"Sapienza" Università di Roma
Ingegneria dell'Informazione,
Informatica e Statistica
Dipartimento di Informatica

Automi: Calcolabilità e Complessità

Appunti integrati con il libro "Introduzione alla teoria della computazione", Michael Sipser

Author
Alessio Bandiera

14 dicembre 2023

Indice

Informazioni e Contatti

Prerequisiti consigliati:

- TODO: DA DECIDERE

Segnalazione errori ed eventuali migliorie:

Per segnalare eventuali errori e/o migliorie possibili, si prega di utilizzare il **sistema di Issues fornito da GitHub** all'interno della pagina della repository stessa contenente questi ed altri appunti (link fornito al di sotto), utilizzando uno dei template già forniti compilando direttamente i campi richiesti.

Gli appunti sono in continuo aggiornamento, pertanto, previa segnalazione, si prega di controllare se l'errore sia ancora presente nella versione più recente.

Licenza di distribuzione:

Contatti dell'autore e ulteriori link:

- Github: **https://github.com/ph04**
- Email: **alessio.bandiera02@gmail.com**
- LinkedIn: **Alessio Bandiera**

1

Linguaggi ed espressioni regolari

1.1 Stringhe e linguaggi

1.1.1 Stringhe

Definizione 1.1.1.1: Alfabeto

Si definisce **alfabeto** un qualsiasi insieme finito, non vuoto; i suoi elementi sono detti **simboli** o **caratteri**.

Esempio 1.1.1.1 (Alfabeto)**.** $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ è un alfabeto, composto da 5 simboli.

Definizione 1.1.1.2: Stringa

Sia Σ un alfabeto; una **stringa su** Σ è una sequenza finita di simboli di Σ; la **stringa vuota** è denotata con ε.

- Data una stringa w di Σ, allora $|w|$ è la lunghezza di w.
- Se w ha lunghezza $n \in \mathbb{N}$, allora è possibile scrivere che $w = w_1 w_2 \cdots w_n$ con $w_i \in \Sigma$ e $i \in [1, n]$.

Esempio 1.1.1.2 (Stringa)**.** Sia $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ un alfabeto; allora una sua possibile stringa è $w = \mathtt{x1y0z}$.

Definizione 1.1.1.3: Stringa inversa

Sia Σ un alfabeto, e $w = w_1 w_2 \cdots w_n$ una sua stringa; allora si definisce l'**inversa** di w come segue:

$$w^{\mathcal{R}} := w_n w_{n-1} \cdots w_1$$

Definizione 1.1.1.4: Concatenazione

Sia Σ un alfabeto, e $x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_n$ due sue stringhe; allora xy è la stringa ottenuta attraverso la **concatenazione** di x ed y.

Per indicare una stringa concatenata con se stessa k volte, si utilizza la notazione

$$x^k = \underbrace{xx \cdots x}_{k \text{ volte}}$$

Si noti che per ogni stringa x su Σ, si ha che $x^0 = \varepsilon$.

Definizione 1.1.1.5: Prefisso

Sia Σ un alfabeto, ed x, y due sue stringhe; allora x è detto essere un **prefisso** di y, se $\exists z \mid xz = y$, con z stringa in Σ.

Esempio 1.1.1.3 (Prefisso)**.** Sia $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ un alfabeto; allora la stringa $x = \mathtt{ab}$ è prefisso della stringa $y = \mathtt{abc}$, poiché esiste una stringa $z = \mathtt{c}$ tale per cui $xz = y$.

1.1.2 Linguaggi

Definizione 1.1.2.1: Linguaggio

Sia Σ un alfabeto; si definisce **linguaggio** un insieme di stringhe di Σ. Un linguaggio è detto **prefisso**, se nessun suo elemento è prefisso di un altro. Il linguaggio vuoto si indica con $\emptyset$.

