Automi, Calcolabilità e Complessità

Alessio Marini, 2122855

Appunti presi durante il corso di **Automi, Calcolabilità e Complessità** nell'anno **2025/2026** del professore Daniele Venturi.
Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni 🙏🏻.

Contatti:
alem1105
marini.2122855@studenti.uniroma1.it

September 27, 2025

Indice

1. Introduzione alla terminologia

Introduciamo delle definizioni e delle operazioni che utilizzeremo durante il corso.

Alfabeto

É un insieme finito di simboli, quindi ad esempio $\Sigma = \{0, 1, x, y, z\}$.

Stringa

Una stringa è una sequenza di simboli che appertongono ad un alfabeto. Quindi, ad esempio, dato l'alfabeto $\Sigma = \{0, 1, x, y, z\}$ una sua stringa è $w = 01z$.

1.1. Operazioni sulle Stringhe

Lunghezza di una Stringa

Data una stringa $w \in \Sigma^*$ indichiamo la lunghezza con $|w|$ ed è definita come il numero di simboli che contiene.

Concatenazione

Data la stringa $x = x_1, ..., x_n \in \Sigma^*$ e la stringa $y = y_1, ..., y_m \in \Sigma^*$ definiamo come **concatenazione di x con y** la stringa $x \cdot y = x_1 ..., x_n y_1 ... y_m$.

Stringa Vuota

Durante il corso indicheremo con ε la stringa vuota, ovvero una stringa tale che $|\varepsilon| = 0$ Se concateniamo una qualsiasi stringa non vuota con una stringa vuota otteniamo la prima stringa:

$$\forall w \in \Sigma^* \ w \cdot \varepsilon = w$$

Conteggio

Data una stringa $w \in \Sigma^*$ e un simbolo $a \in \Sigma$ indichiamo il contaggio di a in w con $|w|_a$ e lo definiamo come il numero di occorrenze del carattere a nella stringa w.

Stringa Rovesciata

Data una stringa $w = a_1 ... a_n \in \Sigma^*$ dove $a_1, ..., a_n \in \Sigma$, definiamo la stringa rovesciata con $w^R = a_n ... a_1$..

Potenza

Data la stringa $w \in \Sigma^*$ e dato $n \in \mathbb{N}$ definiamo la potenza in modo ricorsivo:

$$w^n = \begin{cases} \varepsilon \quad \text{se} \quad n = 0 \\ ww^{\{n-1\}} \quad \text{se} \quad n > 0 \end{cases}$$

Linguaggio

Dato un alfabeto Σ definiamo Σ^* come linguaggio di Σ, ovvero l'insieme di tutte le stringhe di quell'alfabeto.

2. DFA - Automa a Stati Finiti

Il modello di computazione che utilizzeremo per ora è un DFA, questo ha una memoria limitata e permette una gestione dell'input. La memoria gli permette di memorizzare i suoi stati e tramite gli input decide in quale stato futuro muoversi.

Esempio - Una porta automatica

Una porta automatica avrà due stati:
- Aperta
- Chiusa

E due input:
- Rileva qualcuno
- Non rileva nessuno

Quindi lo stato iniziale sarà la porta chiusa, se rileva qualcuno va nello stato di aperta mentre se non rileva nessuno rimane chiusa.

DFA

Definiamo un DFA come una tupla $(Q, \Sigma, \delta, q_0, F)$ dove:
- Q è l'insieme degli stati
- Σ è l'insieme finito dei simboli in input
- $\delta : Q \times \Sigma \to Q$ è la funzione di transizione degli stati, ovvero dato lo stato in cui si trova ed un input, restituisce lo stato in cui andremo
- $q_0 \in Q$ è lo stato iniziale dell'automa
- $F \subseteq Q$ è l'insieme degli stati di accettazione dell'automa, ovvero gli stati dove l'automa si trova dopo aver riconosciuto determinate stringhe e consente la terminazione.

Dato DFA M possiamo definire l'insieme delle stringhe riconosciute dall'automa, ovvero quelle che lo portano in uno stato di accettazione come $L(M)$. Da notare che può anche accadere che $L(M) = \emptyset$. Daremo una definizione più formale di quest'ultimo più avanti.

Dati dei DFA vogliamo iniziare a definire dei linguaggi dedicati a questi, per farlo abbiamo bisogno della **funzione di transizione estesa**.

Funzione di Transizione Estesa

La definiamo come:

$$\delta^* = Q \times \Sigma^* \rightarrow Q$$

Quindi questa a differenza di quella classica non usa degli input singoli ma delle intere stringhe appartenenti al **linguaggio** del DFA.
È definibile in modo ricorsivo:

$$\begin{cases} \delta^*(q, \varepsilon) = \delta(q, \varepsilon) = q \\ \delta^*(q, aw) = \delta^*(\delta(q, a), w) \quad \text{con} \quad w \in \Sigma^* \quad \text{e} \quad a \in \Sigma \end{cases}$$

Quindi data una stringa, partiamo dal primo carattere a sinistra e andiamo avanti utilizzando la funzione di transizione fino ad arrivare ad una stringa vuota.

Adesso diamo le definizioni di **Configurazione** e **Passo di Computazione** che ci serviranno a definire più formalmente un **Linguaggio Accettato** del DFA.

Configurazione

Sia $D := (Q, \Sigma, \delta, q_0, F)$ un DFA, definiamo la coppia $(q, w) \in Q \times \Sigma^*$ come configurazione di D. Inoltre dato un $x \in \Sigma^*$, la **configurazione iniziale** è (q_0, x).

Passo di Configurazione

Indica il passaggio da una configurazione ad un'altra rispettando la funzione di transizione δ, il passaggio lo indichiamo con il simbolo $\vdash_M$ dove M indica il DFA. Possiamo dire quindi che esiste una relazione binaria fra un passo di configurazione e la funzione di transizione:

$$(p, ax) \vdash_M (q, x) \Leftrightarrow \delta(p, a) = q$$

Dove $p, q \in Q$ - $a \in \Sigma$ e $x \in \Sigma^*$. Un passaggio di configurazione può avvenire, quindi, soltanto se la funzione di transizione lo permette.

Possiamo estendere questa relazione con il simbolo $\vdash_M^*$ considerando anche la **chiusura riflessiva e transitiva**:

- **Riflessività:** $(q, x) \vdash_M^* (q, x)$
- **Transitività:** Se $(q, aby) \vdash_M (p, by) \land (p, by) \vdash_M (r, y) \Rightarrow (q, aby) \vdash_M^* (r, y)$
  - Dove $q, p, r \in Q$ - $a, b \in \Sigma$ ed $y \in \Sigma^*$

Definiamo quindi il linguaggio accettato dal DFA.

Linguaggio Accettato

Diciamo che $x \in \Sigma^*$ è accettato da un automa $M = (Q, \Sigma, \delta, q_0, F)$ se $\delta^*(q_0, x) \in F$ oppure usando la relazione del passaggio, se $(q_0, x) \vdash_M^* (q, \varepsilon)$ con $q \in F$.

3. Linguaggi Regolari

Definizione

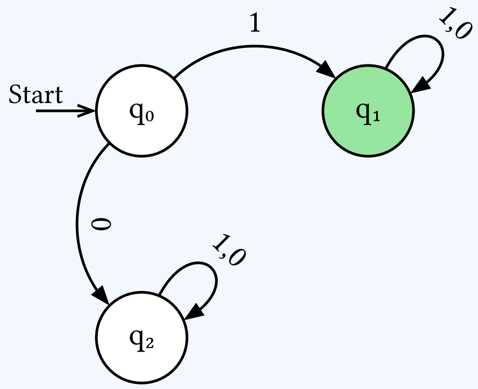
$$\text{REG} = \{L \subseteq \Sigma^* : \exists \ \text{ DFA M t.c. } \ L(M) = L\}$$

Quindi i linguaggi regolari sono tutti quei linguaggi che sono accettati da almeno un DFA.

Uno dei nostri obiettivi nel corso è quello di, dato un linguaggio, progettare dei DFA adatti.

Esempio

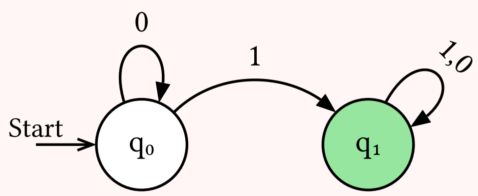
Dato il linguaggio $L = \{x \in \{0,1\}^* \ \text{ t.c. } \ x = 1y, y \in \{0,1\}^*\}$, un possibile DFA potrebbe essere:



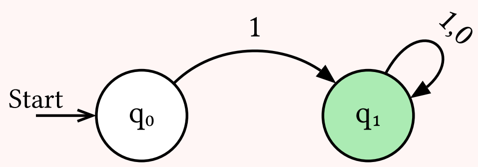
Questo DFA accetta quindi tutte le stringhe che iniziano con il simbolo 1 mentre rifiuta tutte quelle che iniziano con il simbolo 0.

Attenzione - Stato Pozzo

Notiamo che è presente lo stato q_2 dal quale il DFA non esce più una volta entrato, questo è necessario perchè se omesso il DFA accetterebbe tutte le stringhe:



Infatti in questo modo se in q_0 riceve 0 rimane su stesso ma poi continua ad attendere input.
Il modo corretto per lasciare lo stesso significato del primo DFA ma omettere lo stato q_2 è quello di omettere anche il comportamento di q_0 in caso riceviamo 0, ovvero:



Adesso dobbiamo dimostrare formalmente che questo DFA accetta il linguaggio fornito, dobbiamo quindi dimostrare che:

$$\text{DFA accetta } \; x \Leftrightarrow x \in L$$

Innanzitutto facciamo due osservazioni, ovvero che se il DFA si trova in q_1 o q_2 allora non cambierà mai più stato:

- $\delta^*(q_1, u) = q_1 \quad \forall u \in \{0,1\}^*$
- $\delta^*(q_2, u) = q_2 \quad \forall u \in \{0,1\}^*$

Dimostriamo per induzione che il linguaggio è accettato, quindi presa una stringa dobbiamo far vedere che se inizia con 1 terminiamo in q_1 altrimenti in q_2.

Dimostrazione

Caso Base

Come caso base prendiamo una stringa vuota, quindi $|x| = 0$ ovvero $x = \varepsilon$, abbiamo che:

$$\delta^*(q_0, \varepsilon) = \delta(q_0, \varepsilon) = q_0 \notin F$$

Infatti se abbiamo una stringa vuota il DFA non fa nulla e rimane in q_0

Passo Induttivo

Adesso dobbiamo prendere una stringa w tale che $|w| \leq n$ con $n > 0$, la funzione di transizione avrà quindi 3 risultati possibili:

$$\delta^*(q_0, w) = \begin{cases} q_0 & \text{se} \quad w = \varepsilon \\ q_1 & \text{se} \quad w \quad \text{inizia con} \quad 1 \\ q_2 & \text{se} \quad w \quad \text{inizia con} \quad 0 \end{cases}$$

Prendiamo quindi una stringa x tale che $|x| = n + 1$ e la costruiamo come $x = au$ con $a \in \{0,1\}$ e $u \in \{0,1\}^*$, la funzione di transizione ci restituirà:

$$\delta^*(q_0, x) = \delta^*(q_0, au) = \delta^* \left(\underbrace{\delta(q_0, a)}_{\text{ha 2 soluzioni}}, u \right)$$

Le due soluzione del passaggio evidenziato sono:

- $\delta(q_0, a) = q_1$ se $a = 1$
- $\delta(q_0, a) = q_2$ se $a = 0$

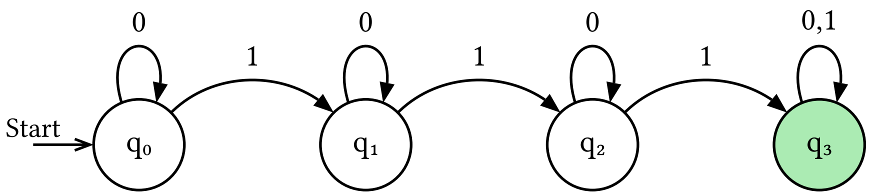
Quindi il DFA andrà sicuremante in uno dei due stati q_1 o q_2 e da lí non si muoverà più, per il ragionamento fatto all'inizio della dimostrazione.

Esercizi

DFA 1

Dato il linguaggio $L = \{x : x \in \{0,1\}^* \wedge W_H(x) \geq 3\}$ con $W_H(x) = \#1$ ovvero il numero di 1 presenti nella stringa. Progettare un automa che accetta il linguaggio e dimostrarlo.

Un possibile automa potrebbe essere:

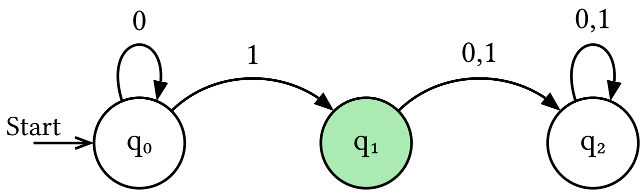


Dimostrazione: Copiare da iPad

DFA 2

Dato il linguaggio $L = \{x : x = 0^n 1 \quad \text{con} \quad n \in \mathbb{N}\}$ progettare un automa che accetta il linguaggio e dimostrarlo.

Un possibile automa potrebbe essere:



Dimostrazione: Copiare da iPad

3.1. Operazioni sui Linguaggi

Definiamo adesso delle operazioni sui linguaggi che ci torneranno utili.

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

$$L_1 \circ L_2 = \{xy : x \in L_1 \wedge y \in L_2\}$$

Da notare che questa operazione non è commutativa quindi $L_1 \circ L_2 \neq L_2 \circ L_1$

Potenza

Possiamo definirla ricorsivamente:

$$\begin{cases} L^0 = \{\varepsilon\} \\ L^{n+1} = L^N \circ L \end{cases}$$

Operatore * «star»

$$L^* = \bigcup_{n \geq 0} L^n = \{\varepsilon\} \cup L^1 \cup L^2 \cup \ldots$$

3.2. Introduzione alla proprietà di chiusura dei Linguaggi Regolari

Vogliamo capire se dati due linguaggi regolari $L_1, L_2 \in \text{REG}$ il linguaggio risultante di operazioni effettuate con questi linguaggi è regolare o no, ad esempio se $L_1 \cup L_2 \in \text{REG}$ oppure se $L_1 \cap L_2 \in \text{REG}$.

Vedremo qualche dimostrazione ma in realtà sarà più semplice dimostrare tutte le chiusure utilizzando gli NFA, ovvero gli automi non deterministici.

3.2.1. Chiusura per Unione

Teorema - Chiusura per Unione

Come prima idea possiamo dire che:

$$L_1, L_2 \in \text{REG} \Rightarrow \exists M_1, M_2 \in \text{DFA t.c. } \; L(M_1) = L_1 \wedge L(M_2) = L_2$$

Quindi dati due linguaggi regolari esistono due automi che li hanno come linguaggi accettati. Noi dobbiamo definire un terzo automa M tale che $L(M) = L_1 \cup L_2$, ma data una stringa x candidata non possiamo provare a vedere prima cosa succede su M_1 e se non la accetta provare M_2 perchè perderemmo la sequenza corretta della stringa su M.

Quello che dobbiamo fare è testare ogni carattere di x in parallelo su M_1 e M_2 e in base al risultato aggiorniamo lo stato di M.

Input Dimostrazione

Vogliamo mostrare che dati

- $M_1 = (Q_1, \Sigma, \delta_1, q_o^1, F_1)$
- $M_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$

Assumiamo lo stesso Σ per semplicità

Tali che: $L(M_1) = L_1 \wedge L(M_2) = L_2$

Costruiamo un terzo DFA $M(Q, \Sigma, \delta, q_0, F)$ t.c. $L(M) = L_1 \cup L_2$

Avremo che:

- $Q = \{(r_1, r_2) : r_1 \in Q_1, r_2 \in Q_2\} = Q_1 \times Q_2$ (Tutte le coppie di stati possibili)
- $\delta : Q \times \Sigma \to Q$
  - $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$
- $F = \{(r_1, r_2) : r_1 \in F_1 \vee r_2 \in F_2\} = \underbrace{(F_1 \times Q_2)}_{\text{il primo stato è accettato}} \cup \underbrace{(F_2 \times Q_1)}_{\text{il secondo stato è accettato}}$

Infatti basta che soltanto uno dei due stati della coppia venga accettato per accettare la coppia.

Da notare che per l'intersezione abbiamo una situazione molto simile, infatti avremo che:

$$F = \{(r_1, r_2) : r_1 \in F_1 \wedge r_2 \in F_2\} = F_1 \times F_2$$

Dimostrazione

Vogliamo mostrare che dato

$$\delta^*(q_0, x) = \delta^*((q_0^1, q_0^2), x)$$

Si ha che $\forall x \in \Sigma^*$

$$= (\delta_1^*(q_0^1, x), \delta_2^*(q_0^2, x))$$

TODO: MANCA UNA PARTE DI DIMOSTRAZIONE (Mostrare che funziona per n e 0)

$$\forall x \in \Sigma^*, x \in L(M) \Leftrightarrow x \in L_1 \cup L_2$$
$$\Rightarrow x \in L(M) \Rightarrow \delta^*(q_0, x) \in F = F_1 \times Q_2 \cup F_2 \times Q_1 = (p, q)$$
$$\text{Dove} \quad p = \delta_1^*(q_0^1, x), q = \delta_2^*(q_0^2, x))$$

Questo significa che

- Se $x \in L_1$ allora $\delta^*(q_0^1, x) \in F_1$ e quindi:

$$\delta^*(q_0, x) = (\delta_1^*(q_0^1, x), \delta_2^*(q_0^2, x)) \in F_1 \times Q_2 \Rightarrow M \quad \text{accetta} \quad x$$

- Se $x \in L_2$ allora $\delta^*(q_0^2, x) \in F_2$ e quindi:

$$\delta^*(q_0, x) = (\delta_1^*(q_0^1, x), \delta_2^*(q_0^2, x)) \in F_2 \times Q_1 \Rightarrow M \quad \text{accetta} \quad x$$

Spiegato a parole, abbiamo che la funzione di transizione dell'automa M equivale ad eseguire lo stesso input sui due automi M_1, M_2. Presa una stringa del linguaggio questa è accettata dall'automa se e solo se appartiene all'unione dei due linguaggi di M_1 e M_2.

Partendo dalla sinistra dell'implicazione abbiamo che $x \in L(M)$ quindi la stringa è accettata e allora la funzione di transizione estesa ci porta in uno stato appartenente ad F. Ricordiamo che lo stato in cui ci troviamo è in realtà una coppia di stati uno dei quali deve essere accettato o da M_1 o da M_2 e questo appunto significa rispettivamente che o $x \in L(M_1)$ oppure $x \in L(M_2)$.

Resto delle dimostrazioni

Per dimostrare il resto delle proprietà introduciamo il concetto di non determinismo.

