- $A \in \text{NL}$
- $\forall L \in \text{NL} \; L \leq_m^L A$

$PATH$ NL-Completo

Dato il linguaggio:

$$PATH = \{< G, s, t > | G \text{ è un grafo ed esiste un cammino } s \to t\}$$

si ha che:

$$PATH \in \text{NL-Completo}$$

Dimostrazione:
Dato un linguaggio $A \in \text{NL}$ e sia N la NTM per cui $L(N) = A$ con spazio logaritmico, costruisco la funzione TM R che su input x:

1. Costruisce il grafo delle configurazioni $G_{N,x}$
2. Ritorna $(G_{N,x}, C_{start}, C_{acc})$

Quindi possiamo dire che:

$$x \in A \iff N(x) \text{ accetta} \iff \exists C_{start} \to C_{acc} \text{ in } G_{N,x}$$
$$\iff R(x) = (G_{N,x}, C_{start}, C_{acc}) \in PATH$$

Usando R spazio polinomiale per costruire il grafo si ha che $A \leq_m^L PATH$, quindi $PATH \in$ NL-Completo.

Linguaggio P-Completo

Un linguaggio A si dice **P-completo** se:

- $A \in \text{P}$
- $\forall L \in \text{P} \;\; L \leq_m^L A$

Un esempio di linguaggio P-Completo è:

$$CIRCUIT - EVAL = \{< C, x > | C(x) = 1\}$$

Linguaggio PSPACE-Completo

Un linguaggio A si dice **PSPACE-completo** se:

- $A \in \text{PSPACE}$
- $\forall L \in \text{PSPACE} \;\; L \leq_m^P A$

Basta che la riduzione sia in tempo polinomiale perché il tempo limita lo spazio, quindi sarà anche in spazio polinimiale.

***TBQL* PSPACE-Completo**

Dato il linguaggio:

$$TBQL = \{< \phi > | Q_1 x_1, \ldots, Q_n x_n \wedge \phi(x_1, \ldots, x_n) = 1\}$$

in cui Q sono quantificatori $(\forall, \exists)$ e x literali, si ha che:

$$TBQL \in \text{PSPACE-Completo}$$

Dimostrazione:
Definiamo un algoritmo ricorsivo $isTrue$ che su input $< Q_1 x_1, \ldots, Q_n x_n, \phi(x_1, \ldots, x_n) >$:

1. Se $n = 0$, quindi ci sono solo costanti:
   1.1 Ritorna la valutazione di $\phi(x_1, \ldots, x_n)$
2. Se $n > 0$ allora per ogni quantificatore Q_i:
   2.1 Se $Q_i = \exists$:
      2.1.1 Ritorna:

$$isTrue(Q_{i+1} x_{i+1}, \ldots, Q_n x_n, \phi(0, x_2, \ldots, x_n)) \vee isTrue(Q_{i+1} x_{i+1}, \ldots, Q_n x_n, \phi(1, x_2, \ldots, x_n))$$

   2.2 Se $Q_i = \forall$:

2.2.1 Ritorna:

$$isTrue(Q_{i+1}x_{i+1}, \ldots, Q_n x_n, \phi(0, x_2, \ldots, x_n)) \\ \wedge \\ isTrue(Q_{i+1}x_{i+1}, \ldots, Q_n x_n, \phi(1, x_2, \ldots, x_n))$$

Quindi $TQBF$ è decidibile in spazio polinomiale, cioè $TQBF \in$ PSPACE.
Dato un linguaggio $A \in$ PSPACE con quindi una TM M per cui $L(M) = A$ con spazio polinomiale.
La TM avrà $2^{O(n^k)}$ configurazioni e considerando il grafo delle configurazioni, definiamo la procedura ricorsiva $\phi_k(C_i, C_j)$ che:

1. Se $k = 0$:
   1.1 Ritorna $(C_i, C_j) \vee (C_i = C_j)$, in cui (C_i, C_j) indica che C_j è raggungibile tramite δ da C_i
2. Se $k > 0$:
   2.1 Ritorna:
$$\exists C_m \forall D, D' \left(\begin{array}{c} (D, D') = (C_i, C_m) \\ \vee \\ (D, D') = (C_m, C_i) \end{array} \right) \implies \phi_{k-1}(D, D')$$

Questa procedura ha complessità spaziale pari a:

$$O(n^k) + |\phi_{k-1}| = O(k \cdot n^k) = \phi_{O(n^k)} = O(n^{2k})$$

Quindi spazio polinomiale, allora possiamo costruire una TM R che su input x:

1. Crea il grafo delle configurazioni $G_{M,x}$, che però essendo la TM detrministica avrà un solo arco uscente per ogni nodo
2. Esegue $\phi_k(C_{start}, C_{acc})$

Quindi $A \leq_m^P TQBF$ e allora $TQBF \in$ PSPACE-Completo.