Esempio 1.1.2.1 (Linguaggio binario)**.** Il lingauggio binario, che verrà utilizzato estensivamente, è il seguente:

$$\Sigma = \{\mathtt{0}, \mathtt{1}\}$$

1.1.3 Funzioni di Hamming

Definizione 1.1.3.1: Distanza di Hamming

Sia Σ un alfabeto, e siano x, y due sue stringhe tali che $|x| = |y|$; si definisce **distanza di Hamming** tra x ed y il numero di caratteri per cui x ed y differiscono. In simboli, date due stringhe $x = x_1 \cdots x_n, y = y_1 \cdots y_n$ con $n \in \mathbb{N}$, si ha che

$$d_H(x, y) := |\{i \in [1, n] \mid x_i \neq y_i\}|$$

Esempio 1.1.3.1 (Distanza di Hamming)**.** Siano $x = \mathtt{1011101}$ ed $y = \mathtt{1001001}$ due stringhe sull'alfabeto $\Sigma = \{\mathtt{0}, \mathtt{1}\}$; poiché differiscono per 2 caratteri, si ha che $d_H(x, y) = 2$.

Definizione 1.1.3.2: Peso di Hamming

Sia $\Sigma = \{\texttt{0}, \dots, \texttt{9}\}$ l'alfabeto composto dalle 10 cifre decimali, e sia x una sua stringa; si definisce **peso di Hamming** di x il numero di elementi di x diversi da $\texttt{0}$. In simboli, data una stringha $x = x_1 \cdots x_n$, con $n \in \mathbb{N}$, si ha che

$$w_H(x) := |\{i \in [1, n] \mid x_i \neq \texttt{0}\}|$$

Osservazione 1.1.3.1: Peso di Hamming di stringhe binarie

Sia $\Sigma = \{\texttt{0}, \texttt{1}\}$ l'alfabeto binario; allora, il peso di Hamming di una sua stringa è il numero di $\texttt{1}$ che la compongono.

1.2 Determinismo

1.2.1 Definizioni

Definizione 1.2.1.1: DFA

Un **DFA** (*Deterministic Finite Automaton*) è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove

- Q è l'**insieme degli stati** dell'automa, un insieme *finito*
- Σ è l'**alfabeto dell'automa**, un insieme *finito*
- $\delta : Q \times \Sigma \to Q$ è la **funzione di transizione**, che definisce la relazione tra gli stati
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'**insieme degli stati accettanti**, sui quali le stringhe possono terminare

Esempio 1.2.1.1 (DFA)**.** Un esempio di DFA è il seguente:

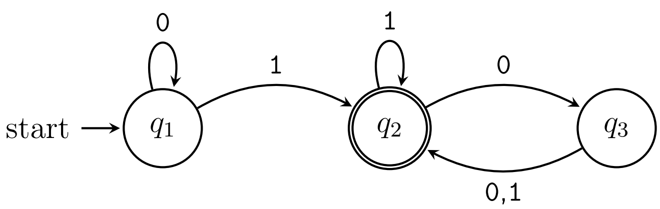


Figura 1.1: Un DFA.

esso può essere descritto secondo la quintupla $(Q, \Sigma, \delta, q_0, F)$ come segue:

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

- δ è la seguente:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- q_1 è lo stato iniziale
- $F = \{q_2\} \subseteq Q$

Definizione 1.2.1.2: Stringhe accettate (DFA)

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA, e sia $w = w_1 \cdots w_n$ una stringa tale per cui $\forall i \in [1, n] \quad w_i \in \Sigma$; allora, M **accetta** w se esiste una sequenza di stati $r_0, \ldots, r_n \in Q$ tali per cui

- $r_0 = q_0$
- $\forall i \in [0, n-1] \quad \delta(r_i, w_{i+1}) = r_{i+1}$
- $r_n \in F$

Definizione 1.2.1.3: Linguaggio di un automa

Sia M un automa; allora il **linguaggio di** M è un insieme $L(M)$ contenente tutte le stringhe accettate da M; simmetricamente, si dice che M **riconosce** $L(M)$.