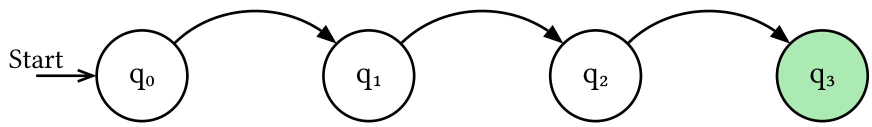
4. Non Determinismo

Per adesso abbiamo visto soltanto automi deterministici, questo significa che trovandoci in uno stato e ricevendo un input possiamo soltanto andare in un altro stato o rimanere fermi, ma in generale un solo movimento.

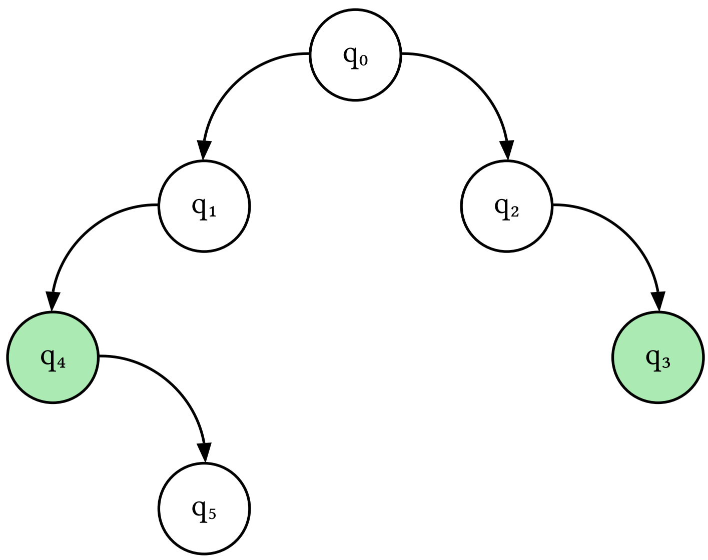
Nel **non determinismo** invece:
- Quando l'automa è in $q \in Q$ e legge $a \in \Sigma$ può andare in diversi stati
- Sono ammessi gli «ε-archi» ovvero l'automa può muoversi senza leggere input. Dallo stesso stato possono partire più «ε-archi».
- Accettazione: Se e solo se esiste un ramo che accetta, vedremo più avanti che quando studiamo un NFA avremo un albero con vari rami, se un ramo accetta allora consideriamo la stringa come accettata per il NFA.

Nel non determinismo quindi abbiamo un input che si dirama in vari stati invece che seguire un cammino di *uno stato alla volta*.

- **Determinismo**



- **Non Determinismo**



Definizione - NFA

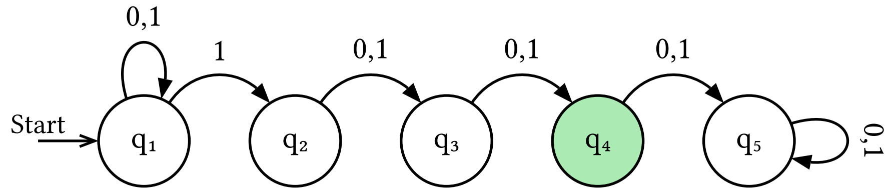
Un NFA è $(Q, \Sigma, \delta, q_0, F)$ dove Q, Σ, q_0, F sono come nei DFA ma:

$$\delta : Q \times \Sigma_\varepsilon \to \mathbb{P}(Q)$$
$$\text{Dove} \quad \Sigma_\varepsilon \cup \{\varepsilon\}$$

e $\mathbb{P}$ è l'insieme delle parti.

Vediamo un esempio e capiamo come ci si muove al loro interno.

Esempio

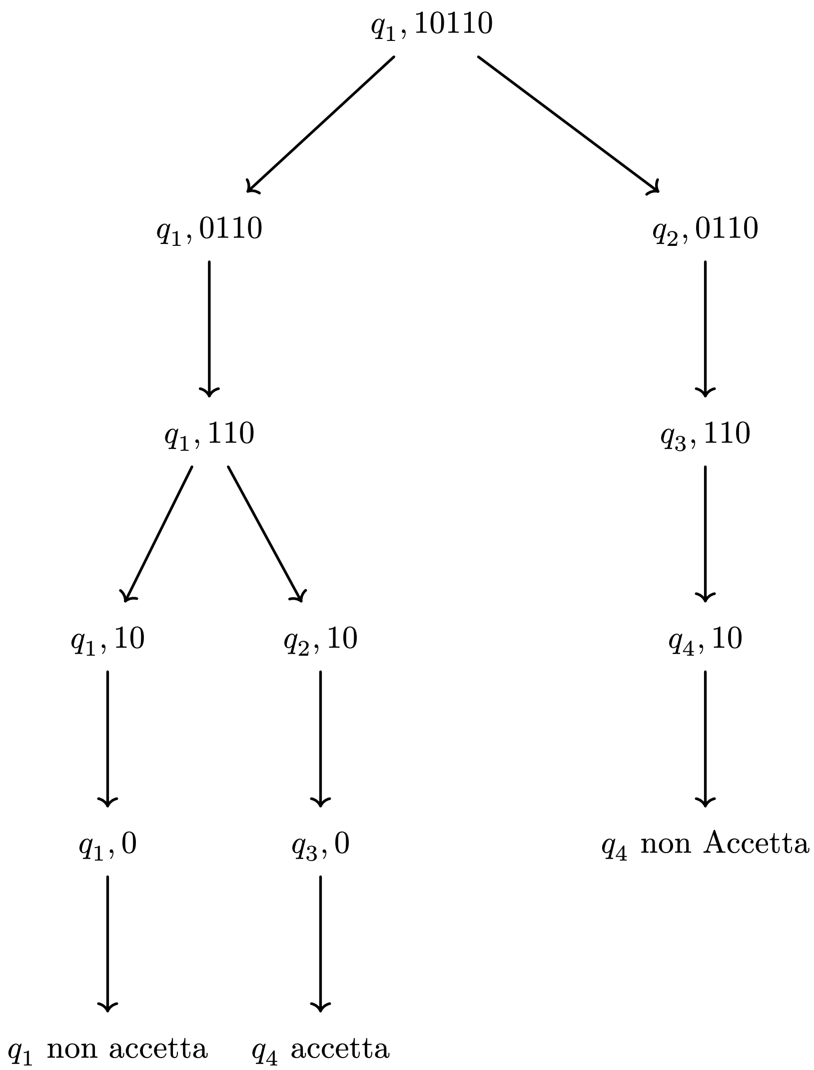


Che ha come linguaggio:

$$L = \{x : x \in \{0,1\}^* \text{ che hanno un '1' in terzultima posizione}\}$$

Da notare che lo stato q_5 possiamo anche ometterlo, ma non dobbiamo indicare in q_4 nessun arco.

Per muoverci, ad esempio nel NFA dell'esempio sopra, ci torna utile disegnare un albero con tutte i cammini che stiamo intrapendendo. Se ad esempio riceviamo in input la stringa «10110»:



4.1. Configurazione negli NFA

Possiamo estendere il concetto di **configurazione** anche per gli NFA.

Dato un NFA N indichiamo come configurazione una coppia $(q, x) \in Q \times \Sigma_\varepsilon^*$ e avremo un passo di configurazione come:

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

Con:
- $x \in \Sigma_\varepsilon^*$
- $a \in \Sigma_\varepsilon$
- $p, q \in Q$

Quindi il risultato di una transizione deve far parte dell'insieme delle parti degli stati:

$$\delta(p, a) \in \mathbb{P}(Q)$$

Quando, l'automa N, accetta $w \in \Sigma_\varepsilon^*$?
- Se e solo se $\exists q \in F \quad \text{t.c.} \quad (q_0, w) \vdash_N^* (q, \varepsilon)$. Dove $\vdash_N^*$ è la relazione estesa.

4.2. Equivalenza tra NFA e DFA

Prendiamo le due classi:
- $\mathcal{L}(\text{DFA}) \subseteq \text{REG}$
- $\mathcal{L}(\text{NFA}) = \{L : \exists \text{ NFA N t.c. } \mathcal{L}(N) = L\}$

Teorema - Per ogni automa finito non deterministico esiste un automa finito deterministico equivalente.

Dimostrazione. Dobbiamo dimostrare la doppia implicazione $\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{NFA})$ e $\mathcal{L}(\text{NFA}) \subseteq \mathcal{L}(\text{DFA})$.

La **prima implicazione** è molto semplice infatti dato un linguaggio $L \in \mathcal{L}(\text{DFA})$ e un DFA D tale che $L = L(D)$ e siccome gli NFA sono una generalizzazione dei DFA avremo che D è anche un NFA e quindi $L \in \mathcal{L}(\text{NFA})$. Quindi $\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{NFA})$.

Per la **seconda implicazione** prendiamo un NFA $= \left(Q_N, \Sigma, \delta_N, q_0^N, F_N\right)$ che riconosce un linguaggio A. Dobbiamo costruire un DFA $D = \left(Q_D, \Sigma, \delta_D, q_0^D, F_D\right)$ che riconosce A.

Consideriamo il caso in cui non abbiamo $\varepsilon -$ archi:
1. $Q_D = \mathbb{P}(Q_N)$ - Uno stato del DFA equivale quindi ad un insieme di stati del NFA.
2. Presi un $R \in Q_D$ e $a \in \Sigma$, sia

$$\delta_D(R, a) = \{q \in Q_N : q \in \delta_N(r, a) \quad \text{per qualche} \quad r \in R\}$$

Quindi la funzione di transizione del DFA equivale ad eseguire la transizione su tutti gli stati di R nel NFA.

Possiamo anche scriverla come:

$$\delta_D(R, a) = \bigcup_{r \in R} \delta(r, a)$$

3. $q_0^D = \{q_0^N\}$

4. $F_D = \{R \in Q : R \text{ contiene uno stato accettante di } N\}$ - Quindi il DFA accetta se e solo se nell'insieme risultante della transizione abbiamo almeno uno stato accettante dell'NFA. I due automi sono equivalenti.

Adesso consideriamo il caso con gli ε-archi, introduciamo delle notazioni. Per ogni R di D definiamo $E(R)$ come la collezione di stati che possono essere raggiunti dagli elementi di R proseguendo solo con ε-archi, includendo anche gli stessi elementi di R, in modo formale possiamo dire:

$$E(R) = \{q : q \text{ può essere raggiunto con } \geq 0 \quad \varepsilon - \text{archi}\}$$

Adesso modifichiamo la funzione di transizione di D in modo da far aggiungere gli stati che possono essere raggiunti da $\varepsilon -$ archi dopo ogni passo, sostituendo $\delta_N(r, a)$ con $E(\delta_N(r, a))$:

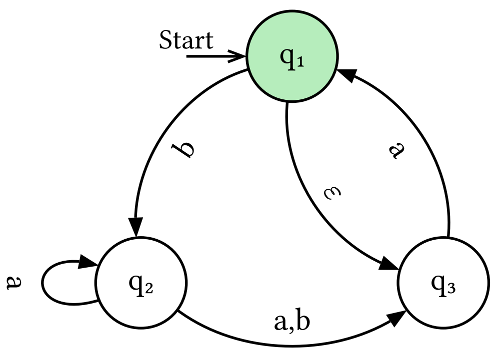
$$\delta_D(R, a) = \{q \in Q : q \in E(\delta_N(r, a)) \text{ per qualche } r \in R\}$$

Dobbiamo anche modificare lo stato iniziale di D in modo che anche nello stato iniziale raggiunga subito tutti gli stati possibili tramite $\varepsilon -$ archi e lo facciamo cambiando q_0^D in $E\left(\{q_0^N\}\right)$.

Abbiamo completato la costruzione del DFA equivalente ad NFA, infatti ad ogni passo del NFA avremo che il DFA entra in uno stato equivalente all'insieme degli stati in cui si trova l'NFA.

4.3. Convertire un NFA in DFA

Prendiamo come esempio l'NFA:



Iniziamo a definire gli elementi del DFA D:

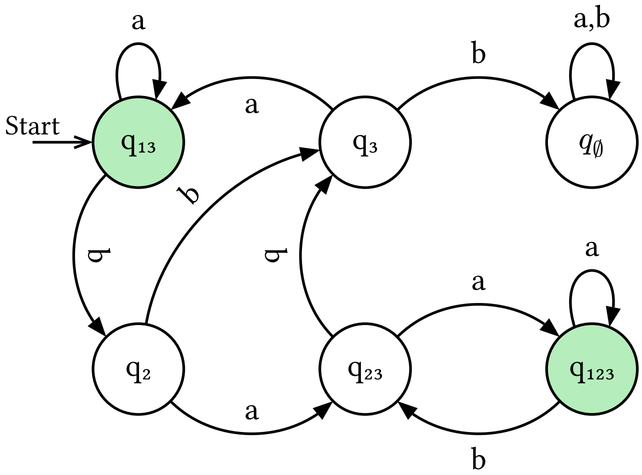
- $Q_D = \left\{q_0, q_{\{1\}}, q_{\{2\}}, q_{\{3\}}, q_{\{1,2\}}, q_{\{1,3\}}, q_{\{2,3\}}, q_{\{1,2,3\}}\right\}$
- $q_0^D = E(\{q_1\}) = q_{\{1,3\}}$ - Consideriamo quindi l'estensione dello stato iniziale con gli $\varepsilon -$ archi
- $F_D = \left\{q_{\{1\}}, q_{\{1,2\}}, q_{\{1,3\}}, q_{\{1,2,3\}}\right\}$ - Sono tutti gli stati che contengono almeno uno stato accettante, in questo caso soltanto q_1

Adesso dobbiamo calcolare δ_D, vediamo alcuni casi ma non tutti:

- $\delta_D\left(q_{\{2\}}, a\right) = q_{\{2,3\}}$
- $\delta_D\left(q_{\{2\}}, b\right) = q_{\{3\}}$

- $\delta_D\left(q_{\{3\}}, a\right) = q_{\{1,3\}}$ - Perché dobbiamo considerare anche l'ε-archi
- $\delta_D\left(q_{\{3\}}, b\right) = q_{\{\emptyset\}}$ - Infatti finisce la stringa ma non siamo in uno stato accettante

Ci sarebbero altre funzioni, ma vediamo cosa otteniamo:

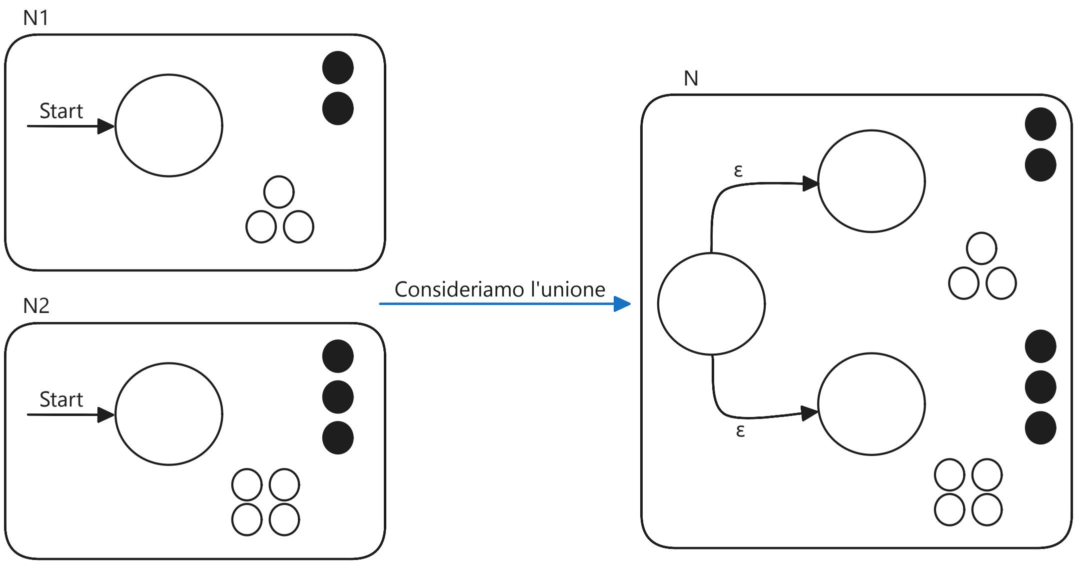


5. Proprietà di chiusura dei Linguaggi Regolari

5.1. Chiusura per Unione

Rivediamo l'unione utilizzando gli NFA, infatti adesso sappiamo che NFA e DFA sono equivalenti.