Equivalenza tra NL e coNL

Date le classi di linguaggi NL e coNL, si ha che:

$$\text{NL} = \text{coNL}$$

Dimostrazione:
Per dimostrarlo basta dimostrare che $\overline{PATH} \in$ NL, perché questo implicherebbe che $PATH \in$coNL e essengo NL-Completo implicherebbe che tutti i linguaggi in NL sono in coNL e quindi NL =coNL.
Dato l'input di $PATH$, cioè $< G, s, t >$ definiamo:

- R_l, l'insieme di tutti i vertici raggiungibili da s in al massimo l passi
- $r_l = |R_l|$

Per decidere $\overline{PATH}$ usiamo un verificatore V che controllerà un certificato nella forma:

$$\text{certificato } r_1, \text{certificato } r_2, \dots,$$
$$\text{certificato } r_n, \text{certificato } \nexists s \to t \in G$$

Il certificato finale, cioè quello "$\nexists s \to t \in G$", avrà forma:

$$s \to v_i, s \to v_j, \dots, s \to v_n$$

in cui v_i, v_j, v_n sono dei nodi raggiungibili da s in massimo l passi e $s \to v_i$ contiene tutti i noi nel cammino.
Per controllare questo certificato, V esegue la procedura:

1. Controlla che tutti cammini nel certificato siano raggiungibili da s
2. Controlla che il certificato abbia r_n cammini
3. Controlla che i nodi finali dei cammini siano diversi da t
4. Controlla che i nodi finali dei cammini siano tutti diversi, per fare questo si da per scontato che i nodi siano in ordine lessicografico

Questa proceura è eseguibile in spazio logaritmico.
Per controllare un generico certificato per r_{l+1} dopo aver controllato il cerificato per r_l, che sarà della forma:

$$v_1 \in R_{l+1}, v_2 \notin R_{l+1}, \dots, v_n \in R_{l+1}$$

Bisogna controllare le due possibili casistiche:

- $v_i \in R_{l+1}$: V controlla che esista il cammino $s \to v_1$ con massimo $l+1$ passi in G
- $v_i \notin R_{l+1}$: viene verificato nello stesso modo del certificato "$\nexists s \to t \in G$"

Anche questo è fattibile in spazio logaritmico, quindi $\overline{PATH} \in$ NL.

4.8 Teoremi di gerarchia

4.8.1 Teorema di gerarchia per TM

Funzione tempo costruibile

Una funzione $t : \mathbb{N} \to \mathbb{N}$ con $t(n) \geq \log n$ è **tempo costruibile** se la funzione

$$g : 1^n \to 1^{t(n)}$$

è calcolabile in tempo $O(t(n))$.

Teorema di gerarchia temporale

Date due funzioni $t_1(n) < t_2(n)$ tempo costruibili, allora esiste un linguaggio:

$$L \in \text{DTIME}(t_2(n)) - \text{DTIME}(t_1(n))$$

Dimostrazione:
Definiamo una codifica:

$$[\cdot]_{\text{TM}} : \Sigma^* \to \{\text{TM}\}$$

che trasforma una stringa in una codifica di TM, nel caso la codifica non sia valida viene mappata ad una TM di default. Si ha quindi che per ogni TM M esiste una stringa x per cui $[x]_{\text{TM}} = M$.
Costruiamo una TM D che su input x:

1. Calcola $[x]_{\text{TM}} = M$
2. Simula $M(x)$ per $t_{1.5}(n)$ passi con $t_1(n) < t_{1.5}(n) < t_2(n)$
3. Se $M(x)$ accetta, D rifiuta e viceversa
4. Se $M(x)$ non ha terminato D accetta

La TM D è un decisore per cui $L(D) \in \text{DTIME}(t_2(n))$.
Presa una qualsiasi TM Q con tempo $t_1(n)$, D con un qualsiasi input x avrà sempre risultato opposto rispetto a Q, implicando che $D \neq Q$. Essendo D diversa da qualsiasi TM che ha tempo $t_1(n)$, allora $L(D) \notin \text{DTIME}(t_1(n))$.