Esempio 1.2.1.2 (Linguaggio di un automa)**.** Si consideri il seguente automa M_1:

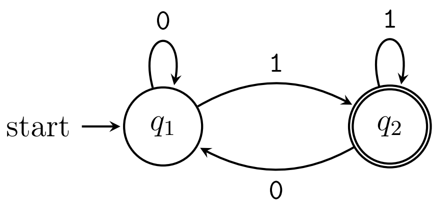


Figura 1.2: Un automa M_1.

sapendo che $\Sigma = \{0, 1\}$, che q_1 è lo stato iniziale, e che $F = \{q_2\}$, è facilmente verificabile che

$$L(M_1) = \{w \mid w = w_1 \cdots w_{n-1} 1, n \in \mathbb{N}\}$$

ovvero, M_1 accetta tutte e sole le stringhe che terminano per 1.

Definizione 1.2.1.4: Linguaggi riconosciuti da automi

Dato un linguaggio A, ed un automa M, si dice che M **riconosce** A se e solo se

$$A = \{w \mid M \text{ accetta } w\}$$

Definizione 1.2.1.5: Linguaggi di una classe di automi

Sia $\mathcal{C}$ una classe di automi; allora, l'**insieme dei linguaggi** riconisciuti dagli automi della classe $\mathcal{C}$ è denotato col seguente simbolismo:

$$L(\mathcal{C}) := \{L \mid \exists M \in \mathcal{C} : L(M) = L\}$$

dove L è un linguaggio, ed M è un automa della classe $\mathcal{C}$.

1.2.2 Linguaggi regolari

Definizione 1.2.2.1: Linguaggio regolare

Un linguaggio è detto **regolare** se e solo se esiste un DFA che lo riconosce. La classe dei linguaggi regolari è denotata con REG. Allora, in simboli, si ha che

$$\mathsf{REG} := \mathcal{L}(\mathsf{DFA})$$

Esempio 1.2.2.1 (Linguaggi regolari)**.** Sia $\Sigma = \{0, 1\}$ l'alfabeto binario, ed L il seguente linguaggio:

$$L := \{w \mid w = 0^n1, n \in \mathbb{N} - \{0\}\}$$

Tale linguaggio è regolare, poiché esiste il seguente DFA che lo riconosce:

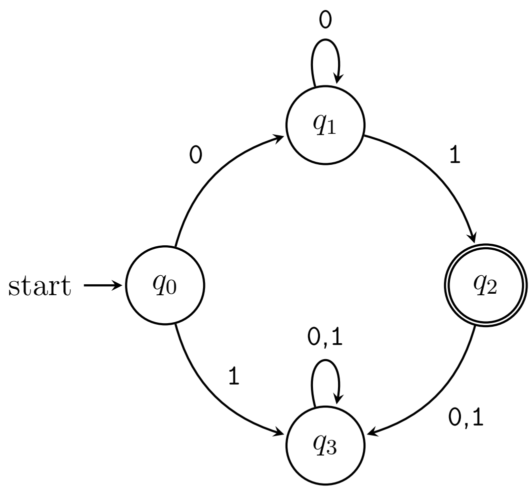


Figura 1.3: Un DFA che riconosce L.

1.3 Non determinismo

1.3.1 Definizioni

Definizione 1.3.1.1: NFA

Un **NFA** (*Nondeterministic Finite Automaton*) è un automa in cui possono esistere varie scelte per lo stato successivo in ogni punto. Durante la computazione, ogni volta che viene incontrata una scelta, la macchina si *divide*, e ognuno dei vari automi risultanti computa le varie scelte indipendentemente.

Formalmente, un NFA è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove

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

dove $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$.

Se il simbolo di input successivo non compare su alcuno degli archi uscenti dallo stato occupato da una copia della macchina, quella copia cessa di proseguire; inoltre, se *una qualunque copia* della macchina è in uno stato accettante, l'NFA accetta la stringa di input. Si noti che questa divisione è descritta dall'insieme potenza $\mathcal{P}(Q)$, poiché da ogni stato si può arrivare ad un *insieme* di stati.

Si noti che il determinismo è un caso particolare di non determinismo, dunque un DFA è sempre anche un NFA; in simboli DFA $\subseteq$ NFA.