Uniamo due DFA in un NFA equivalente ai due:



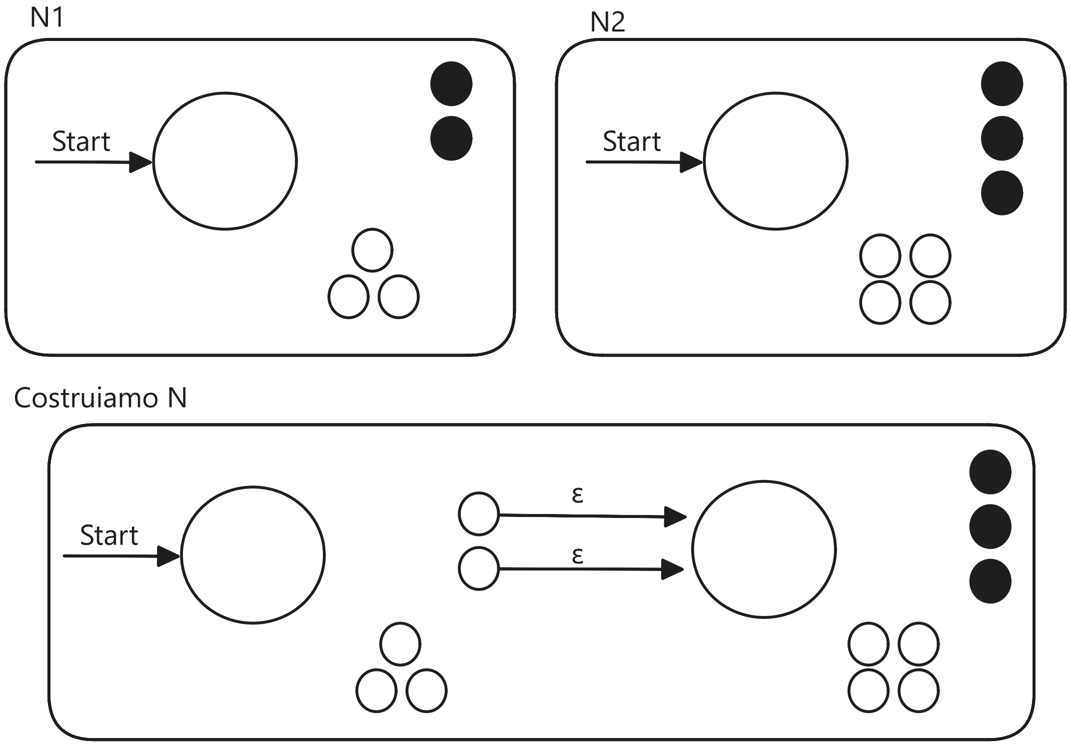
Infatti possiamo semplicemente considerare un NFA che come primo stato ci porta in modo parallelo su entrambi i DFA, avremo quindi che:

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$
- $F = F_1 \cup F_2$
- $\forall q \in Q, a \in \Sigma_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) \quad \text{se} \quad q \in Q_1 \\ \delta_2(q,a) \quad \text{se} \quad q \in Q_2 \\ \{q_0^1, q_0^2\} \quad \text{se} \quad q = q_0 \wedge a = \varepsilon \\ \emptyset \quad \text{se} \quad q = q_0 \wedge a \neq \varepsilon \end{cases}$$

5.2. Chiusura per Concatenazione

Dati due NFA N_1, N_2 per L_1, L_2 costruisco NFA per $L = L_1 \circ L_2$



Quindi li stati finali di N_1 li facciamo diventare dei normali stati che però hanno un $\varepsilon -$ arco verso lo stato iniziale di N_2

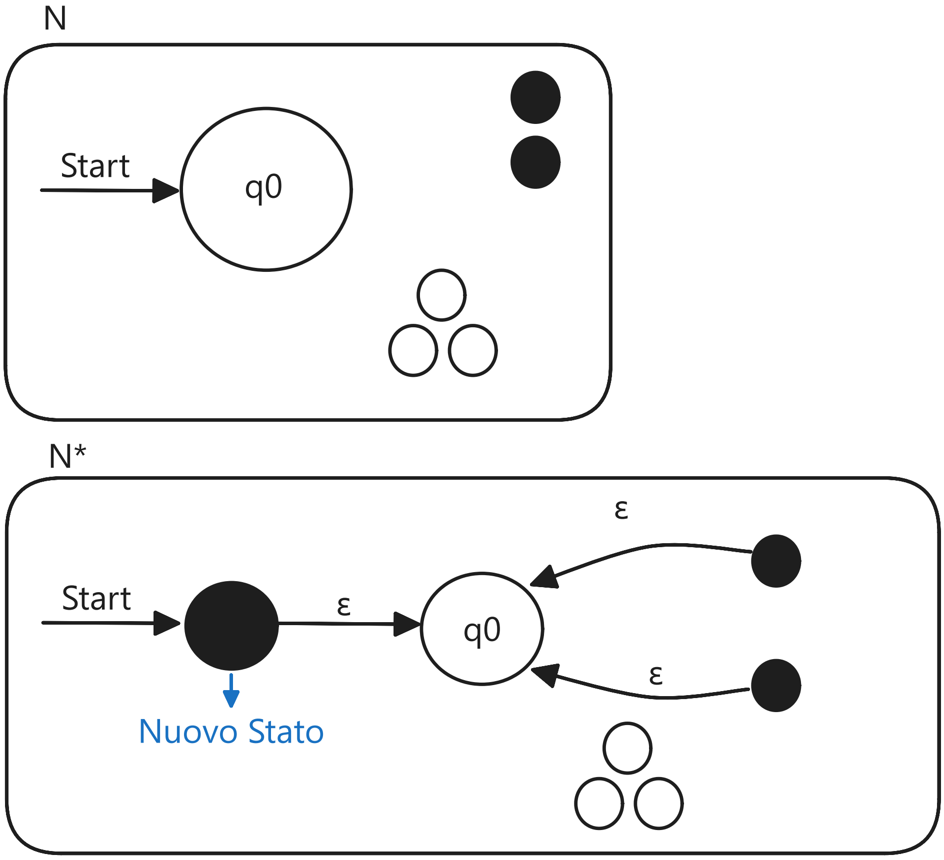
Formalmente:

- $N = \{Q, \Sigma, \delta, q_0, F\}$
- $Q = Q_1 \cup Q_2$
- $q_0 = q_0^1$
- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$
- $F = F_2$
- $\forall q \in Q, \forall a \in \Sigma_\varepsilon$:

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{se} \quad q \in Q_1 \land q \notin F_1 \\ \delta_1(q, a) & \text{se} \quad q \in F_1 \land a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_0^2\} & \text{se} \quad q \in F_1 \land a = \varepsilon \\ \delta_2(q, a) & \text{se} \quad q \in Q_2 \end{cases}$$

5.3. Chiusura per Operazione «*» star

Dato un NFA N t.c. $L(N) = L$ devo costruire NFA N^* t.c. $L(N^*) = L^*$



Formalmente abbiamo $N^* = (Q', \Sigma, \delta', {q_0}', F')$ e $N = (Q, \Sigma, \delta, q_0, F)$:

- ${q_0}'$ nuovo stato iniziale
- $F' = F \cup \{{q_0}'\}$
- $Q' = Q \cup \{{q_0}'\}$
- $\forall q \in Q', \forall a \in \Sigma_\varepsilon$:

$$\delta(q, a) = \begin{cases} \delta(q, a) \ \text{ se } \ q \in Q \land q \notin F \\ \delta(q, a) \ \text{ se } \ q \in F \land a \notin \varepsilon \\ \delta(q, a) \cup \{q_0\} \ \text{ se } \ q \in F \land a = \varepsilon \\ \{q_0\} \ \text{ se } \ q = {q_0}' \land a = \varepsilon \\ \emptyset \ \text{ se } \ q = {q_0}' \land a \neq \varepsilon \end{cases}$$

6. Espressioni Regolari

Possiamo vederle come delle espressioni algebriche, ma definiscono dei linguaggi su un certo alfabeto.

Esempio

$$(0 \cup 1) \bigcirc 0^* \quad \text{che equivale a} \quad \{0, 1\} \bigcirc 0^*$$

Definizione

Sia Σ un alfabeto possiamo definire un'espressione regolare ssu Σ (denotata con $\mathrm{re}(\Sigma)$) in modo ricorsivo:

$$\text{caso base} \begin{cases} \emptyset \in \mathrm{re}(\Sigma) \\ \varepsilon \in \mathrm{re}(\Sigma) \\ a \in \mathrm{re}(\Sigma) \quad \text{con} \quad a \in \Sigma \end{cases}$$

$$\text{induzione} \begin{cases} R_1 \cup R_2 \quad \text{se} \quad R_1, R_2 \in \mathrm{re}(\Sigma) \\ R_1 \bigcirc R_2 \quad \text{se} \quad R_1, R_2 \in \mathrm{re}(\Sigma) \\ (R_1)^* \quad \text{se} \quad R_1 \in \mathrm{re}(\Sigma) \end{cases}$$

Ogni espressione regolare ha un solo linguaggio associato:

$$L(r) \quad \text{t.c.} \quad r \in \mathrm{re}(\Sigma)$$

Vediamolo ricorsivamente:

$$\text{caso base} \begin{cases} L(r) = \emptyset \quad \text{se} \quad r = \emptyset \\ L(r) = \varepsilon \quad \text{se} \quad r = \varepsilon \\ L(r) = \{a\} \quad \text{se} \quad r = a \end{cases}$$

$$\text{induzione} \begin{cases} L(r) = L(R_1) \cup L(R_2) \quad \text{se} \quad r = R_1 \cup R_2 \\ L(r) = L(R_1) \bigcirc L(R_2) \quad \text{se} \quad r = R_1 \bigcirc R_2 \\ L(r) = (L(R_1))^* \quad \text{se} \quad r = R_1^* \end{cases}$$

Qualche esempio:

- $0^*10^* = \{w : w \;\; \text{contiene esattamente un } 1\}$
- $\Sigma^*1\Sigma^* = \{w : w \;\; \text{contiene almeno un } 1\}$
- $\Sigma^*001\Sigma^* = \{w : w \;\; \text{contiene } 001 \text{ come sottostringa}\}$
- $(0 \cup 1000)^* = \{w : \text{ogni occorrenza di } 1 \text{ é seguita da } 000\}$

Teorema

Un linguaggio regolare è regolare se e solo se esiste un'espressione regolare che lo descrive:

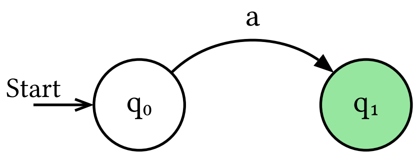
$$\text{REG} \equiv L(\text{re})$$

Per dimostrarlo dobbiamo dimostrare la doppia implicazione $L(\text{re}) \subseteq \text{REG}$ e $\text{REG} \subseteq L(\text{re})$.

Iniziamo dimostrando $L(\text{re}) \subseteq \text{REG}$, quindi data un'espressione regolare r costruiamo un NFA N_r tale che $L(N_r) = L(r)$.

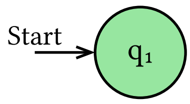
I casi base sono 3, quando l'espressione regolare è un solo carattere, quando l'espression regolare è la stringa vuota oppure quando è l'insieme vuoto.

Data l'espressione regolare $r = a$ con $a \in \Sigma$ costruiamo l'NFA che la riconosce:



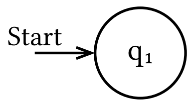
Dove:
- $N_r = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$
- $\delta(q_0, a) = q_1$
- $\delta(q, b) = \emptyset$ con $q \neq q_1 \land b \neq a$

Se invece abbiamo $r = \varepsilon$, possiamo costrutire:



Definiamo:
- $N_r = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$
- $\delta(q_1, b) = \emptyset, \forall b \in \Sigma$

Se invece $r = \emptyset$ costruiamo:



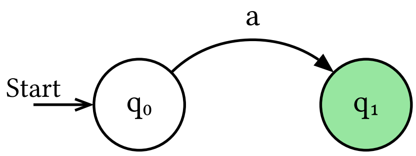
Definiamo:
- $N_r = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$
- $\delta(q_1, b) = \emptyset, \forall b \in \Sigma$

Adesso per il caso induttivo dobbiamo considerare un'espressione regolare $r = R_1 \cup R_2$ e per ipotesi induttiva sappiamo che $\exists M_1, M_2$ t.c. $L(M_1) = L(R_1) \land L(M_2) = L(R_2)$ e per chiusura di REG sappiamo che questo implica che $\exists M \;\text{ t.c. }\; L(M) = L(R_1) \cup L(R_2)$. Questo è analogo per la concatenazione e l'operatore star.

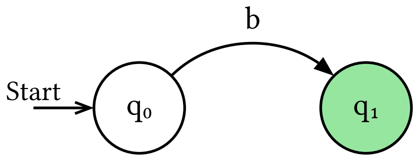
Esempio

Vogliamo costruire un NFA che riconosce il linguaggio $(\mathrm{ab} \cup \mathrm{a})^*$

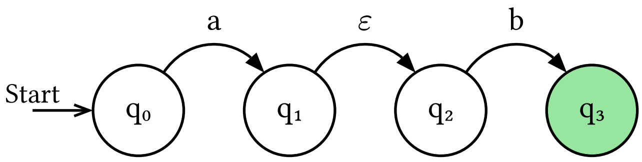
- Possiamo costruire prima l'automa che riconosce a:



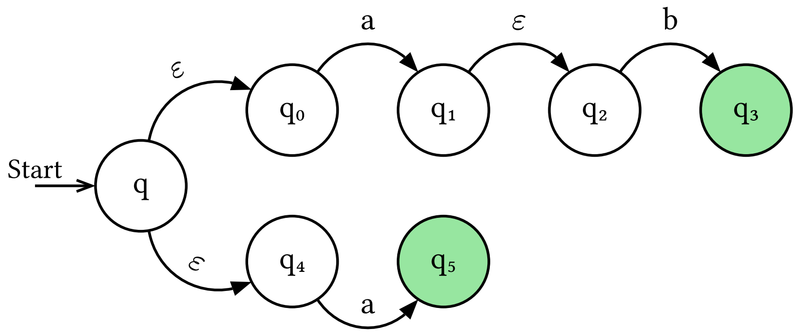
- Poi costruiamo l'automa che riconosce b:



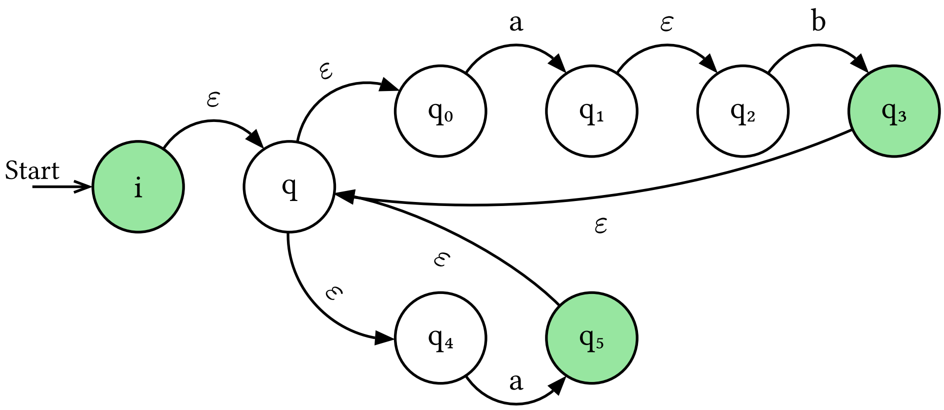
- Adesso possiamo ricavare quello che riconosce ab:



- Costruiamo l'automa che riconosce $(\mathrm{ab} \cup \mathrm{a})$:



- Infine costruiamo l'NFA che riconosce il linguaggio dell'espressione regolare:



Adesso dobbiamo dimostrare la seconda implicazione, quindi $\mathrm{REG} \subseteq L(\mathrm{re})$. Dobbiamo quindi prendere un NFA e vogliamo trovare l'espressione regolare corrispondente.

Prima di farlo vediamo come **convertire un NFA in un'espressione regolare**.

6.1. Convertire NFA in espressione regolare

Partiamo da un NFA N con $L \in \text{REG} : L(N) = L$ e introduciamo il concetto di **GNFA (NFA Generalizzato)**.

Per GNFA intendiamo un NFA dove le etichette degli archi sono delle espressioni regolari.

Forma Canonica del GNFA

Un GNFA si trova in forma canonica quando:
- Lo stato iniziale ha solo archi uscenti verso tutti gli altri stati
- Lo stato finale ha solo archi entranti
- Fatta eccezione per lo stato finale ed iniziale esiste un arco fra ogni coppia di stati

Più formalmente dato GNFA=$(Q, \Sigma, \delta, q_{\text{START}}, q_{\text{ACC}})$:

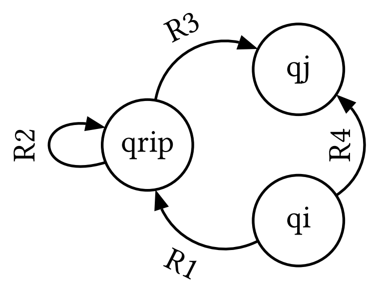
$$\delta : Q \setminus \{q_{\text{ACC}}\} \times Q \setminus \{q_{\text{START}}\} \to \mathcal{R} = \text{re}(\Sigma)$$

Dove $\mathcal{R}$ è l'insieme di tutte le espressioni regolari sul linguaggio Σ

Definiamo adesso la funzione `Convert(G)` che prende in input un grafo e restituisce l'espressione regolare associata.

Definiamo `Convert` :
- Definiamo $k =$ numero di stati in G.
- Se $k = 2$ significa che abbiamo soltanto $q_{\text{start}}, q_{\text{acc}}$ e un singolo arco con etichetta $R \in \mathcal{R}$. Avremo R come output.
- Se $k > 2$ scegliamo uno stato q_{rip} diverso da q_{start} e q_{acc} e definiamo $G' = \{Q', \Sigma, \delta', q_{\text{start}}, q_{\text{acc}}\}$ dove:
  - $Q' = Q \setminus \{q_{\text{rip}}\}$
  - $\delta' : Q' \setminus \{q_{\text{acc}}\} \times Q' \setminus \{q_{\text{start}}\} \to \mathcal{R}$

Adesso $\forall q_i \in Q' \setminus \{a_{\text{acc}}\}, q_j \in Q' \setminus \{q_{\text{start}}\}$ consideriamo l'automa:



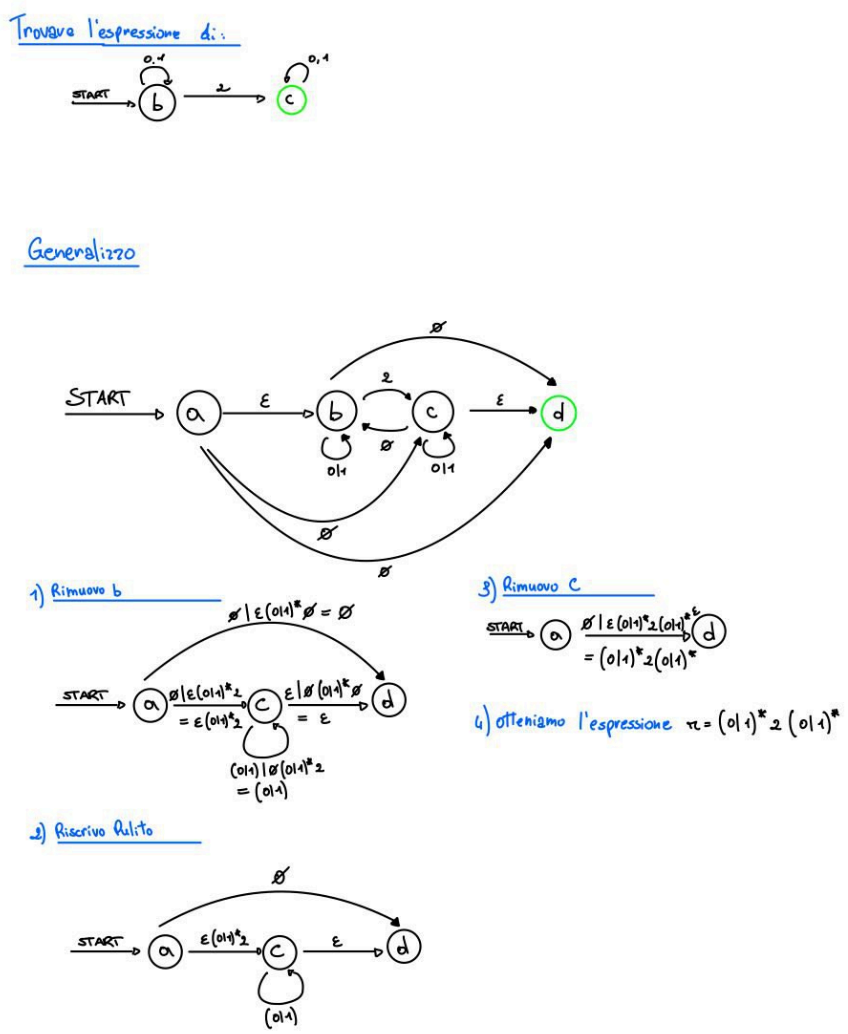
Dove:
- $\text{R1} = \delta(q_i, q_{\text{rip}})$
- $\text{R2} = \delta(q_{\text{rip}}, q_{\text{rip}})$
- $\text{R3} = \delta(q_j, q_{\text{rip}})$
- $\text{R4} = \delta(q_i, q_j)$

Consideriamo questo automa perché ci interessano soltanto gli archi che collegano q_i e q_j oppure che riguardano q_{rip}. Avremo quindi che

$$\delta'(q_i, q_j) = (\mathrm{R1})(\mathrm{R2})^*(\mathrm{R3}) \cup (\mathrm{R4})$$

Abbiamo quindi definito un automa che ci permette di muoverci fra q_i e q_j anche senza q_{rip}, dobbiamo continuare a ripetere questo procedimento finché non rimaniamo soltanto con lo stato iniziale e lo stato accettante.