Funzione spazio costruibile

Una funzione $s : \mathbb{N} \to \mathbb{N}$ con $s(n) \geq \log n$ è **spazio costruibile** se la funzione

$$g : 1^n \to 1^{s(n)}$$

è calcolabile in spazio $O(s(n))$.

Teorema di gerarchia spaziale

Date due funzioni $s_1(n) < s_2(n)$ spazio costruibili, allora esiste un linguaggio:

$$L \in \text{DSPACE}(s_2(n)) - \text{DSPACE}(s_1(n))$$

4.8.2 Teoremi di gerarchia temporale per NTM

Il teorema di gerarchia spaziale è facilmente applicabile sulle NTM perchè PSPACE = NPSPACE, quindi si può convertire una NTM in una TM ed eseguire lo stesso procedimento del teorema per le TM descritto sopra.

Teorema di gerarchia di tempo per NTM

Date due funzioni $t_1(n) < t_2(n)$ tempo costruibili, per cui $t_1(n+1) = O(t_2(n))$ allora esiste un linguaggio:

$$L \in \text{NTIME}(t_2(n)) - \text{NTIME}(t_1(n))$$

Dimostrazione:
Sia N_n la NTM corrispondente alla codifica 1^n secondo la codifica delle NTM, dividiamo l'input 1^n in intervalli nella forma:

$$[l_1 = 0, u_1], [l_2, u_2], \ldots, [l_k, u_k]$$

Costruiamo una NTM D che su input 1^n:

1. Trova l'indice i per cui $l_i \leq n \leq u_i$
2. Se $n \neq u_i$:
   2.1 Simula $N_i(1^{n+1})$ per $t_2(n)$ passi
3. Se $n = u_i$:
   3.1 Simula $N_i(1^{l_i})$ in modo deterministico per $\log(t_2(n))$ passi
   3.2 Se $N_i(1^{l_i})$ accetta, D rifiuta e viceversa

La TM D è un decisore per cui $L(D) \in \text{NTIME}(t_2(n))$.
Presa la NTM N con tempo $t_1(n)$ per cui $N(1^n) = D(1^n)$ per ogni n. Sia i l'indice per cui $N_i = N$. Sappiamo per ipotesi che $N(l_i) = D(l_i)$, ma $D(l_i)$ simula $N_i(l_i + 1)$ per $t_2(l_i)$ passi e fa l'opposto. Sapendo inoltre che $t_1(l_i + 1) = t_2(l_i)$ implichiamo che $D(l_i) = N(l_i + 1) = N(l_i)$. Possiamo continuare ad aumentare fino ad arrivare ad $N(u_i)$ in cui $D(u_i)$ simula $N_i(u_i)$ per $\log(t_2(u_i))$ passi. Si ha quindi che per un u_i abbastanza grande (ad esempio $2^{t(l_i)^3}$) $N_i(l_i)$ viene eseguito per $t_1(l_i)$ passi e $D(l_i)$ quindi farà l'opposto. QUesto implica che D sia diversa da ogni NTM con tempo $t_1(n)$, quindi $L(D) \notin \text{NTIME}(t_2(n))$.

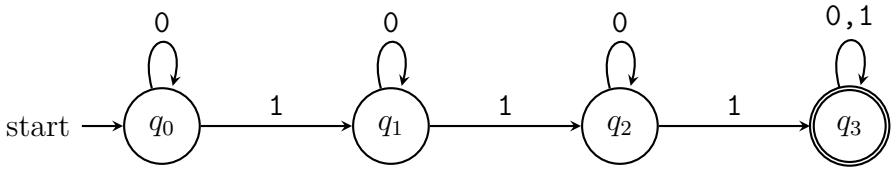
E

Esercizi

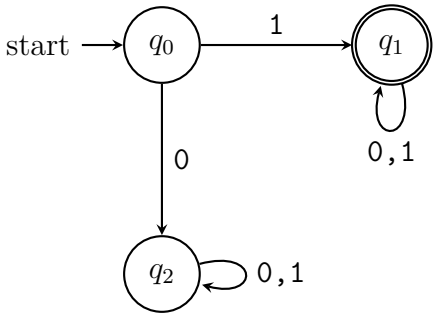
E.1 Esercizi sui linguaggi regolari

E.1.1 Costruire un automa da un linguaggio

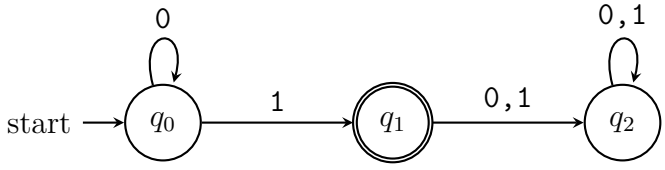
1. Dato un linguaggio $L(D) = \{x \in \{0,1\}^* | w_H(x) \geq 3\}$, per cui $w_H(x) = \{\text{n}^\circ \text{ di 1 in } x\}$, costruire un DFA che accetta questo linguaggio:



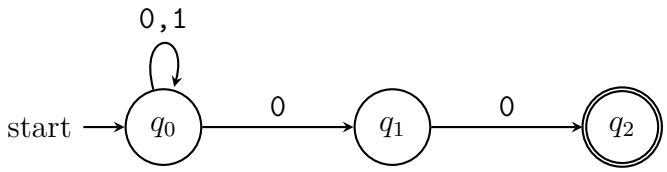
2. Dato un linguaggio $L(D) = \{x \in \{0,1\}^* | x = 1y \wedge y \in 0,1^*\}$, costruire un DFA che accetta questo linguaggio:



3. Dato un linguaggio $L(D) = \{x \in \{0,1\}^* | x = 0^n 1\}$, costruire un DFA che accetta questo linguaggio:



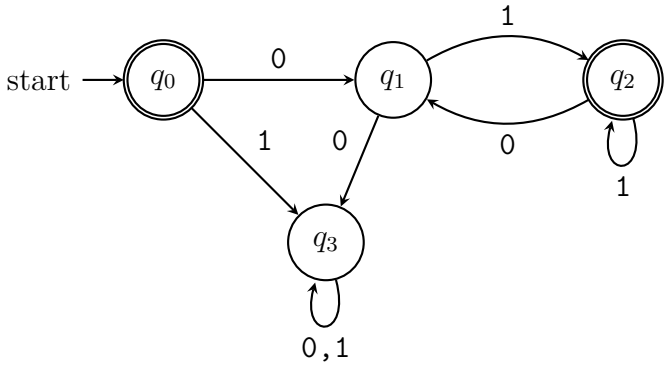
4. Dato un linguaggio $L = \{w \in \{0,1\}^* | w = x00\ x \in \{0,1\}^*\}$, costruire un NFA che accetta questo linguaggio:



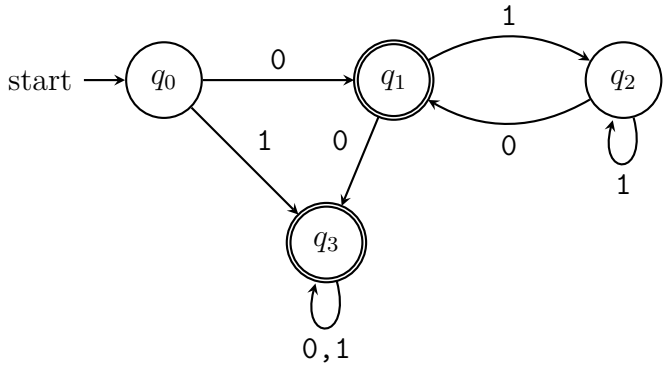
Dimostrazione per induzione:

- Caso base: $w = 00 \implies w \in L$
- Passo induttivo: $w = w'00$ con $w' = \{0,1\}^*$
  $w \in L$ perchè $\forall w'$ c'è sempre un ramo di computazione che si trova nello stato q_0. Quindi leggendo 00 alla fine arriva nello stato q_2, quindi $w \in L$.

5. Dato un linguaggio $L = \{w \in \{0,1\}^* | w \notin (01^+)^*\}$, costruire un DFA che accetta questo linguaggio:
   Posso creare un DFA che accetta il linguaggio complementare $\overline{L} = \{w \in (01^+)^*\}$:

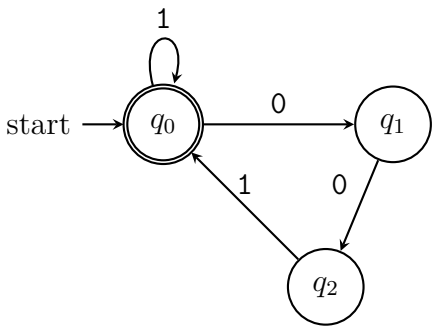


   Quindi il DFA che accetta il linguaggio iniziale è quello con gli stati accettanti invertiti rispetto a quello sopra:



E.1.2 Costruire un automa da un'espressione regolare

1. Data l'espressione regolare $r = 1^*(001^+)^*$ costruire il DFA equivalente a r:



E.1.3 Dimostrare che un linguaggio non è regolare

1. Dimostrare che il linguaggio

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

dove w^R è w rovesciata ($w = 100 \implies w^R = 001$) non è regolare:
Presa la stringa $w = 0^p 110^p \in L$ con $|w| \geq p$ e $|xy| < p$ allora y contiene solo 0.
Scrivo $w = 0^k 0^l 0^m 110^p$ con:

- $x = 0^k$ $k \geq 0$
- $y = 0^l$ $l > 0$
- $z = 0^m 110^p$
- $k + l + m = p$

Con $i = 2$ la stringa $xy^2z = 0^k 0^{2l} 0^m 110^p \implies k + 2l + m > p \implies xy^2z \notin L \implies L$ non è regolare.

2. Dimostrare che il linguaggio

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

non è regolare:
Presa la stringa $w = 1^{p^2}$ con $|w| > p$ e $|xy| < p$.
Scrivo $w = 1^k 1^l 1^{p^2-k-l}$ con:

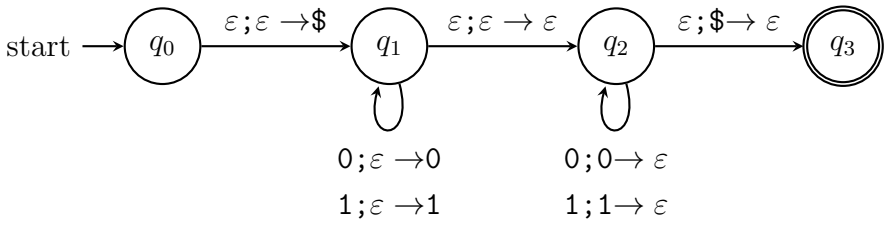
- $x = 1^k$ $k \geq 0$
- $y = 1^l$ $l > 0$
- $z = 1^{p^2-l-k}$
- $k + l \leq p$

Con $i = 2$ la stringa $xy^2z = 1^k 1^{2l} 1^{p^2-l-k} = 1^{p^2+l} \implies p^2 < |xy^2z| < (p+1)^2 \implies xy^2z \notin L \implies L$ non è regolare.

E.2 Esercizi sui linguaggi acontestuali

E.2.1 Costruire un automa da un linguaggio acontestuale

1. Dato il linguaggio $L = \{ww^R | w \in \{0,1\}^*\}$ dove w^R è w rovesciata ($w = 100 \implies w^R = 001$), costruire il PDA associato:



E.3 Esercizi sulla calcolabilità

E.3.1 Dimostrare la non decidibilità tramite riduzione

1. Il linguaggio:
$$L = \{< M > | M \in \text{TM} \wedge L(M) = \{0,1\}^{2k+1}\}$$
è indecidibile.
Per dimostarlo lo riduciamo al linguaggio A_{TM}, trovando una funzione:
$$f : \Sigma^* \to \Sigma^* | < M, w > \in A_{\text{TM}} \iff f(< M >) \in L$$
La funzione f sarà calcolata da una TM che su input $< M, w >$:
   1. Costruisce la TM M' che con input x:
      1.1 Se $|x|$ è pari, rifiuta
      1.2 Se $|x|$ è dispari esegue $M(w)$
      1.3 Se M accetta, M' accetta, sennò rifiuta
   2. Restituisce in output $< M' >$

   Dimostrazione di correttezza:

   1. $< M, w > \in A_{\text{TM}} \implies < M' > \in L$:
   Se $< M, w > \in A_{\text{TM}}$ allora M accetta w, per costruzione $L(M') = \{0,1\}^{2k+1}$ e quindi $< M' > \in L$
   2. $< M, w > \notin A_{\text{TM}} \implies < M' > \notin L$:
   Se $< M, w > \notin A_{\text{TM}}$ allora M rifiuta w o va in loop, per costruzione $L(M') = \emptyset$ e quindi $< M' > \notin L$

   Da questo segue che $A_{\text{TM}} \leq_m L$, ma poiché $A_{\text{TM}} \notin \text{DEC} \implies L \notin \text{DEC}$.