Esempio 1.3.1.1 (NFA)**.** Un esempio di NFA è il seguente:

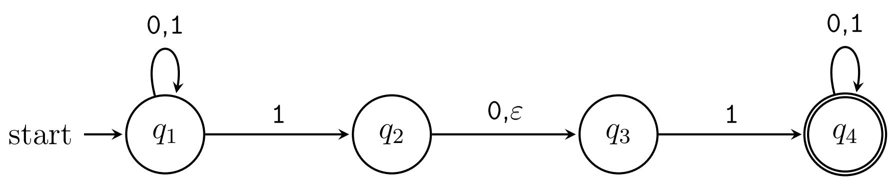


Figura 1.4: L'NFA N.

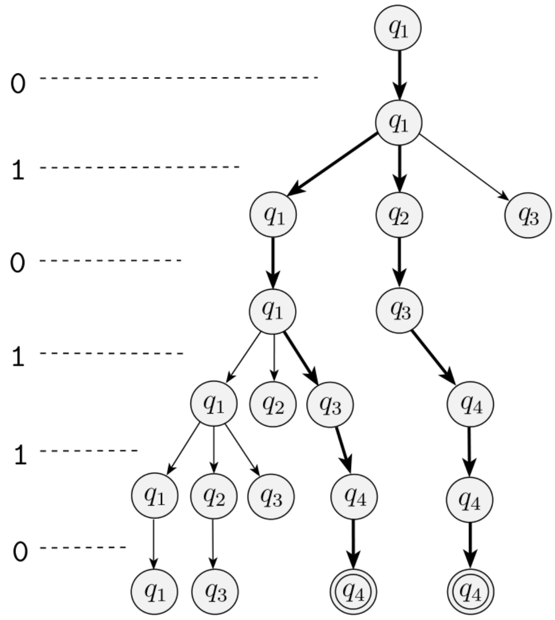
esso può essere descritto secondo la quintupla $N = (Q, \Sigma, \delta, q_0, F)$ come segue:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- δ è la seguente:

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\varnothing$
q_2	$\{q_3\}$	$\varnothing$	$\{q_3\}$
q_3	$\varnothing$	$\{q_4\}$	$\varnothing$
q_4	$\{q_4\}$	$\{q_4\}$	$\varnothing$

- q_1 è lo stato iniziale
- $F = \{q_4\} \subseteq Q$

Ad esempio, se N legge l'input `010110`, la sua computazione è la seguente:



Nota: nel momento in cui vengono incontrati ε-archi, si giunge allo stato successivo della computazione nello stesso step dell'input appena elaborato, senza produrre un passo ulteriore, producendo inoltre un nuovo ramo di computazione.

Definizione 1.3.1.2: Stringhe accettate (NFA)

Sia $N = (Q, \Sigma, \delta, q_0, F)$ un NFA, e sia $w = w_1 \cdots w_n$ una stringa tale per cui $\forall i \in [1, n] \quad w_i \in \Sigma$; allora, N **accetta** w se esiste una sequenza di stati $r_0, \ldots, r_n \in Q$ tali per cui

- $r_0 = q_0$
- $\forall i \in [0, n-1] \quad r_{i+1} \in \delta(r_i, w_{i+1})$
- $r_n \in F$

1.3.2 Equivalenze

Definizione 1.3.2.1: Equivalenza tra automi

Due automi si dicono **equivalenti** se e solo se riconoscono lo stesso linguaggio.

Teorema 1.3.2.1: DFA ed NFA

Le classi dei DFA e degli NFA sono equivalenti; in simboli

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

Dimostrazione.

Prima implicazione. Si noti che

$$\mathcal{L}(\mathsf{NFA}) \subseteq \mathcal{L}(\mathsf{DFA}) \iff \forall A \in \mathcal{L}(\mathsf{NFA}) \quad \exists M \in \mathsf{DFA} \mid A = L(M)$$

Allora, sia $A \in \mathcal{L}(\mathsf{NFA})$, e dunque esiste un NFA $N = (Q, \Sigma_\varepsilon, \delta, q_0, F)$ in grado di riconoscerlo. Inoltre, si definisca

$$\forall k \geq 0 \quad E(R) := \bigcup_{r \in R} \delta^k(r, \varepsilon)$$

l'insieme degli stati raggiungibili da stati in R, applicando (anche ripetutamente) un numero arbitrario di ε-archi (si noti che per $k = 0$ si ha che $R \subseteq E(R)$).