Esempio



Quando avrò tempo lo farò più carino :P

Adesso possiamo concludere la dimostrazione iniziata nel capitolo precedente.

Dobbiamo dimostrare che quello che otteniamo da `Convert(G)` è equivalente a G.

Se $k = 2$ è sicuramente vero.

L'espressione regolare R descrive tutte le stringhe che portano, in G, da q_{start} a q_{acc}.

Supponiamo che sia vero per $k-1$ stati e dimostriamo che è vero per k stati mostrando che $L(G) = L(G')$ dove G' è l'automa con uno stato rimosso.

Se l'automa G accetta una stringa w significa che esiste un ramo di computazione che permette a G di percorrere gli stati $q_{\text{start}} ... q_{\text{acc}}$, se questa sequenza non contiene q_{rip} allora abbiamo che $L(G) = L(G')$ perché le nuove espressioni regolari conterranno le vecchie per unione.

Se invece q_{rip} è presente nella sequenza avremo comunque che gli stati a lui adiacenti (q_1, q_2) in G' hanno degli archi che tengono conto di tutti i modi per percorrere un cammino da q_1 a q_2 direttamente o passando per q_{rip} e quindi otteniamo di nuovo $L(G) = L(G')$.

7. Pumping Lemma

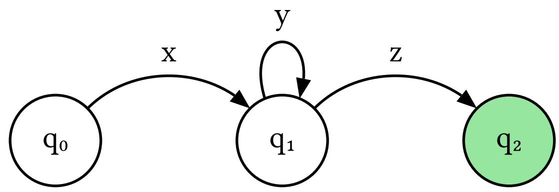
Serve a dimostare che un linguaggio non è regolare.

Teorema - Pumping Lemma

Se L è regolare, allora esiste p t.c. presa $w \in L$ con $|w| \geq p$, allora w può essere scomposta in $w = xyz$ in modo che:
1. $\forall i \geq 0$ si ha che $xy^i z \in L$
2. $|y| > 0$
3. $|xy| \leq p$

Vedremo che questo p è il numero di stati dell'automa, infatti prima di dimostrare ragionamo su questo caso:

Fissiamo $p =$#stati automa ed M t.c. $L(M) = L$ e siccome $|w| \geq p$, scomponiamo in questo modo:



Una ripetizione, in questo caso y, deve esistere sempre dato che la stringa è più grande del numero di stati. Significa appunto che uno stato deve sicuramente ripetersi.

7.1. Dimostrazione

Sia $M = (Q, \Sigma, \delta, q_1, F)$ t.c. $L(M) = L$ e sia $p = |Q|$. Consideriamo inoltre $w = w_1 w_2 ... w_n$ con $n \geq p$.

Consideriamo anche la sequenza di stati $r_1, ..., r_{n+1}$ attraversati da M su input w, avremo che $r_1 = q_1$ e $r_{n+1} \in F$. Ovviamente avremo che $n + 1 \geq p + 1$.

Per il **pigeonhole principle**, nella sequenza considerata ci sarà sicuramente uno stato che si ripete, sia questo stato r_j nella prima apparizione e r_l nella seconda ($j \neq l$, lo stato è lo stesso ma consideriamo due iterazioni diverse ovvero j quando lo incontriamo e l quando si ripete per la prima volta), avremo ovviamente che $l \leq p + 1$ perché r_l si presenta tra le prime $p + 1$ posizioni nella sequenza che inizia con r_1.

Scomponiamo la stringa in $w = xyz$ e poniamo:
- $x = w_1, ... w_{j-1}$. Ovvero la stringa prima del primo stato che si ripete.
- $y = w_j ... w_{l-1}$. Prima della prima ripetizioni.
- $z = w_l ... w_n$. Tutto il resto della stringa.

Abbiamo che:
- x porta M da $r_1 = q_1$ ad r_j
- y porta M da r_j ad $r_l = r_j$
- z porta M da $r_j = r_l$ a $r_{n+1} \in F$

Quindi notiamo che possiamo ripetere y quante volte vogliamo e la stringa ottenuta xy^iz apparterrà sempre al linguaggio. **Dimostrata la prima condizione**.

Siccome $j \neq l$ per costruzione allora $|y| > 0$. **Seconda condizione**.

Infine $l \leq p + 1$ e allora $|xy| = l - 1 \leq p$. **Terza condizione**

7.2. Esempi

Utilizziamo il pumping lemma.

1) Mostrare che $L = \{o^n 1^n : n \geq 0\}$ non è regolare.

Scegliamo una stringa $0^p 1^p$ con p che sarà il nostro **valore di pumping** che scegliamo per contraddire la prova. Vogliamo comunque $|w| \geq q$ per rientrare nelle condizioni.

Se il linguaggio fosse regolare allora presa $w = 0^p 1^p$, per qualsiasi scomposizione $w = xyz$ t.c. $|xy| \leq p$ avremo che y è composta da soli "0":

$$w = \underbrace{0...}_{\text{x}}\underbrace{...}_{\text{y}}01.....1$$

Per falsificare la condizione quindi ci basta prendere una $i \geq 2$ e avremo una stringa $xy^iz = o^q 1^p$ con $q > p$ che non rientra nel linguaggio dato che il numero di 0 ed 1 non è lo stesso.

2) Mostrare che il linguaggio $L = \{w \in \{0, 1\}^* : \#_0 w = \#_1 w\}$ ovvero le stringhe hanno lo stesso numero di 0 ed 1 ma in qualsiasi ordine.

Proviamo a scomporre con $w \in L$ t.c. $w = (01)^p$ e con $|w| = 2p \geq p$

Otteniamo una stringa:

$$\underbrace{01}_{\text{y}}\underbrace{0101010...01}_{\text{z}} \quad x = \varepsilon$$

Notiamo però che questa scomposizione non va bene per falsificare le condizioni, infatti qualsiasi i prendiamo aumentiamo sia il numero di 0 che di 1 quindi la stringa appartiene al linguaggio.

Proviamo con la stringa $w = 0^p 1^p$ con $|w| = 2p$ e rispettiamo $|xy| \leq p$ e $|y| > 0$.

Siccome $|xy| \leq p$ allora y è fatta solo da 0:

$$\underbrace{0....0}_{\text{x}}\underbrace{0.....0}_{\text{y}}\underbrace{1....1}_{\text{z}}$$

In questo caso aumentando y aumentiamo soltanto gli 0 e quindi la stringa non appartiene a L.

Più precisamente abbiamo che:

- $|y| = k > 0$ ma $k \leq p$ e inoltre $|x| = p - k$, $|z| = p$, tuttavia $|xy^2z| = (p - k) + 2k + p = 2p + k$ ma il numero di 0 è $(p - k) + 2k = p + k$ mentre quello degli 1 è sempre p che è $< p + k$, quindi non rientra nel linguaggio.

Con la stessa stringa possiamo provare anche questa scomposizione:

$$\underbrace{0.....}_{x}\ \underbrace{...00..}_{y}\underbrace{...01...1}_{z}$$

Anche in questo caso aumentiamo solo gli 0 e quindi non rientriamo nel linguaggio. Più precisamente:

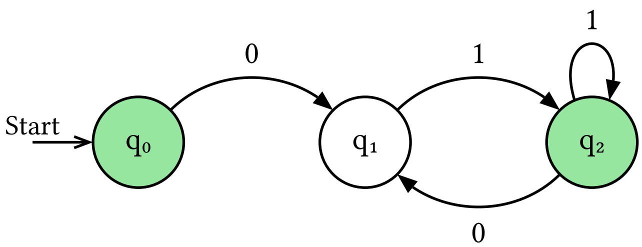
- $|y| = k > 0$
- $|z| = p + l$
- $|x| = p - k - l$
- Assumendo $l > 0$

Tuttavia $|xy^2z| = (p - k - l) + 2k + p + l = 2p + k$ ma il numero di 0 è $(p - k - l) + 2k + l = p + k$ mentre quello degli 1 è $p < p + k$.

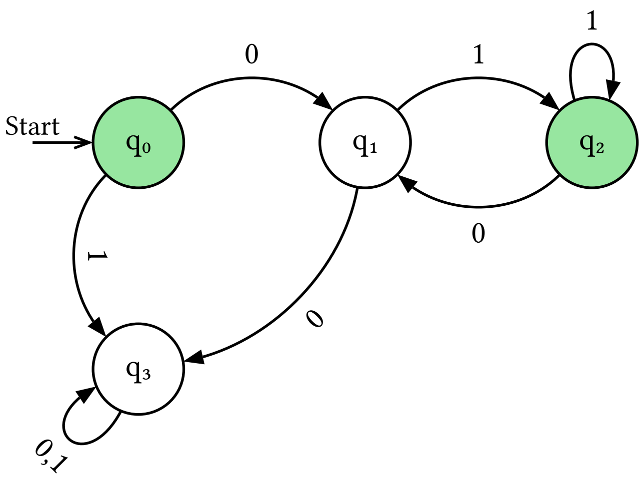
Esercizio

Costruire un DFA per il linguaggio $L = \{w \in \{0,1\}^* : w \notin (01^+)^*\}$. Realizzare dei DFA che non rispettano delle regole potrebbe essere complicato considerando solo questa richiesta. Un'idea però potrebbe essere quella di realizzare un DFA che rispetta quella regola e poi usare il complemento per ottenere un linguaggio, sempre regolare per la proprietà di chiusura, che rispetta la richiesta iniziale.

Costruiamo quindi un DFA che riconosce $(01^+)^*$:

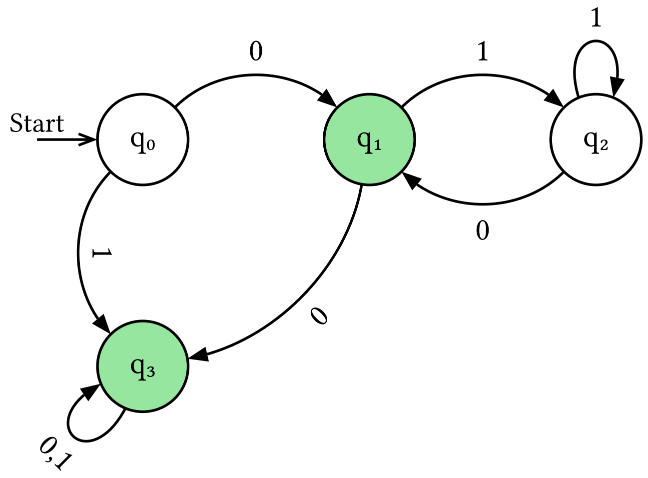


Per realizzare il complemento dobbiamo scambiare gli stati accettanti con quelli non accettanti, per non perdere delle stringhe però è importante **completare l'automa**:



Esercizio - continuo

A questo punto possiamo eseguire il complemento:



Abbiamo ottenuto l'automa che cercavamo inizialmente.

8. Grammatiche Acontestuali

Le grammatiche sono un mezzo di computazione utile in diverse applicazioni come ad esempio i compilatori, queste coincidono con un automa.

Ad esempio possiamo indicare come grammatica quella che genera le stringhe $0^n 1^n$ con $n \geq 0$

Una grammatica è una sequenza di **sostituzioni e produzioni**, facciamo un esempio di grammatica:

$$
\begin{gathered}
A \rightarrow 0A1 \\
A \rightarrow B \\
B \rightarrow \#
\end{gathered}
$$

Queste sono chiamate **regole della grammatica**; **A,B** sono **variabili** e **0,1,#** sono detti **terminali**. In ogni grammatica c'è sempre una variabile speciale ovvero la **variabile iniziale**.

Per costruire delle stringhe si parte dalla variabile iniziale e si applicano le regole come vogliamo, usando l'esempio di prima possiamo costruire:

$$A \rightarrow \text{0A1} \rightarrow \text{00A11} \rightarrow \text{000A111} \rightarrow \text{000B111} \rightarrow \text{000\#111}$$

Abbiamo applicato, in ordine, le regole **1, 1, 1, 2, 3**. Ad ogni produzione che effettuiamo possiamo associare un **albero sintattico**:

TODO: IMMAGINE

Definizione - CFG (Context-free grammar)

Una CFG è una tupla (V, Σ, R, S) dove:
- V è l'insieme finito delle variabili
- Σ è l'insieme finito dei terminali $(V \cap \Sigma = \emptyset)$
- R è l'insieme di regole
- $S \in V$ è la variabile iniziale

Se $u, v, w \in \Sigma \cup V$ e $(A \rightarrow w) \in R$ allora uAv produce uwv e lo scriviamo come $uAv \Rightarrow uwv$

Diciamo inoltre che u deriva v, con la notazione $u \overset{*}{\Rightarrow} v$ se:
- $u = v$

oppure
- $\exists u_1, ..., u_k$ con $k \geq 0$ t.c.:

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

Quindi se esiste una sequenza di sostituzione che ci permettono di arrivare a v partendo da u.

Questo ci permette di definire il linguaggio assocciato alla grammatica, ovvero:

$$\text{Sia } G = (V, \Sigma, R, S) \text{ allora } L(G) = \left\{ w \in \Sigma^* : S \overset{*}{\Rightarrow} w \right\}$$

Esempio

Sia $G = (V = \{S\}, \Sigma = \{a, b\}, R, S)$ e $R : S \to \text{asb}|\text{ss}|\varepsilon$.

Avremo che la stringa ab $\in L(G)$, ottenuta tramite la sequenza:

$$S \Rightarrow \text{asb} \Rightarrow \text{a}\varepsilon\text{b} \Rightarrow \text{ab}$$

Oppure anche la stringa aabb $\in L(G)$, con la sequenza:

$$S \Rightarrow \text{asb} \Rightarrow \text{aasbb} \Rightarrow \text{aa}\varepsilon\text{bb} = \text{aabb}$$

Vedremo delle tecniche per costruire delle grammatiche:
- Unendo grammatiche
- Passando da un DFA ad una grammatica
- Sfruttando la ricorsione

8.1. Unione di Grammatiche

Supponiamo di avere $G_i = (V_i, \Sigma_i, R_i, S_i)$ tutte CFG tali che $\forall i \in [1, 2, 3, ..., k]$ con $k \in \mathbb{N}$, vogliamo costruire una grammatica $G = (V, \Sigma, R, S)$ t.c. $L(G) = \cup_i L(G_i)$.

Lo facciamo costruendola in questo modo:
- $V = \cup_i V_i \cup \{S\}$, senza perdere generalità possiamo assumere che $V_i \cap U_j = \emptyset$ con $\forall i, j \in [1, 2, 3, ..., k]$ t.c. $i \neq j$
- S è la nuova variabile iniziale
- $\Sigma = \cup_j \Sigma_i$
- $R = \cup_j R_i \cup \{S \to S_1 \mid ... \mid S_k\}$

Dimostriamo che l'unione è anche essa una grammatica acontestuale, dobbiamo mostrare che $\cup_i L(G_i) = L(G)$ e quindi la doppia implicazione.

Prima parte - $\cup_i L(G_i) \subseteq L(G)$

Sia $w \in \cup_i L(G_i) : \exists j \in [k]$ t.c. $w \in L(G_j)$ ovvero esiste una sostituzione $S_j \underset{G_j}{\overset{*}{\Rightarrow}} w$ ma allora per definizione $S \Rightarrow S_j \underset{G_j}{\overset{*}{\Rightarrow}} w$ ovvero $w \in L(G)$

Seconda parte - $L(G) \subseteq \cup_i L(G_i)$

Sia $w \in L(G)$ ovvero $S \underset{G}{\overset{*}{\Rightarrow}} w$, per definizione $\exists j \in [k]$ t.c. $S \Rightarrow S_j \underset{G}{\overset{*}{\Rightarrow}} w$, siccome V_j è disgiunto da tutte le altre variabili abbiamo che $S_j \underset{G}{\overset{*}{\Rightarrow}} w$ implica che $w \in L(G_j) \cup_i L(G_i)$

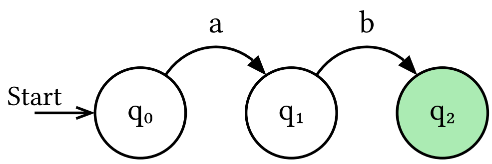
8.2. Da DFA a CFG

Dato un DFA $D = (Q, \Sigma, \delta, q_0, F)$ voglio definire $G = (V, \Sigma, R, S)$ t.c. $L(G) = L(D)$.

La costruiamo con:
- Σ rimane lo stesso
- $V = \{V_q : q \in Q\}$
- $S = V_{q_0}$
- Si aggiunge la regola $V_q \to aV_p$ per ogni $p, q \in Q, a \in \Sigma$ t.c. $\delta(q, a) = p$

- $\forall V_q \in F$ aggiungo $V_q \to \varepsilon$

Quindi creo una variabile della grammatica per ogni stato del DFA e i terminali sono le etichette tra uno stato e l'altro. *Esempio*:



Abbiamo quindi le regole:

$$S = V_{q_0} \to aV_{q_1}; V_{q_1} \to bV_{q_2}; V_{q_2} \to \varepsilon$$

La costruzione della stringa ab è:

$$S \Rightarrow aV_{q_0} \Rightarrow abV_{q_1} \Rightarrow ab$$

Ricorsione

Per ricorsione nelle gremmatiche intendiamo regole del tipo:

$$R \to 0R1$$

Regole di questo tipo sono utili per ricordare delle «informazioni limitate»

8.3. Forma Normale di Chomsky

Una CFG è in forma normale se ogni regola è del tipo:

$$A \to BC$$
$$A \to a$$

con ABC variabili ed a terminale inoltre $B, C \neq S$ ed è ammessa la regola $S \to \varepsilon$.

Teorema - Ogni CFG ammette una CGF equivalente in forma normale.

Dimostrazione.