2. Il linguaggio:
$$L = \{< M > | M \in \text{TM} \wedge L(M) = \{0^n 1^n 0^n | n \leq 0\}\}$$
è indecidibile.
Per dimostarlo lo riduciamo al linguaggio A_{TM}, trovando una funzione:
$$f : \Sigma^* \to \Sigma^* | < M, w > \in A_{\text{TM}} \iff f(< M >) \in L$$
La funzione f sarà calcolata da una TM che su input $< M, w >$:
   1. Costruisce la TM M' che con input x:
      1.1 Esegue $M(w)$

1.2 Se M accetta controlla se $x = 0^n1^n0^n$

1.3 Se lo è accetta, sennò rifiuta

2. Restituisce in output $< M' >$

Dimostrazione di correttezza:

1. $< M, w > \in A_{\text{TM}} \implies < M' > \in L$:
   Se $< M, w > \in A_{\text{TM}}$ allora M accetta w, per costruzione $L(M') = \{0^n1^n0^n | n \leq 0\}$ e quindi $< M' > \in L$
2. $< M, w > \notin A_{\text{TM}} \implies < M' > \notin L$:
   Se $< M, w > \notin A_{\text{TM}}$ allora M rifiuta w o va in loop, per costruzione $L(M') = \emptyset$ e quindi $< M' > \notin L$

Da questo segue che $A_{\text{TM}} \leq_m L$, ma poiché $A_{\text{TM}} \notin \text{DEC} \implies L \notin \text{DEC}$.

3. Il linguaggio:
$$L = \{< M > | M \in \text{TM} \wedge L(M) \supseteq \{0w | w \in \Sigma^*\}\}$$
è indecidibile.
Per dimostarlo lo riduciamo al linguaggio A_{TM}, trovando una funzione:
$$f : \Sigma^* \to \Sigma^* | < M, w > \in A_{\text{TM}} \iff f(< M >) \in L$$
La funzione f sarà calcolata da una TM che su input $< M, w >$:

1. Costruisce la TM M' che con input x:

   1.1 Esegue $M(w)$

   1.2 Se M accetta, M' accetta, sennò rifiuta

2. Restituisce in output $< M' >$

Dimostrazione di correttezza:

1. $< M, w > \in A_{\text{TM}} \implies < M' > \in L$:
   Se $< M, w > \in A_{\text{TM}}$ allora M accetta w, per costruzione $L(M') \supseteq \{0w | w \in \Sigma^*\}$ e quindi $< M' > \in L$
2. $< M, w > \notin A_{\text{TM}} \implies < M' > \notin L$:
   Se $< M, w > \notin A_{\text{TM}}$ allora M rifiuta w o va in loop, per costruzione $L(M') = \emptyset$ e quindi $< M' > \notin L$

Da questo segue che $A_{\text{TM}} \leq_m L$, ma poiché $A_{\text{TM}} \notin \text{DEC} \implies L \notin \text{DEC}$.

4. Il linguaggio:
$$L = \{< M_1, M_2 > | M_1, M_2 \in \text{TM} \wedge L(M_1) \cup L(M_2) = \Sigma^*\}$$
è indecidibile.
Per dimostarlo lo riduciamo al linguaggio A_{TM}, trovando una funzione:
$$f : \Sigma^* \to \Sigma^* | < M, w > \in A_{\text{TM}} \iff f(< M >) \in L$$
La funzione f sarà calcolata da una TM che su input $< M, w >$:

1. Costruisce le TM M_1, M_2 che con input x:
   1.1 M_1 rifuta sempre
   1.2 M_2 esegue $M(w)$ e accetta se M accetta w, sennò rifiuta
2. Restituisce in output $< M_1, M_2 >$

Dimostrazione di correttezza:

1. $< M, w > \in A_{\text{TM}} \implies < M_1, M_2 > \in L$:
   Se $< M, w > \in A_{\text{TM}}$ allora M accetta w, quindi $L(M_2) = \Sigma^*$ e essendo per costruzione $L(M') = \emptyset$ si ha che $< M_1, M_2 > \in L$
2. $< M, w > \notin A_{\text{TM}} \implies < M_1, M_2 > \notin L$:
   Se $< M, w > \notin A_{\text{TM}}$ allora M rifiuta w o va in loop, quindi $L(M_2) = \emptyset$ e essendo per costruzione $L(M') = \emptyset$ si ha che $< M_1, M_2 > \notin L$

Da questo segue che $A_{\text{TM}} \leq_m L$, ma poiché $A_{\text{TM}} \notin \text{DEC} \implies L \notin \text{DEC}$.