Seguiamo delle regole per passare da una qualsiasi CFG alla sua forma normale:
1. Se la variabile iniziale compare a destra di una regola aggiungiamo una nuova variabile iniziale S_0 insieme alla regola $S_0 \to S$
2. Eliminare le ε-regole, se ad esempio abbiamo $A \to \varepsilon$ dobbiamo andare a controllare tutte le regole dove la A compare a destra e considerare che possiamo andare in ε aggiungendo quindi una nuova regola.
3. Elimino le regole unitarie $A \to B$, per ogni occorrenza $B \to u$ aggiungo $A \to u$ a meno che questa regola non è già stata eliminata.
4. Trasformare le regole restanti, se abbiamo $A \to u_1 u_2 ... u_k$ con $k \geq 3$ spezziamo la regola in

$$A \to u_1 A_1 \quad A_1 \to u_2 A_2 ... A_{k-2} \to u_{k-1} u_k$$

Dove A_i nuova variabile. Se u_i è terminale allora lo sostituisco con U_i e aggiungo la regola $U_i \to u_i$

Esempio:

Consideriamo una grammatica con le regole:
- $S \to ASA \mid aB$
- $A \to B \mid S$
- $B \to b \mid \varepsilon$

Usiamo le regole per passare in formale normale:
1. Aggiungiamo la variabile S_0 e la regola $S_0 \to S$
2. Eliminiamo le ε-regole, la prima è $B \to \varepsilon$ questo significa che una volta rimossa dobbiamo considerare che in ogni regola dove B sta a destra possiamo andare anche in ε, otteniamo quindi:

- $S_0 \to S$
- $S \to ASA \mid aB$
- $A \to B \mid S \mid \varepsilon$
- $B \to b$

Adesso eliminiamo $A \to \varepsilon$ e andiamo a controllare quelle dove la A compare a destra, otteniamo:
- $S_0 \to S$
- $S \to ASA \mid aB \mid SA \mid AS \mid S$
- $A \to B \mid S$
- $B \to b$

3. Eliminiamo le regole unitarie, partiamo da $S \to S, S_0 \to S$, ovvero sostituiamo la variabile a destra con la regola rimossa, otteniamo:

- $S_0 \to ASA \mid aB \mid SA \mid AS$
- $S \to ASA \mid aB \mid SA \mid AS$
- $A \to B \mid S$
- $B \to b$

Vanno rimosse anche $A \to B, A \to S$:
- $S_0 \to ASA \mid aB \mid SA \mid AS$
- $S \to ASA \mid aB \mid SA \mid AS$
- $A \to b \mid ASA \mid aB \mid SA \mid AS$
- $B \to b$

4. Finiamo le sostituzioni rimuovendo le regole che hanno a destra 3 o più variabili e quelle che hanno terminali e variabili insieme, in questo caso non vanno bene le regole che ci portano nella stringa *ASA* e in aB, creiamo quindi una variabile A_1 che ci porta in *SA* e sostituiamo questa variabile nelle regole, per le regole che ci portano in aB creiamo la variabile U:

- $S_0 \to AA_1 \mid UB \mid SA \mid AS$

- $S \to AA_1 \mid UB \mid SA \mid AS$
- $A \to b \mid AA_1 \mid UB \mid SA \mid AS$
- $B \to b$
- $U \to a$
- $A_1 \to SA$

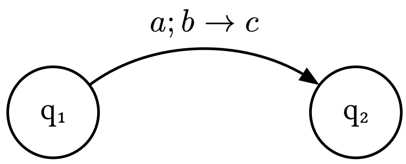
La grammatica è adesso in forma normale di Chomsky.

9. PDA (Push-Down Automata o Automi a Pila)

I PDA sono un'estensione dei DFA che riescono a riconoscere anche i linguaggi non regolari, loro infatti sono equivalenti alle CFG.

Possiamo vederli come degli NFA con associata una pila di tipo LIFO. Ad ogni passo di computazione il PDA può operare sulla cima della pila con diverse operazioni:
- Sostituzione del simbolo in cima
- PUSH ovvero inserimento di un simbolo in cima
- POP ovvero rimozione del simbolo in cima

Graficamente abbiamo che:



Il PDA sopra permette di spostarci da q_1 a q_2 se leggiamo a e se in cima alla pila abbiamo b, nel passaggio sostitusce b con c.

Sempre facendo riferimento all'esempio, questo significa che se leggiamo a ma sulla pila non abbiamo b allora non possiamo effettuare la transizione.

Osservazioni

Dato che il PDA avrà più rami di computazione per via del non determinismo è importante notare che ogni ramo avrà una sua pila.

Inoltre l'alfabeto della pila può anche essere diverso dall'alfabeto dell'automa.

La funzione di transizione è definita come:

$$\text{Dominio} : Q \times \Gamma_\varepsilon \times \Gamma_\varepsilon, \text{Immagine} : \mathcal{P}(Q \times \Gamma_\varepsilon)$$

Diamo una **definizione formale di PDA**.

Un PDA è una tupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$ dove Q, Σ, q_0, F sono come nei DFA / NFA mentre Γ è l'alfabeto usato dalla pila.

Cosa succede in una transizione? Prendiamo $(q, c) \in \delta(p, a, b)$, abbiamo diversi casi:
- Se $a, b, c \neq \varepsilon$ allora la transizione leggendo a ci porta dallo stato p, con b in cima alla pila, allo stato q con c in cima alla pila.
- Se $c \neq \varepsilon, b = \varepsilon$ e a in lettura allora fa PUSH di c. Ovvero abbiamo la siturazione $a; \varepsilon \to c$
- Se $c = \varepsilon, b \neq \varepsilon$ e a in lettura allora fa POP di b. Ovvero $a; b \to \varepsilon$

Notiamo quindi che le configurazioni di un PDA sono del tipo $Q \times \Sigma^* \times \Gamma^*$

Quando, i PDA, si trovano in uno stato di accettazione? Un PDA M accetta una stringa $w = w_1 ... w_m$ t.c. $w_i \in \Sigma$ se $\exists r_0, ..., r_m \in Q$ e stringhe $s_0, ..., s_m \in \Gamma^*$ t.c.:
- All'inizio $r_0 = q_0$ e $s_0 = \varepsilon$
- $r_m \in F$

- $\forall i = 0, ..., m-1$:
  - $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$
  - $s_i = at$ e $s_{i+1} = bt$ con $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$

Quindi partiamo con r_0 uguale allo stato iniziale e con lo stack s vuoto, ad ogni transizione passiamo da r_i a r_{i+1} se abbiamo in cima allo stack un carattere a e leggendo w_i, infatti avremo che lo stack ad s_i è uguale alla stringa composta da a ovvero l'ultimo carattere e t la stringa precedente, all'ultima transizione ovvero s_{i+1} avremo come carattere in cima b e poi t che comprende tutti gli altri caratteri.

Quindi possiamo mettere in relazione due configurazioni:

$$(p, ax, by) \underset{M}{\vdash} (q, x, cy) \text{ se e solo se } (q, c) \in \delta(p, a, b)$$

Con $a \in \Sigma; x \in \Sigma^*; a, b \in \Gamma, y \in \Gamma^*; p, q \in Q$

Come per i DFA e NFA possiamo estendere la chiusura con la chiusura simmetrica e transitiva:

$$L(M) = \left\{ w \in \Sigma^* : (q_0, w, \varepsilon) \overset{*}{\underset{M}{\vdash}} (q, \varepsilon, y) \quad q \in F, y \in \Gamma^* \right\}$$

Nota

Il PDA accetta **indipendentemente dal contenuto della pila** quindi, senza perdere generalità, possiamo assumere che la pila deve essere vuota.

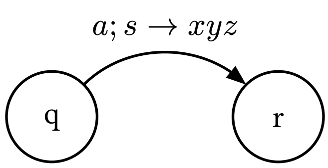
9.1. Corrispondenza tra PDA e CFG

Teorema

Un linguaggio è acontestuale se e solo se esiste un PDA che lo riconosce.

Lemma - Se L è acontestuale allora $\exists M \in$ PDA t.c. $L = L(M)$. Ovvero una delle due implicazioni del teorema.

Dimostrazione - Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, scriviamo le transizioni in questo modo:



Questa transizione implica che $(r, xyz) \in \delta(q, a, s)$.

Consideriamo la grammatica $G = (V, \Sigma, R, S)$.

Definiamo il PDA:
- $Q = \{q_{\text{START}}, q_{\text{LOOP}}, q_{\text{ACC}})\} \cup Q'$ dove Q' sono degli stati ausiliari che ci serviranno per definire δ.
- $\Gamma = V \cup \Sigma$
- $F = \{q_{\text{ACCEPT}}\}$

- Dato $q_{\text{START}} \in Q$, inseriamo $S\$$ nello stack, si ha che:

$$\delta(q_{\text{START}}, \varepsilon, \varepsilon) = \{(q_{\text{LOOP}}, S\$)\}$$

A questo punto, nello stato q_{LOOP} abbiamo diversi casi:
- Se la cima dello stack contiene una variabile $(A \in V)$ allora

$$\delta(q_{\text{LOOP}}, \varepsilon, A) = \{(q_{\text{LOOP}}, w) : A \to w \text{ con } w \in G\}$$

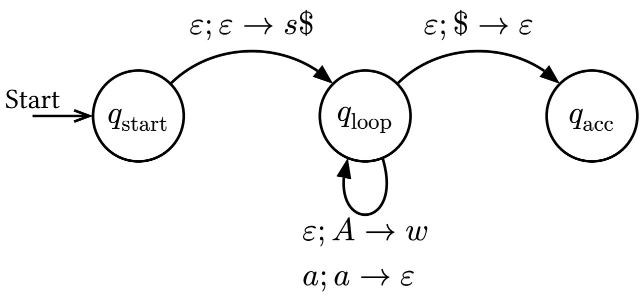
- Se la cima contiene un terminale $(a \in \Sigma)$ allora

$$\delta(q_{\text{LOOP}}, a, a) = \{(q_{\text{LOOP}}, \varepsilon)\}$$

- Se la cima contiene \$ allora

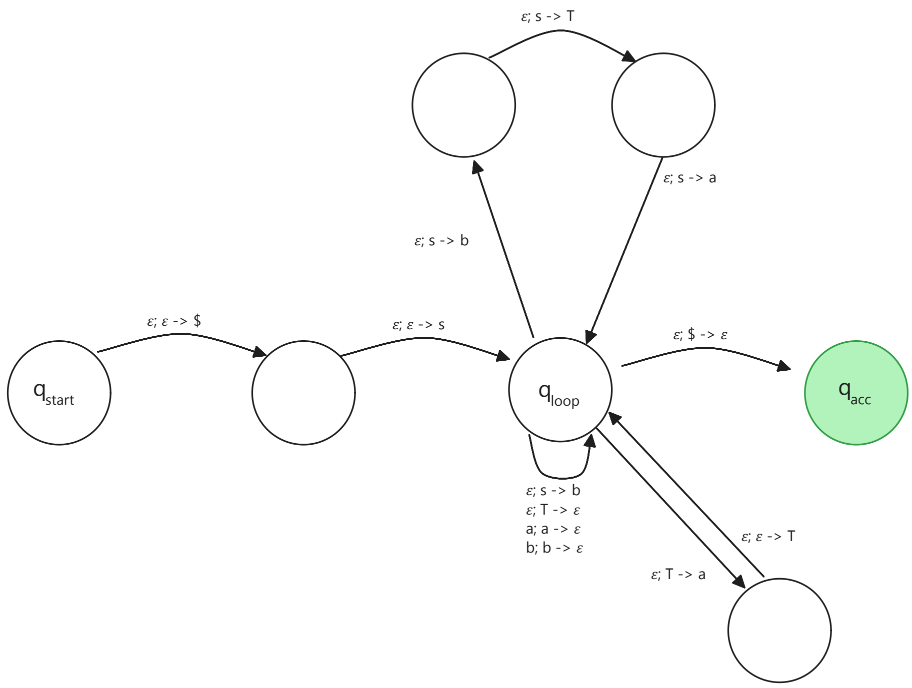
$$\delta(q_{\text{LOOP}}, \varepsilon, \$) = \{(q_{\text{ACC}}, \varepsilon)\}$$

Graficamente abbiamo che:



Esempio

Consideriamo la grammatica G con le regole $S \to aTb \mid b$ e $T \to Ta \mid \varepsilon$, costruire il PDA equivalente, usando le regole della dimostrazione precedente:



Dimostriamo quindi la seconda parte dell-implicazione.

Lemma - Se L è riconosciuto da un PDA M allora L è acontestuale.

Abbiamo un PDA P e vogliamo costruire una grammatica G che genera tutte le stringhe accettate da P. Generiamo una grammatica che per ciascuna coppia di stati $p, q \in P$ avrà una variabile A_{pq}, questa variabile genera tutte le stringhe che portano P da p con pila vuota a q con pila vuota. Da notare che queste stringhe possono portare P da p a q indpendentemente dal contenuto della pila ma lasciandola comunque nella stessa condizione di prima.

Forniamo a P delle caratteristiche per semplificare la dimsotrazione ma senza perdere di generalità:
- Ha un unico stato accettante p_{acc}
- Svuota lo stack prima di accettare
- Ciascuna transizione o fa un POP o fa un PUSH, non può farle entrambe

Per fornire l'ultima caratterstica al PDA dobbiamo sostituire tutte le transizione che sostituiscono simboli (POP e PUSH contemporaneamente) con una sequenza di transizioni, inoltre sostituiamo anche ogni transizione che non fa POP o PUSH con una sequenza di transizioni che inserisce ed elima un simbolo dalla pila.

Per qualsiasi stringa x che dobbiamo generare, l'automa P farà come prima azione un PUSH, dato che non può eliminare dalla pila vuota, l'ultima azione invece sarà sicuramente un POP perchè la pila deve essere vuota.

Durante questa computazione possono verificarsi due casi:

1. Il simbolo eliminato alla fine è lo stesso simbolo che è stato inserito all'inizio, in questo caso la pila potrebbe essere vuota soltanto all'inizio e alla fine della computazione su x.

Questa possibilità la gestiamo con la regola:

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

dove a è l'input della prima transizione e b nell'ultima, r è lo stato che segue p ed s è lo stato che precede q.

2. Il simbolo inserito all'inizio viene tolto durante la computazione ma non alla fine, la pila si svuota in quel punto.

Questa possibilità la simuliamo con la regola:

$$A_{pq} \to A_{pr}A_{rq}$$

Dimostrazione - Prendiamo $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{acc}}\})$ e assumiamo che sia nella forma di prima. La grammatica G sarà definita da $V = \left\{A_{pq} : p, q \in Q\right\}$ ed $S = A_{q_0 q_{\text{acc}}}$. Le regole sono:
- $\forall p, q, r, s \in Q; u \in \Gamma; a, b \in \Sigma_\varepsilon$
  - Se $(r, u) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, u)$ allora $A_{pq} \to aA_{rs}b$
- $\forall p, q \in Q; A_{pq} \to A_{pr}A_{rq}$
- $\forall p \in Q; A_{pp} \to \varepsilon$

Proviamo che questa costruzione funziona dimostrando che A_{pq} genera x se e solo se x porta P da p con pila vuota a q con pila vuota. Consideriamo la doppia implicazione.

Fatto 1 - Se A_{pq} genera x allora x può portare P da p con pila vuota a q con pila vuota. Dimostriamolo per induzione sul numero di passi nella derivazione di x da A_{pq}

- **Caso Base**: La derivazione è in un solo passo, in questo caso deve usare una regola dove nella parte destra non ci sono variabili, l'unica regola è $A_{pp} \to \varepsilon$ e ovviamente questa regola porta da p con pila vuota a p con pila vuota.
- **Induzione**: Assumiamo il fatto vero per k derivazioni con $k \geq 1$ e dimostriamo per $k+1$.

Supponiamo che $A_{pq} \overset{*}{\Rightarrow} x$ in $k+1$ passi, il primo passo può usare una delle due regole di prima, vediamo entrambi i casi:

- Caso $A_{pq} \to aA_{rs}b$: Dividiamo x in $x = ayb$ e consideriamo la parte y generata da A_{rs}. Dato che $A_{rs} \overset{*}{\Rightarrow} y$ in k passi, l'ipotesi induttiva ci dice che P può andare da r con la pila vuota a s con la pila vuota. Dato che $A_{pq} \to aA_{rs}b$ è una regola di G allora $(r, u) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, u)$ per qualche simbolo u della pila. Quindi se p inizia con pila vuota, legge a e può andare in r con u nella pila, leggendo y può andare in s e lasciare u sulla pila, poi può leggere b ed andare in q eliminando u.
- Caso $A_{pq} \to A_{pr}A_{rq}$: Consideriamo le parti di $x = yz$, dato che $A_{pr} \overset{*}{\Rightarrow} y$ in al più k passi e $A_{rq} \overset{*}{\Rightarrow} z$ in al più k passi, l'ipotesi induttiva ci dice che y può portare P da p ad r e z può portare P da r a q con la pila vuota all'inizio e alla fine.

Quindi x può portare P da p a q con la pila vuota.

Fatto 2 - Se x può portare P da p a q con la pila vuota allora A_{pq} genera x. Dimostriamo sempre per induzione sul numero di passi nella computazione da p a q con input x.

- **Caso Base**: La computazione ha 0 passi quindi inizia e termina nello stesso stato p. Dobbiamo mostrare che $A_{pp} \overset{*}{\Rightarrow} x$. P non può leggere alcun carattere in 0 passi quindi $x = \varepsilon$ e per costruzione di G abbiamo la regola $A_{pp} \to \varepsilon$ quindi il caso base è dimostrato
- **Passo Induttivo**: Assumiamo l'enunciato vero per computazioni lunghe $k \geq 0$, dimostriamo per $k+1$. Supponiamo ci sia una computazione in P dove x lo porta da p a q con pile vuote in $k+1$ passi, anche qui o la pila è vuota solo all'inizio e solo alla fine o si svuota in altri momenti.

Nel primo caso il simbolo inserito all'inizio deve essere quello rimosso alla fine, chiamiamo u questo simbolo e sia a il simbolo di input letto nella prima mossa e b quello letto nell'ultima, r lo stato dopo la prima mossa ed s lo stato prima dell'ultima allora $(r, u) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, u)$ e quindi la regola $A_{pq} \to aA_{rs}b$ è in G.

Consideriamo la parte y di $x = ayb$, l'input può portare P da r ad s senza toccare u e quindi far muovere P da r a s con una pila vuota su input y, abbiamo eliminato il primo e l'ultimo passo dei $k+1$ quindi abbiamo ottenuto $k-1$ passi, per ipotesi induttiva abbiamo $A_{rs} \overset{*}{\Rightarrow} y$ e quindi $A_{pq} \overset{*}{\Rightarrow} x$

Nel secondo caso sia r uno stato in cui si svuota la pila oltre alla fine o inizio, allora le computazioni da p a r e da r a q contengono al più k passi, sia y l'input letto nella prima parte e sia z l'input letto nella seconda, l'ipotesi induttiva ci dice che $A_{pr} \stackrel{*}{\Rightarrow} y$ e $A_{rq} \stackrel{*}{\Rightarrow} z$. Siccome la regola $A_{pq} \rightarrow A_{pr}A_{rq}$ è in G allora $A_{pq} \stackrel{*}{\Rightarrow} x$.

Possiamo dire quindi che **ogni linguaggio regolare è context-free**.

10. Pumping Lemma per i linguaggi CFL

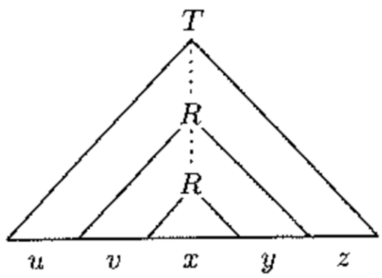
Segue 1:1 il libro in modo che me la studio da qui

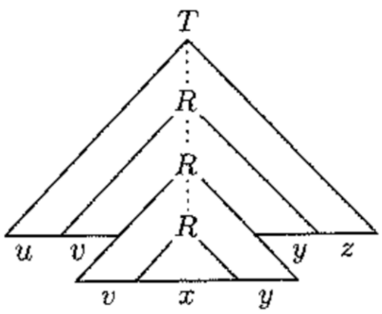
Se A è un linguaggio context-free allora esiste un numero p tale che, se s è una stringa in A di lunghezza almeno p allora può essere divisa in cinque parti $s = uvxyz$ che soddisfano:

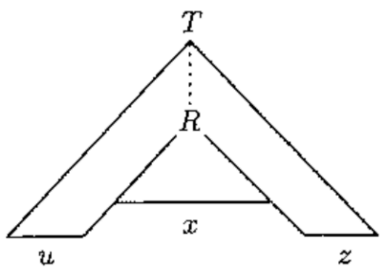
- $\forall i \geq 0, uv^i xy^i z \in A$
- $|vy| > 0$, serve per dire che entrambe non sono la stringa vuota altrimenti il teorema sarebbe banalmente vero.
- $|vxy| \leq p$, afferma che queste 3 parti hanno al più lunghezza p, utile per dimostrare che alcuni linguaggi non sono context-free.

Preso un CFL A e una CFG G che lo genera dobbiamo mostrare che ogni stringa sufficientemente lunga $s \in A$ può essere iterata e restare in A. Sia s una stringa molto lunga in A, essa è derivabile da G e quindi ha un albero sintattico.

Anche l'albero sintattico sarà molto lungo e ci deve essere un cammino dalla variabile alla radice a uno dei simboli terminali su una foglia. Per il principio della piccionaia qualche simbolo di variabile R si deve ripetere in questo cammino lungo. Questa ripetizione ci permette di sostituire il sottoalbero sotto la seconda occorrenza di R con il sottoalbero sotto la prima occorrenza di R e ottenere un albero sintattico consentito:







Questo significa che possiamo dividere la stringa in cinque parti $uvxyz$ e possiamo replicare il secondo e quarto pezzo e ottenere ancora una stringa nel linguaggio, formalmente $uv^i xy^i z \in A$ per ogni $i \geq 0$.

Vediamo la **dimostrazione**: Sia G una CFG per il CFL A e sia b il massimo numero di simboli nel lato destro di una regola (assumiamo che sia almeno 2), questo significa che in ogni albero sintattico di questa grammatica un nodo non può avere più di b figli, abbiamo quindi che:

- Ci sono al più b foglie in un passo dalla variabile iniziale.
- Ci sono al più b^2 foglie in 2 passi dalla variabile iniziale

- Ci sono al più b^h foglie in h passi dalla variabile iniziale.

Quindi se l'altezza dell'albero è al più h, la lunghezza della stringa generata è al più b^h. Viceversa se una stringa generata ha lunghezza maggiore o uguale a $b^h + 1$ allora ciascuno dei suoi alberi sintattici deve avere un'altezza maggiore o uguale a $h + 1$.

Sia V il numero delle variabili in G, poniamo p (lunghezza del pumping) uguale a $b^{|V+1}$. Ora se s è una stringa in A e la sua lunghezza è maggiore o uguale a p allora il suo albero sintattico deve avere altezza maggiore o uguale a $|V| + 1$ dato che $b^{|V|+1} \geq b^{|}V| + 1$.

Presa la stringa s e τ il suo albero sintattico che abbia il più piccolo numero di nodi, τ deve avere altezza maggiore o uguale a $|V| + 1$ quindi il suo cammino più lungo radice-foglia ha lunghezza almeno $|V| + 1$, questo cammino:
- Ha almeno $|V| + 2$ nodi
- Uno etichettato da un terminale
- Gli altri etichettati da variabili, almeno $|V + 1$

Siccome G ha solo $|V|$ variabili allora qualche variabile R è presente più volte su questo cammino, scegliamo una variabile che si ripete più in basso per comodità.

Dividiamo la stringa in cinque parti come nella figura precedente, ogni occorrenza di R ha un sottoalbero sotto essa che genera una parte della stringa s, l'occorrenza più in alto di R ha un sottoalbero più grande e genera vxy mentre quella più in basso soltanto x. Entrambi questi sottoalberi sono generati dalla stessa variabile, quindi possiamo sostituire l'uno con l'altro e ottenere comunque un albero corretto.

- Sostituire continuamente il più piccolo con il più grande fornisce gli alberi per le stringe $uv^i xy^i z$ per ogni $i > 1$
- Sostituire il più grande con il più piccolo genera la stringa uxz.

Questo dimostra la condizione 1.

Per ottenere la condizione 2 dobbiamo essere certi che $v, y \neq \varepsilon$. Se lo fossero, l'albero ottenuto sostituendo il più piccolo al più grande avrebbe meno nodi di τ e genererebbe ancora s, questo però non è possibile perché abbiamo scelto τ in modo che sia l'albero per s con meno nodi.

Per ottenere la condizione 3 dobbiamo essere sicuri che la lunghezza di vxy sia al più p. Nell'albero sintattico per s l'occorrenza più in alto di R genera vxy, abbiamo scelto R in modo che entrambe le occorrenze di essa cadano nelle $|V| + 1$ variabili più in basso del cammino e abbiamo scelto il più lungo cammino nell'albero sintattico, in modo che il sottoalbero in cui R genera vxy sia alto al più $|V| + 1$. Ma un albero con questa altezza può generare una stringa di lunghezza al più $b^{|V|+1} = p$.

Esempi

Mostrare che $B = \{a^n b^n c^n \mid n \geq 0\}$ non è context-free, dimostriamolo per assurdo quindi assumiamo che lo sia per poi arrivare ad una contraddizione.

Quindi sia B un CFL e p la lunghezza del pumping per B scegliamo la stringa $s = a^p b^p c^p$, $s \in B$ e $|s| \geq p$. Il lemma afferma che s può essere iterata ma noi mostreremo che non è vero, ovvero non importa come diviamo s in $uvxyz$ infatti violeremo sempre una delle tre condizioni del lemma.

Per prima cosa la condizione 2 stabilisce che v o y non sono vuote, consideriamo quindi i due casi dove le sottostringhe v, y contengano più di un tipo di simbolo dell'alfabeto o no:

1. Quando entrambe contengono solo un tipo di simbolo allora v non contiene entrambi i simboli a, b o entrambi i simboli b, c e lo stesso vale per y, questo significa che la stringa uv^2xy^2z non può contenere lo stesso numero di a, b, c e quindi non può essere un elemento di B. Abbiamo violato la condizione 1.

2. Quando v o y contengono più di un tipo di simbolo allora uv^2xy^2z può contenere un ugual numero dei tre siboli ma non nell'ordine corretto. Non appartiene quindi a B.

Uno di questi due casi deve verificarsi e poichè entrambi portano ad un assurdo la contraddizione si verifica sempre. B non è un CFL.

11. Macchine di Turing

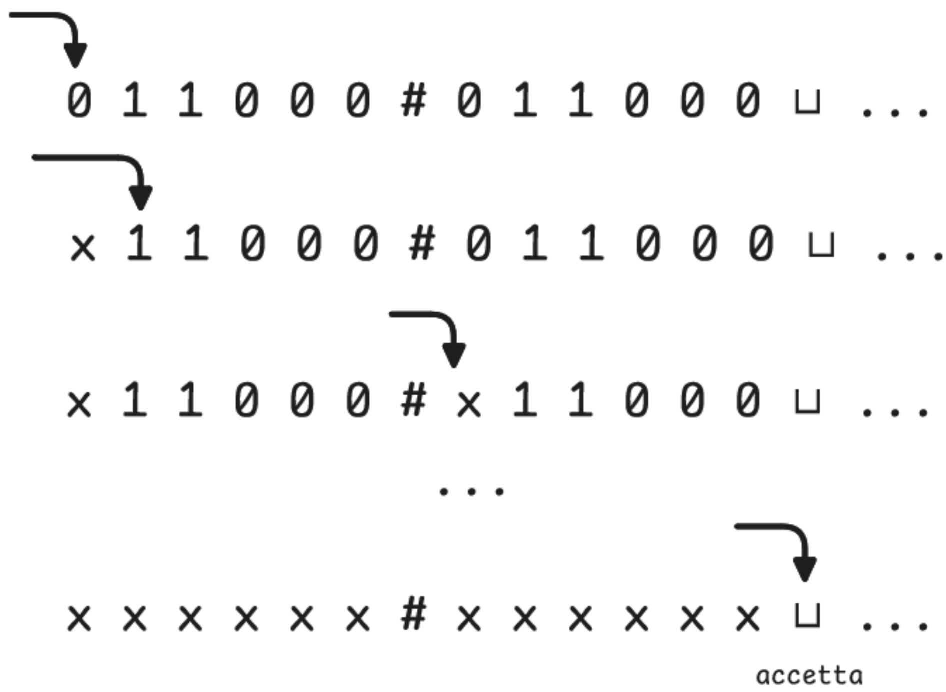
Sono simili agli automi ma con una memoria illimitata e senza restrizioni, sono dei modelli molto più precisi dei computer ma anche loro non sono in grado di risolvere tutti i tipi di problemi, quest'ultimi vanno oltre i limiti teorici della computazione. Come memoria utilizzano un nastro ed hanno una testina posizionata su questo nastro, inizialmente questo contiene soltanto la stringa in input, per scrivere e leggere informazioni sposta la testina. La macchina può raggiungere gli stati di accettazione o rifiuto ma potrebbe anche non raggiungerli andando avanti per sempre.

In breve

- Possono leggere e scrivere sul nastro
- Il nastro è infinito
- La testina si può muovere a destra e sinistra
- Gli stati di accettazione e rifiuto hanno effetto immediato

Proviamo a costrutire una macchina di Turing M_1 che testa l'appartenenza di una stringa al linguaggio $B = \{w\#w \mid w \in \{0,1\}^*\}$.

La macchina deve quindi capire se l'input è formato da due stringhe uguali separate da un #. L'idea è di leggere un carattere per stringa e vedere se sono uguali contrassegnandoli, se arriviamo fino alla fine allora siamo in B altrimenti appena troviamo un carattere che non combacia siamo in uno stato di riufiuto.



Questa immagina e la descrizione sopra però illustrano soltanto il funzionamento della macchina ma non tutti i dettagli, possiamo fornire delle descrizione formali analoghe a quelle degli automi finiti e a pila che specificano ogni parte della TM. In pratica però non vengono quasi mai fornite dato che tendono ad essere molto lunghe.

Definizione Formale

Una macchina di Turing è un 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ dove Q, Σ, Γ sono insiemi finiti e:

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

La funzione di transizione ci dice come la TM effettua un passo, se abbiamo:

$$\delta(q, a) = (r, b, L)$$

La TM si trova in un certo stato q e la testina punta alla casella contenente un simbolo a, la transizione avviene scrivendo il simbolo b al posto di a e passa allo stato r, la componente L o R indica se la testina si muove a sinistra o destra.

Come avviene la computazione? Inizialmente la TM riceve il suo input $w = w_1 w_2 ... w_n \in \Sigma^*$ sulle n celle più a sinistra del nastro, il resto del nastro è composto dal simbolo `blank`. La TM segue la funzione di transizione scrivendo e leggendo valori e spostando la testina, da notare che se ci troviamo ad un estremo e cerchiamo di spostare la testina oltre allora questa rimarrà ferma. La TM si ferma se raggiunge uno stato di accettazione o rifiuto altrimenti va avanti per sempre.

Configurazione di una Turing Machine

Per configurazione intendiamo l'impostazione di:

- Stato corrente
- Contenuto del nastro
- Posizione della testina

Per uno stato q e due stringhe u, v sull'alfebto Γ scriviamo uqv per indicare la configurazione dove q è lo stato dove ci troviamo mentre il contenuto del nastro è uv e la posizione attuale della testina è il primo simbolo di v.

Ad esempio la configurazione $1011q_7 01111$ rappresenta la configurazione dove il nastro contiene 101101111, lo stato corrente è q_7 e la testina è posizionata sul secondo 0

Formalizziamo il concetto di **computazione**: Si dice che la configurazione C_1 **produce** la configurazione C_2 se la TM può passare da C_1 a C_2 in un unico passo. Supponiamo di avere $a, b, c \in \Gamma$ e $u, v \in \Gamma^*$ e gli stati q_i, q_j e quindi $uaq_i bv$ e $uq_j acv$ sono due configurazioni. Diciamo che:

$$uaq_i bv \quad \text{produce} \quad uq_j acv$$

Se nella funzione di transizione abbiamo $\delta(q_i, b) = (q_j, c, L)$ nel caso di uno spostamento a sinistra, se invece avviene uno spostamento a destra diciamo che:

$$uaq_ibv \text{ produce } uacq_jv$$

se $\delta(q_i, b) = (q_j, c, R)$

Esiste il caso particolare dove la testina si trova ad una delle estremità, se siamo a sinistra q_ibv produce q_jcv se la transizione comporta una mossa a sinistra e produce cq_jv se comporta una mossa destra. Per l'estremità destra la configurazione uaq_i è equivalente a $uaq_i \sqcup$ e possiamo gestirla come prima ma invertita quindi se fa un movimento a destra non si muove mentre a sinistra si.

Definiamo, con input w:
- q_0w come **configurazione iniziale**
- **Configurazione di accettazione** quella dove lo stato è q_{accept}
- **Configurazione di rifiuto** quella dove lo stato è q_{reject}

Queste ultime due sono dette **configurazioni di arresto** e non producono ulteriori configurazioni.

L'insieme di stringhe che una TM M accetta rappresenta il **linguaggio di M** denotato con $L(M)$

Turing-Riconoscibile

Un linguaggio si dice **turing-riconoscibile** se esiste una macchina di Turing che lo riconosce.

Come detto prima le TM oltre a rifiutare potrebbero anche andare in loop per alcuni input, potrebbe risultare difficile in alcuni casi distinguere una macchina in loop da una che richiede molto tempo, preferiamo per questo delle TM che si fermano ad ogni input e che non ciclano mai, queste sono dette **decisore** perchè in ogni caso prende una decisione o di accettazione o di rifiuto. Un decisore **decide** un certo linguaggio se riconosce tale linguaggio.

Turing-Decidibile

Un linguaggio si dice **turing-decidibile** o decidibile se esiste una macchina di Turing che lo decide.

12. Macchina di Turing Multinastro

È come una normale macchina di Turing ma ha vari nastri ognuno con la propria testina, l'input si trova sul nastro 1 mentre tutti gli altri sono vuoti. Modifichiamo la funzione di transizione per garantire le operazioni su k nastri:

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

L'espressione:

$$\delta(q_i, a_1, ..., a_k) = (q_j, b_1, ..., b_k, L, R, ..., L)$$

Significa che se la macchina si trova in q_i e le testine da 1 a k leggono i simboli da a_1 ad a_k la macchina va in q_j e scrive i simboli da b_1 a b_k e muove ogni testina a sinistra, destra o la restare ferma.

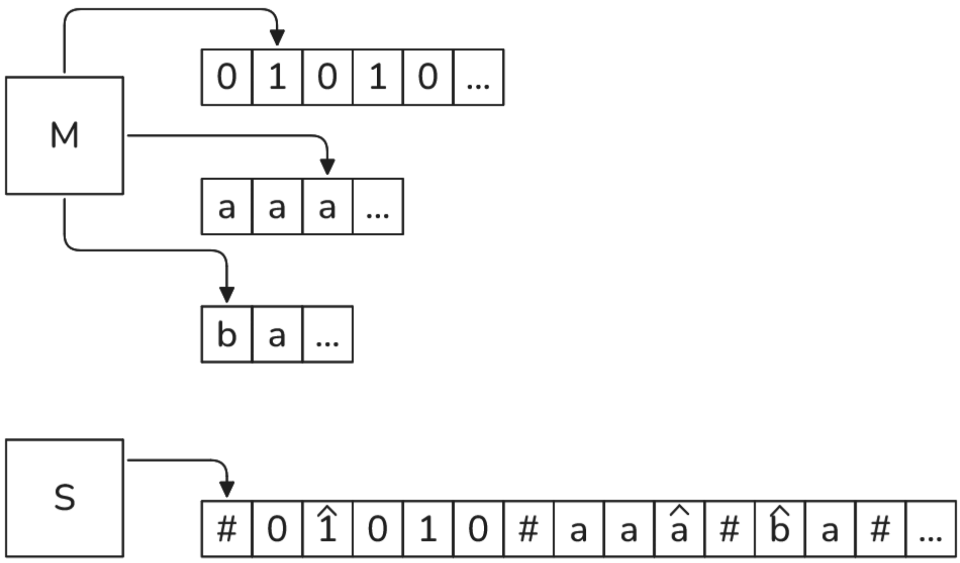
Queste macchine multinastro potrebbero sembrare più potenti della classiche macchine di Turing ma in realtà sono equivalenti, riconoscono quindi lo stesso linguaggio.

Teorema

Per ogni macchina di Turing multinastro esiste una macchina di Turing a nastro singolo equivalente.

Dimostrazione - Indichiamo con M una macchina multinastro e con S una a nastro singolo, vogliamo simulare M tramite S, quindi se M ha k nastri dobbiamo fare in modo di memorizzare tutte le informazioni di questi nastri su un singolo nastro, utilizziamo il simbolo $\#$ come delimitatore per separare i contenuti dei nastri. Dobbiamo però memorizzare anche la posizione delle testine e lo facciamo scrivendo $\hat{}$ sopra ai simboli puntati da una testina.

Esempio di simulazione di 3 nastri in un singolo nastro:



Corollario

Un linguaggio è Turing-Riconoscibile se e solo se qualche macchina di Turing multinastro lo riconosce.

Un linguaggio è Turing-Riconoscibile allora è riconosciuto da una macchina di Turing ordinaria che è un caso particolare di una multinastro.

L'altro verso del corollario, ovvero una macchina multinastro riconosce un linguaggio allora questo è turing riconosbile è dato dal teorema che ci dice che macchine multinastro e a singolo sono equivalenti.

12.1. Macchina di Turing non Deterministica

In qualsiasi punto della computazione la macchina può procedere effettuando più di una scelta, la funzione di transizione diventa:

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

La computazione quindi è un albero dove i rami corrispondono alle scelte effettuate, se un ramo porta ad uno stato di accettazione allora la macchina accetta.

Teorema

Per ogni macchina di Turing non deterministica esiste una macchina di Turing deterministica equivalente.

Idea Dimostrazione - L'idea è di simulare qualsiasi TM non deterministica N con una TM deterministica D ovvero provare tutte le possibili scelte che può fare N durante la computazione non deterministica, se D trova uno stato di accettazione su uno qualsiasi dei rami allora accetta.

Immaginiamo la computazione di N su un input w come un albero dove i rami rappresentano una scelta non deterministica e i nodi le configurazioni di N, la radice sarà quindi la configurazione iniziale. Dobbiamo quindi eseguire una ricerca su questo albero, la prima idea sarebbe una DFS ma in questo modo potrebbe finire ad esplorare un ramo infinito e non terminare mai più, dobbiamo utilizzare una BFS, una visita in ampiezza, in questo modo esploriamo prima tutti i rami alla stessa profondità.

Dimostrazione - Costruiamo una TM D con 3 nastri (sappiamo che è equivalente ad averlo soltanto uno):

- Il nastro 1 contiene la stringa in input e non viene mai modificato
- Il nastro 2 mantiene una copia del nastro di N corrispondente a qualche diramazione della sua computazione non determinisitca.
- Il nastro 3 tiene traccia della posizione di D nell'albero delle computazioni di N.

Consideriamo la rappresentazione dei dati sul terzo nastro. Ogni nodo può avere al massimo b figli dove b è la dimensione del più grande insieme di scelte possibili date dalla funzione di transizione di N. Assegniamo ad ogni nodo un indirizzo ovvero una stringa nell'alfabeto $\Gamma_b =$

$\{1, 2, ..., b\}$, assegneremo ad esempio l'indirizzo 231 al nodo a cui si arriva partendo dalla radice, spostandosi sul secondo figlio poi da lui al terzo figlio ed infine al primo figlio. **Ogni simbolo ci indica quindi il successivo figlio da considerare**.

Gli indirizzi sono **non validi** se un simbolo non corrisponde a nessuna scelta, invece una stringa vuota indica l'indirizzo della radice.

Descriviamo quindi D:
- Il nastro 1 contiene l'input w e i nastri 2 e 3 sono vuoti.
- Copia il nastro 1 sul nastro 2 ed inizializza la stringa sul nastro 3 a ε
- Utilizza il nastro 2 per simulare N con input w su una ramificazione della sua computazione non deterministica. Prima di ogni passo però consulta il simbolo successivo sul nastro 3 per determinare quale scelta fare tra quelle consentite dalla funzione di transizione di N. Se non rimangono simboli o se non sono validi interrompe il cammino andando alla fase successiva. Si va alla fase successiva anche in caso di rifiuto, se invece troviamo un caso accettante allora accettiamo.
- Sostituisce la stringa sul nastro 3 con la stringa successiva rispetto all'ordine delle stringhe, simula la ramificazione successiva della computazione di N andando al passo 2.ù

Corollario

Un linguaggio è Turing-Riconoscibile se e solo se esiste una macchina di Turing non deterministica che lo riconosce.

Corollario

Un linguaggio è decidibile se e solo se esiste una macchina di Turing non deterministica che lo decide.

Storia degli Algoritmi

Lo metto perché sembra interessante.

Inizialmente non c'era una definizione formale di algoritmo, si consideravano un insieme di istruzioni da eseguire per l'esecuzione di un compito. Nel 1900, il matematico **David Hilbert** identificò 23 problemi matematici e li pose come sfida per il secolo successivo, il decimo di questi problemi riguardava proprio gli algoritmi.

Il problema consiste nell'ideare un algoritmo per verificare se un polinomio abbia o meno una radice intera, lui non usò il termine *algoritmo* ma *«un processo in base al quale esso può essere determinato da un numero finito di operazioni»*.

Noi adesso sappiamo che non esiste un algoritmo per risolvere questo problema, ma al tempo per dimostrare che non esisteva c'era bisogno di una definizione formale di algoritmo, questa arrivò nel 1936 da **Alonzo Church** ed **Alan Turing**.

Church utilizzò un sistema di notazione chiamato λ-calcolo mentre Turing con le sue «macchine», è stato dimostrato che queste definizioni sono equivalenti e adesso la definizione è chiamata **Tesi di Church-Turing**.

Nel 1970 fu dimostrato che non esiste algoritmo capace di verifcare se un polinomio ha radici intere.

13. Decidibilità

Vediamo esempi di problemi relativi ad automi finiti.

Definiamo:

$$A_{\text{DFA}} = \{< B, w > \mid B \text{ è un DFA che accetta la stringa di input } w\}$$

Questo linguaggio contiene tutte le codifiche di tutti i DFA con le stringhe che essi accettano. Il problema di verificare se un DFA B accetta l'input w coincide con il problema di verificare se $< B, w >$ è un elemento del linguaggio A_{DFA}, mostrare che il linguaggio è decidibile equivale a mostrare che il problema computazione è decidibile.

Con il seguente teorema mostriamo che A_{DFA} è decidibile quindi stiamo anche mostrando che il problema di verificare se un dato automa finito accetta una determinata stringa è decidibile.

Teorema

A_{DFA} è un linguaggio decidibile.

Per dimostrarlo abbiamo bisogno di una TM M che decide A_{DFA}:

- M = «Su input $< B, w >$ dove B è un DFA e w è una stringa»:
  1. Simula B su input w.
  2. Se la simulazione termina in uno stato di accettazione, accetta altrimenti rifiuta.

La dimostrazione è semplicemente una simulazione di B su input w, se B termina in uno stato di accettazione allor M accetta altrimenti rifiuta.

Possiamo dimostrare un teorema simile anche per automi a stati finiti non deterministici:

$$A_{\text{NFA}} = \{< B, w > \mid B \text{ è un NFA che accetta la stringa di input } w\}$$

Teorema

A_{NFA} è un linguaggio decidibile.

Per dimostrarlo prendiamo una TM N che decide A_{NFA} e potremmo progettare N in modo che operi M ovvero simulando un NFA invece che un DFA. Per cambiare però progettiamo N in modo che usi M come sottoprocedura quindi N prima converte l'NFA in DFA e poi lo passa ad M dato che questa lavora sui DFA.

- N = «Su input $< B, w >$ dove B è un NFA e w è una stringa:»
  1. Converte l'NFA B in un DFA C equivalente
  2. Esegue la TM M su input $< C, w >$
  3. Se M accetta, accetta. Altrimenti rifiuta.

Infine, in modo analogo si può determinare se un'espressione regolare genera una determinata stringa:

$$A_{\text{REX}} = \{< R, w > | R \text{ è un'espressione regolare che genera la stringa } w\}$$

Teorema

A_{REX} è un linguaggio decidibile.

Per dimostrarlo progettiamo la TM P:

- P = «Su input $< R, w >$ dove R è un'espressione regolare e w è una stringa:»
  1. Converte l'espressione regolare R in un NFA A equivalente
  2. Esegue la TM N su input $< A, w >$
  3. Se N accetta, accetta. Altrimenti rifiuta.

Con questi 3 teoremi possiamo quindi dire che, ai fini della decidibilità, è equivalente presentare alla macchina di Turing un DFA, un NFA o un'espressione regolare perché la macchina è in grado di convertire una codifica nell'altra.

Affrontiamo adesso un problema diverso, il **test del vuoto** per il linguaggio di un automa finito. Consideriamo:

$$E_{\text{DFA}} = \{< A > \mid A \text{ è un DFA e } L(A) = \emptyset\}$$

Teorema

E_{DFA} è un linguaggio decidibile.

Dimostrazione - Il DFA accetta almeno una stringa se e solo se dallo stato iniziale può raggiungere uno stato di accettazione percorrendo il verso delle frecce del DFA. Per verificarlo progettiamo una TM T che utilizza un algoritmo di marcatura.

- T = «Su input $< A >$ dove A è un DFA»:
  1. Marca lo stato iniziale di A
  2. Ripete fino a quando non vengono più marcati nuovi stati:
     3. Marca qualsiasi stato che ha una transizione proveniente da uno stato già marcato
     4. Se nessuno stato di accettazione risulta marcato, accetta. Altrimenti rifiuta.

Un altro linguaggio deciidibile è:

$$EQ_{\text{DFA}} = \{< A, B > \mid A \text{e} B \text{ sono DFA e } L(A) = L(B)\}$$

Teorema

EQ_{DFA} è un linguaggio decidibile.

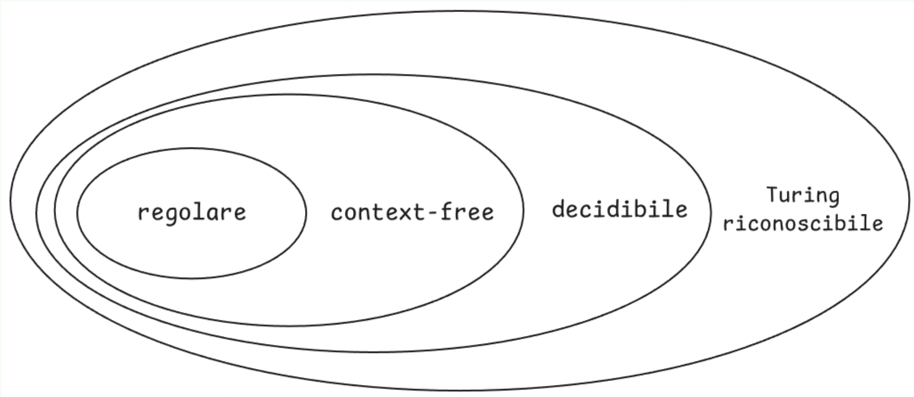
Dimostrazione - Costruiamo un nuovo DFA C a partire da A e B dove C accetta solo quelle stringhe che sono accettate da A o da B ma non da entrambi, quindi se A e B riconoscono lo stesso linguaggio allora C non accetterà nulla. Abbiamo quindi che:

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

Questa espressione è anche denotata come **differenza simmetrica** di $L(A)$ e $L(B)$, questa è utile perché $L(C) = \emptyset$ se e solo se $L(A) = L(B)$. Una volta costruito C possiamo utilizzare il teorema precedente per verificare se $L(C)$ è vuoto o meno, infatti se è vuoto $L(A) = L(B)$.

- F = «Su input $< A, B >$ dove A e B sono DFA»:
  1. Costruisce il DFA C
  2. Esegue la TM T del teorema precedente su input $< C >$
  3. Accetta se T accetta, altrimenti rifiuta.

Relazioni tra le classi di linguaggi



14. Indecidibilità

Vediamo il primo teorema che stabilisce l'indecidibilità di un linguaggio con il seguente problema: Determinare se una macchina di Turing accetta una determinata stringa in input, chiamiamo tale linguaggio A_{TM}:

$$A_{\text{TM}} = \{< M, w > \mid M \text{ è una TM e } M \text{ accetta } w\}$$

Teorema

A_{TM} è indecidibile.

Prima di dimostrarlo osserviamo che A_{TM} è **Turing-Riconoscibile**, questo teorema mostra quindi che i riconoscitori sono più potenti dei decisori infatti richiedere che una TM si fermi su ogni input limita le tipologie di linguaggi che possono essere riconosciuti. Costruiamo la TM U che riconosce A_{TM}:

- U = «Su input $< M, w >$ dove M è una TM e w è una stringa»:
  1. Simula M su input w
  2. Se durante la computazione M entra nello stato di accettazione, accetta. Altrimenti rifiuta.

Questa macchina cicla su $< M, w >$ se M cicla su w e questo è il motivo per cui non decide A_{TM} infatti se l'algoritmo avesse modo di determinare che M non si ferma su w allora sarebbe in grado di rifiutare w, dimostreremo che un algoritmo non ha modo di determinarlo.

Vogliamo dimostrare l'indecidibilità del linguaggio

$$A_{\text{TM}} = \{< M, w > \mid M \text{ è una TM ed } M \text{ accetta } w\}$$

Dimostrazione - Assumiamo che A_{TM} è decidibile e otteniamo una contraddizione. Supponiamo che H sia un decisore per A_{TM} e quindi su input $< M, w >$ dove M è una TM e w una stringa abbiamo che H si ferma ed accetta se M accetta w, si ferma e rifiuta se M non accetta w. Assumiamo quindi che H è una TM dove:

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

Adesso costruiamo una nuova TM D avente H come sottoprocedura, questa nuova TM chiama H per determinare cosa fa M quando l'input di M è la sua stessa descrizione $< M >$, una volta che D ha determinato questa informazione, fa il contrario, quindi rifiuta se M accetta e viceversa. Descriviamo D:

- D = «Su input $< M >$ dove M è una TM»:
  1. Esegue H su input $< M, < M \gg$
  2. Dà in output l'opposto di ciò che H dà in input, rifiuta se H accetta e viceversa.

Quindi:

$$D(< M >) = \begin{cases} \text{accetta} & \text{se } M \text{ non accetta } < M > \\ \text{rifiuta} & \text{se } M \text{ accetta } < M > \end{cases}$$

E cosa succede quando eseguiamo D con la sua stessa descrizione $< D >$?

$$D(< D >) = \begin{cases} \text{accetta} & \text{se } D \text{ non accetta } < D > \\ \text{rifiuta} & \text{se } D \text{ accetta } < D > \end{cases}$$

Quindi indipendentemente da ciò che fa D, essa è costretta a fare il contrario ed è quindi una contraddizione, nè la TM D nè la TM H possono esistere.

Si utilizza la tecnica della diagonalizzazione quando si esaminano le tavole del comportamento delle TM H, D, in queste tavole si indicizzano le righe con tutte le TM $M_1, M_2 ...$ e le colonne con le descrizioni $< M_1 >, < M_2 >, ...$ e le entrate dicono se la macchina di una determinata riga accetta l'input di una data colonna.

Se aggiungiamo D alla tabella, dato che deve comparire, cosa inseriamo quando D riceve come input $< D >$?

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	accetta	rifiuta	accetta	rifiuta		accetta	
M_2	accetta	accetta	accetta	accetta	$\cdots$	accetta	$\cdots$
M_3	rifiuta	rifiuta	rifiuta	rifiuta		rifiuta	
M_4	accetta	accetta	rifiuta	rifiuta		accetta	
$\vdots$		$\vdots$			$\ddots$		
D	rifiuta	rifiuta	accetta	accetta		?	
$\vdots$		$\vdots$					$\ddots$

Teorema

Un linguaggio è decidibile se e solo se è Turing-Riconoscibile e coTuring-Riconoscibile.

Un linguaggio è coTuring riconoscibile se $\overline{L}$ è Turing-Riconoscibile.

Corollario

Se L non è decidibile, allora almeno uno tra L ed $\overline{L}$ non è Turing-Riconoscibile.

Corollario

$\overline{A_{\text{TM}}}$ non è Turing-Riconoscibile

Dimostrazione - Sappiamo che A_{TM} è Turing-Riconoscibile, se lo fosse anche $\overline{A_{\text{TM}}}$ allora A_{TM} sarebbe decidibile. Sappiamo però, dai teoremi precedenti, che A_{TM} non è decidibile e quindi $\overline{A_{\text{TM}}}$ non può essere Turing-Riconoscibile.

15. Riducibilità

Una riduzione è un modo di convertire un problema in un altro problema in modo tale che una soluzione al secondo problema può essere usata per risolvere il primo problema.

15.1. Problemi indecidibili della Teoria dei Linguaggi:

Prima abbiamo visto l'indecidibilità di A_{TM}, ovvero il problema di determinare se una macchina di Turing accetta un dato input. Adesso consideriamo il problema HALT_{TM} ovvero quello di determinare se una macchina di Turing si ferma su un dato input, va bene sia un accettazione che un rifiuto. Questo è noto come il **problema della fermata**.

Utilizziamo l'indecidibilità di A_{TM} per dimostrare l'indecidibilità di HALT_{TM}, riducendo A_{TM} a HALT_{TM}.

Sia:

$$\text{HALT}_{\text{TM}} = \{< M, w > \mid M \text{ è una TM e } M \text{ si ferma su input } w\}$$

Teorema

HALT_{TM} è **indecidibile**.

Dimostriamo per assurdo il teorema, assumiamo HALT_{TM} decidibile ed utilizziamo questa assunzione per dimostrare che A_{TM} è decidibile, ottenendo quindi una contraddizione. Dobbiamo quindi mostrare che A_{TM} è riducibile da HALT_{TM}.

Supponiamo di avere una TM R che decide HALT_{TM}, utilizziamo R per verificare se M si ferma su w, se R indica che M non si ferma su w, rifiutiamo perché $< M, w >$ non è in A_{TM}; se invece R indica che M si ferma su w possiamo fare la simulazione senza rischiare di ciclare.

Se la TM R esistesse, potremmo decidere A_{TM} ma sappiamo che A_{TM} è indecidibile, abbiamo una contraddizione e possiamo quindi dire che R non esiste, quindi HALT_{TM} è indecidibile.

Dimostrazione - Ricapitolando, per ottenere una contraddizione assumiamo che la TM R decide HALT_{TM}, costruiamo la TM S per decidere A_{TM} che opera come segue:

- $S =$ «Su input $< M, w >$, una codifica di una TM M ed una stringa w»:
  1. Esegue la TM R su input $< M, w >$
  2. Se R rifiuta, rifiuta
  3. Se R accetta, simula M su w finché non si ferma
  4. Se M ha accettato, accetta altrimenti rifiuta.

Se R decide HALT_{TM} allora S decide A_{TM}, poiché A_{TM} è indecidibile allora anche HALT_{TM} deve essere indecidibile.

Vediamo altri teoremi e le loro dimostrazioni come esempi per utilizzare la riducibilità per dimostrare l'indecidibilità

Sia:

$$E_{\text{TM}} = \{< M > \mid M \text{ è una TM e } L(M) = \emptyset\}$$

Teorema

E_{TM} è **indecidibile**

Idea - Come prima assumiamo E_{TM} decidibile e mostriamo che A_{TM} è decidibile ottenendo quindi una contraddizione.

Sia R una TM che decide E_{TM}, utilizziamo R per costruire la TM S che decide A_{TM}, come deve agire S su input $< M, w >$?

S esegue R su input $< M >$ e vede se R accetta, se lo fa sappiamo che $L(M)$ è vuoto e quindi che M non accetta w. Se R rifiuta allora sappiamo che $L(M)$ non è vuoto e di conseguenza M accetta qualche stringa ma non sappiamo se w fa parte di queste.

Cambiamo approccio, invece di eseguire R su $< M >$, eseguiamo R su una modifica di $< M >$, ovvero modifichiamo M in modo da garantire che M rifiuta tutte le stringhe tranne w, ma su w funziona come al solito. Adesso possiamo utilizzare R per determinare se la macchina modificata riconosce il linguaggio vuoto. Infatti l'unica stringa che adesso la macchina può accettare è w, per cui il suo linguaggio sarà non vuoto se e solo se accetta w. Se R accetta quando riceve in input la descrizione della macchina modificata, sappiamo che la macchina modificata non accetta nulla e che M non accetta w.

Dimostrazione - Descriviamo la macchina modificata e chiamiamola M_1:

- M_1 = «Su input x»:
  1. Se $x \neq w$ rifiuta
  2. Se $x = w$ esegue M su input w e accetta se M accetta.

La macchina ha w come parte della sua descrizione, verifica se $x = w$ confrontando carattere per carattere con w per determinare se coincidono. Assumiamo che la TM R decide E_{TM} e costruiamo la TM S che decide A_{TM} nel seguente modo:

- S = «Su input $< M, w >$, una codifica di una TM M e una stringa w»:
  1. Usa la descrizione di M e w per costruire la TM M_1 vista sopra.
  2. Esegue R su input $< M_1 >$
  3. Se R accetta, rifiuta; se R rifiuta, accetta

Se R fosse un decisore per E_{TM} allora S lo sarebbe per A_{TM} ma questo non può esistere, quindi E_{TM} è indecidibile.

Un altro problema è quello di determinare se una macchina di Turing riconosce un linguaggio che può essere riconosciuto anche da un modello di calcolo più semplice. Ad esempio sia $\text{REGULAR}_{\text{TM}}$ il problema di determinare se una macchina di Turing ha un automa finito equivalente, che equivale a determinare se la TM riconosce un linguaggio regolare. Sia:

$$\text{REGULAR}_{\text{TM}} = \{< M > \mid M \text{ è una TM ed } L(M) \text{ è un linguaggio regolare}\}$$

Teorema

$\text{REGULAR}_{\text{TM}}$ è **indecidibile**

Idea - Anche in questo caso riduciamo da A_{TM}, assumiamo che $\text{REGULAR}_{\text{TM}}$ sia deciso da una TM R e costruiamo una TM S che decide A_{TM}.

L'idea è che S prenda il suo input $< M, w >$ e modifichi M in modo che la risultante TM riconosca un linguaggio regolare se e solo se M accetta w. Chiamiamo M_2 la macchina così modificata e progettiamo M_2 in modo che riconosca il linguaggio non regolare $\{0^n 1^n \mid n \geq 0\}$ se M non accetta w e riconosca il linguaggio regolare Σ^* se M accetta w. Va specificato come S può costruire una tale M_2 da M e w. La TM M_2 non è costruita con lo scopo di essere eseguita su qualche input ma soltanto per dare in input la sua descrizione al decisore per $\text{REGULAR}_{\text{TM}}$ che assumiamo esista. Quando il decisore da la sua risposta possiamo usarla per rispondere se M accetta w o meno, possiamo quindi decidere A_{TM}, contraddizione.

Dimostrazione - Definiamo R come una TM che decide $\text{REGULAR}_{\text{TM}}$ e costruiamo una TM S che decide A_{TM}, funzionamento di S:

- $S =$«Su input $< M, w >$ dove M è una TM e w è una stringa:»
  1. Costruisce la seguente TM M_2 $M_2 =$«Su input x»:
     1. Se x ha la forma $0^n 1^n$, accetta
     2. Se x non ha tale forma, esegue M su input w e accetta se M accetta w
  2. Esegue R su input $< M_2 >$
  3. Se R accetta, accetta altrimenti rifiuta

Un altro problema indecidibile è quello di dimostrare se due macchine di Turing sono equivalenti.

Sia:

$$\text{EQ}_{\text{TM}} = \{< M_1, M_2 > \mid M_1 \text{ ed } M_2 \text{ sono TM ed } L(M_1) = L(M_2)\}$$

Teorema

EQ_{TM} è **indecidibile**.

Dimostriamolo facendo una riduzione da E_{TM}.

Idea - Mostriamo che se EQ_{TM} fosse decidibile allora anche E_{TM} lo sarebbe ottenendo quindi una contraddizione. E_{TM} é il problema di determinare se il linguaggio di una TM é vuoto mentre EQ_{TM} quello di determinare se i linguaggi di due TM sono uguali, se uno di questi due é vuoto allora ci ritroviamo con il problema di determinare se il linguaggio dell'altra macchina é vuoto cioé E_{TM}

Dimostrazione - Consideriamo una TM R che decide EQ_{TM} e costruiamo la TM S che decide E_{TM}:

- $S =$ «Su input $< M >$, dove M é una TM»:
  - Esegue R su input $< M, M_1 >$ dove M_1 é una TM che rifiuta ogni input

  - Se R accetta, accetta altrimenti rifiuta

Se R decide $\mathrm{EQ}_{\mathrm{TM}}$, S decide E_{TM}. Ma sappiamo che E_{TM} é indecidibile quindi anche $\mathrm{EQ}_{\mathrm{TM}}$ é indecidibile.

15.2. Riducibilitá tramite funzione

Essere in grado di ridurre il problema A al problema B utilizzando una riduzione mediante funzione significa che esiste una funzione calcolabile che trasforma istanze del problema A in istanze del problema B. Questa funzione se esiste si chiama **riduzione** e ci permette di risolvere le istanze di A risolvendo le istanze di B.

Funzione Calcolabile

Una funzione $f : \Sigma^* \to \Sigma^*$ é una **funzione calcolabile** se esiste una macchina di Turing M che su qualsiasi input w si ferma avendolo solo $f(w)$ sul nastro.

Riducibilitá tramite Funzione

Un linguaggio A si dice **riducibile mediante funzione** al linguaggio B, e si denota con $A \leq_m B$, se esiste una funzione calcolabile $f : \Sigma^* \to \Sigma^*$ dove per ogni w:

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

La funzione f é chiamata **riduzione** da A a B.

Teorema

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

Dimostrazione - Siamo M il decisore per B ed f la riduzione da A a B. Descriviamo un decisore N per A come segue:
- $N =$ Su input w:
  - Computa $f(w)$
  - Esegue M su input $f(w)$ e restituisce lo stesso output di M

Quindi se $w \in A$ allora $f(w) \in B$ perché f é una riduzione da A a B. Quindi, M accetta $f(w)$ ogni volta che $w \in A$.

Corollario

Se $A \underset{m}{\leq} B$ e A è **indecibile** allora B è indecidibile.

Teorema

Se $A \leq_m B$ e B è **Turing-Riconoscibile** allora A è Turing-Riconoscibile.

Corollario

Se $A \leq_m B$ e A non è **Turing-Riconoscibile** allora B non è Turing-Riconoscibile.

Teorema

$\mathrm{EQ}_{\mathrm{TM}}$ non é né Turing-Riconoscibile né co-Turing riconoscibile

Dimostrazione - Per prima cosa dimostriamo che $\mathrm{EQ}_{\mathrm{TM}}$ non é Turing-Riconoscibile, lo facciamo dismotrando che A_{TM} é riducibile a $\overline{\mathrm{EQ}_{\mathrm{TM}}}$. La funzione di riduzione f opera nel seguente modo:

- $F =$ Su input $< M, w >$ dove M é una TM e w una stringa:
  - Costruisce le seguenti due macchine M_1, M_2
    - M_1 = Su ogni input Rifiuta
    - M_2 = Su ogni input eseuge M su w, se accetta, accetta
  - Restituisce $< M_1, M_2 >$

Quindi M_1 non accetta alcuna stringa e se M accetta w allora M_2 accetta qualsiasi stringa, le macchine non sono equivalenti. Viceversa se M non accetta w allora M_2 non accetta alcuna strina e le macchine sono equivalenti. Questo significa che f riduce A_{TM} a $\overline{\mathrm{EQ}_{\mathrm{TM}}}$. Adesso per dimostrare che $\overline{\mathrm{EQ}_{\mathrm{TM}}}$ non é Turing-Riconoscibile diamo una riduzione da A_{TM} a $\mathrm{EQ}_{\mathrm{TM}}$, in questo modo dimostriamo che $A_{\mathrm{TM}} \leq_m \mathrm{EQ}_{\mathrm{TM}}$. Costruiamo la TM G che calcola la riduzione g nel seguente modo:

- $G =$ Su input $< M, w >$ dove M é una TM e w é una stringa:
  - Costruisce le seguenti due macchine M_1, M_2:
    - M_1 accetta su ogni input
    - M_2 esegue M su w e se accetta, accetta.
  - Restituisce $< M_1, M_2 >$

La differenza tra f, g si trova in M_1, in f rifiuta sempre mentre in g rifiuta sempre mentre sia in f che g M accetta w se e solo se M_2 accetta ogni input, in g M accetta w se e solo se M_1 e M_2 sono equivalenti ed é per questo che g é una riduzione da A_{TM} a $\mathrm{EQ}_{\mathrm{TM}}$

16. Teoremi di Incompletezza di Godel

Uno dei più grandi problemi della matematica è stato quello di volerla formalizzare attraverso la logica in modo tale che fosse:

- **Consistente**: Non può generare contraddizioni
- **Completo**: In grado di dimostrare ogni affermazione vera.
- **Decidibile**: Esiste un algoritmo per determinare la verità o la falsità di ogni affermazione.

Questo è noto come **problema di Hilbert** ma in un solo giorno Godel dimostrò che non era possibile creare un sistema logico abbastanza complesso per contenere l'intera aritmetica, questo è noto come **primo teorema di incompletezza**.

In poco tempo Godel dimostrò anche il **secondo teorema di incompletezza** secondo cui nessun sistema consistente può dimostrare la propria consistenza. Dobbiamo quindi soltanto sperare che il sistema logico attualmente in uso, ZFC, sia consistente.

Le dimostrazioni fatte da Godel sono molto complesse ma grazie a Turing possiamo semplificarle, introduciamo un sistema di prova Π:

- Per ogni affermazione vera o falsa esiste una sua rappresentazione come stringa $< x >$ di lunghezza finita.
- Per ogni dimostrazione esiste una rappresentazione $< \Pi >$ come stringa di lunghezza finita.
- Esiste $V \in \text{TM}$ decisore t.c. $\forall (< x, w >) \in \Sigma^*$ si ha che $< x, w > \in L(V)$ se e solo se w è una dimostrazione per x in Π.

Un'affermazione $< x >$ è detta **dimostrabile** se $\exists < w > \in \Sigma^*$ t.c. $< x, w > \in L(V)$, è detta **indipendente** se né x né $\neg x$ sono dimostrabili in Π. Ma Π deve anche essere **computabile** ovvero gli assiomi devono essere riconoscibili, o meglio dato $A \in \Pi$ un assioma, A deve essere decidibile.

Definizione

Sia Π un sistema di prova. Diremo che:

- Π è **consistente** se $\forall < x >$ (per ogni affermazione) al più uno tra x e $\neg x$ è dimostrabile
- Π è **valido** se ogni affermazione vera è dimostrabile
- Π è **completo** se $\forall < x >$ almeno uno tra x e $\neg x$ è dimostrabile
- Π è **incompleto** se esiste un'affermazione indipendente

Osservazioni

- Se un sistema è valido è anche consistente infatti se x è dimostrabile allora è anche vera implicando che $\neg x$ è falsa.
- Un sistema consistente però non implica che sia valido
- Se un sistema è valido e completo allora $\forall < x >$ solo uno tra x e $\neg x$ è dimostrabile
- Se un sistema è consistente e completo allora $\forall < x >$ solo uno tra x e $\neg x$ è dimostrabile.

Quindi tutto e solo ciò che è vero è dimostrabile.

Teorema

Sia Π un sistema di dimostrazione abbastanza potente da comprendere l'aritmetica allora Π non può essere sia valido che completo

Dimostrazione - Sia $L = \{< x >: x \text{ è dimostrabile in } \Pi\}$ e definiamo la TM R_Π che prova a decidere L:

- R_Π = Data la stringa $< x >$ in input:
  - Ripeti per ogni $k = 1, 2, 3...$:
    - Ripeti per ogni $w \in \Sigma^*$ con $|w| = k$:
      - Se $V(< x, w >) = \text{ACC}$, R_Π accetta
      - Se $V(< x, w >) = \text{REJ}$, R_Π rifiuta

Quindi:
- Se Π è valido, quando R_Π termina fornisce una risposta corretta ma non è detto che termini.
- Se Π è completo allora R_Π è un decisore, infatti se $R_\Pi(< x >) = \text{loop}$ allora né x né $\neg x$ sono dimostrabili (non è completo).
- Se Π è valido e completo allora R_Π è un decisore tale che $\forall < x >$ se x è vera $R(< x >) = \text{ACC}$ e se x è falsa $R(< x >) = \text{REJ}$

Supponiamo per assurdo che Π sia valido e completo e quindi abbia un dimostratore in V. Data $M \in \text{TM}$ e y input considero la seguente affermazione:

$$\Phi_{M,y} = M(y) \text{ termina}$$

E definisco la TM $D(< M, w >) = R_\Pi\left(< \Phi_{M,y} >\right)$

Se Π è sia valido che completo allora R_Π è un decisore e di conseguenza lo è anche D ma $L(D) = \text{HALT}_{\text{TM}}$ che non è decidibile, contraddizione, R_Π non può essere sia valido che completo.

Primo Teorema di Incompletezza

Sia Π un sistema di dimostrazione abbastanza potente da comprendere l'aritmetica. Se Π è consistente allora Π non è completo.

Dimostrazione - Consideriamo la seguente affermazione $\Phi_M =$ "$M(< M >)$ termina" e definiamo la TM R_Π:

- R_Π = Data la stringa $< x >$ in input:
  - Ripeti per ogni $k = 1, 2, 3...$:
    - Ripeti per ogni $w \in \Sigma^*$ con $|w| = k$:
      - Se $V\left(< \Phi_{R_\Pi}, w >\right) = \text{ACC}$, R_Π va in loop
      - Se $V\left(< \neg\Phi_{R_\Pi}, w >\right) = \text{ACC}$, R_Π termina

Claim - Se Φ_{R_Π} o $\neg\Phi_{R_\Pi}$ è dimostrabile allora Π è incosistente.

Dimostrazione - Supponiamo che $\neg\Phi_{R_\Pi}$ sia dimostrabile, allora esiste una dimostrazione w t.c. $V\left(< \neg\Phi_{R_\Pi}, w >\right) = \text{ACC}$ implicando che $R_{\Pi(< R_\Pi >)}$ termina. Poiché l'esecuzione termina esiste una dimostrazione che descrive l'esecuzione di R_Π. (uguale a Φ_{R_Π}) Allo stesso modo, supponiamo che Φ_{R_Π} sia dimostrabile allora esiste una dimostrazione w t.c. $V\left(< \Phi_{R_\Pi}, w >\right) = \text{ACC}$ implicando che $R_{\Pi(< R_\Pi >)}$ va in loop. Siccome l'esecuzione va in loop, è noto il comportamento delal traccia, ed esiste dunque una dimostrazione che descrive l'esecuzione. Abbiamo dimostrato che ci basta dimostrare Φ_{R_Π} o $\neg\Phi_{R_\Pi}$ per rendere Π inconsistente.

Assumiamo Π consistente, allora per il claim né Φ_{R_Π} né $\neg\Phi_{R_\Pi}$ sono dimostrabili e dunque Π non è completo.

Secondo Teorema di Incompletezza

Sia Π un sistema di dimostrazione abbastanza potente da comprendere l'aritmetica. Se Π è consistente allora Π non può dimostrare l'affermazione «Π è consistente».

Dimostrazione - Sia Π un sistema definito dal teorema, allora tale sistema è in grado di descrivere il funzionamento di una TM. Sia R_Π definita tramite Φ_{R_Π} come nel teorema precedente.

Claim - Se Φ_{R_Π} o $\neg\Phi_{R_\Pi}$ è dimostrabile allora Π è inconsistente

Dimostrazione - Uguale al teorema precedente

Assumiamo che Π è consistente, allora per il claim né Φ_{R_Π} né $\neg\Phi_{R_\Pi}$ sono dimostrabili. Supponiamo ora per assurdo che l'affermazione «Π è consistente» sia dimostrabile tramite $w \in \Sigma^*$, allora unendo w alla dimsotrazione del claim otteniamo una dismotrazione per l'affermazione «né Φ_{R_Π} né $\neg\Phi_{R_\Pi}$ sono dimostrabili in Π». Tuttavia per costruzione di R_Π, ciò corrisponde anche una dimostrazione per $\neg\Phi_{R_\Pi}$ in quanto è garantito che la TM va in loop creando una contraddizione. «Π è consistente» non è dimostrabile in Π.

17. Complessitá di Tempo

Anche quando un problema é decidibile, e quindi risolvibile, potrebbe non esserlo nella pratica se la soluzione richiede una quantitá enorme di tempo o memoria.

Consideriamo il linguaggio $A = \{0^k 1^k \mid k \geq 0\}$, A é un linguaggio decidibile ma di quanto tempo necessita una TM a nastro singolo per decidere A? Esaminiamo la seguente TM M_1 e contiamo il numero di passi necessari:

- M_1 su input w:
  1. Scandisce il nastro e rifiuta se trova uno 0 a destra di un 1.
  2. Ripete se il nastro contiene almeno uno 0 ed almeno un 1.
  3. Scandisce il nastro, cancellando uno 0 ed un 1.
  4. Se rimane almeno uno 0 dopo che ogni simbolo 1 é stato cancellato o se rimane almeno un 1 dopo che ogni simbolo 0 é stato cancellato, rifiuta. Altrimenti se non rimangono né simboli 0 né simboli 1, accetta.

Introduciamo la terminologia necessaria per analizzare i tempi. Il numero di passi che un algoritmo utilizza puó dipendere da diversi parametri, per semplicitá consideriamo la lunghezza della stringa in input. Analizzeremo il **caso peggiore** ovvero il tempo di esecuzione massimo e il **caso medio** ovvero la media di tutti i tempi su input di una determinata lunghezza.

Definizione

Sia M una TM deterministica che si ferma su tutti gli input. Il tempo di esecuzione di M é la funzione:

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

Dove $f(n)$ é il numero massimo di passi che M utilizza su un qualsiasi input di lunghezza n. Se $f(n)$ é il tempo di esecuzione di M diciamo che M ha un tempo di esecuzione $f(n)$ e che M é una macchina di Turing di tempo $f(n)$.

17.1. Notazione O-grande ed o-piccola