Allora, sia $M = (Q', \Sigma, \delta', q_0', F)$ il DFA definito come segue:

- $Q' := \mathcal{P}(Q)$, scelto tale da rappresentare ogni possibile stato di N;
- $\forall R \in Q', a \in \Sigma \quad \delta'(R, a) := \bigcup_{r \in R} E(\delta(r, a))$, scelta tale in quanto, per un certo insieme di stati $R \in Q'$ di N, a $\delta'(R, a)$ viene assegnata l'unione degli stati che sarebbero stati raggiunti in N dagli $r \in R$ con a, calcolati dunque attraverso $\delta(r, a)$, aggiungendo infine i possibili ε-archi;
- $q_0' := E(\{q_0\})$, scelto tale da far iniziare M esattamente dove aveva inizio N, comprendendo anche i possibili ε-archi iniziali;
- $F' := \{R \in Q' \mid \exists r \in R : r \in F\}$, che corrisponde all'insieme degli insiemi di stati di N contenenti almeno uno stato accettante in N.

Allora M è in grado di riconoscere A per costruzione, poiché il DFA costruito emula l'NFA di partenza, tenendo anche in considerazione gli ε-archi. Dunque, sia N che M riconoscono A, e per definizione sono di conseguenza equivalenti.

Seconda implicazione. Poiché il determinismo è un caso particolare del non determinismo, si ha che $\mathsf{DFA} \subseteq \mathsf{NFA} \implies \mathcal{L}(\mathsf{DFA}) \subseteq \mathcal{L}(\mathsf{NFA})$.

$\square$

Corollario 1.3.2.1: Linguaggi regolari ed NFA

Un linguaggio è regolare se e solo se esiste un NFA che lo riconosce; in simboli

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

Dimostrazione. Per il Teorema 1.3.2.1, si ha che

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

e dunque segue la tesi. □

1.4 Operazioni regolari

1.4.1 Unione

Definizione 1.4.1.1: Unione

Siano A e B due linguaggi su un alfabeto Σ; allora, si definisce l'**unione** di A e B il seguente linguaggio:

$$A \cup B := \{x \mid x \in A \vee x \in B\}$$

Si noti che, per ogni linguaggio L, è vero che $\emptyset \cup L = L \cup \emptyset = L$.

Esempio 1.4.1.1 (Unione)**.** Sia $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}\}$ l'alfabeto composto da 26 lettere, e siano $A = \{\mathtt{uno}, \mathtt{due}\}$ e $B = \{\mathtt{tre}, \mathtt{quattro}\}$ due linguaggi su Σ. Allora, si ha che

$$A \cup B = \{\mathtt{uno}, \mathtt{due}, \mathtt{tre}, \mathtt{quattro}\}$$

Proposizione 1.4.1.1: Chiusura sull'unione (REG)

Siano A e B due linguaggi regolari su un alfabeto Σ; allora $A \cup B$ è regolare.

Dimostrazione I. Per definizione, A e B sono linguaggi regolari, dunque esistono due DFA

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$
$$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$

tali da riconoscere rispettivamente A e B. Allora, sia $M = (Q, \Sigma, \delta, q_0, F)$ il DFA definito come segue:

- $Q := Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1 \wedge r_2 \in Q_2\}$, scelto tale in quanto permette di avere tutte le possibili combinazioni di stati dei due automi di partenza;
- $\forall (r_1, r_2) \in Q, a \in \Sigma \quad \delta((r_1, r_2), a) := (\delta_1(r_1, a), \delta_2(r_2, a))$, scelta tale in quanto permette di simulare entrambi gli automi di partenza contemporaneamente, mandando ogni stato di M_1 ed M_2 dove sarebbe andato nei rispettivi automi di appartenenza;

- $q_0 := (q_1, q_2)$, scelto tale in quanto deve essere lo stato in cui entrambe gli automi in ipotesi iniziavano;
- $F := (F_1 \times Q_2) \cup (Q_1 \times F_2) = \{(r_1, r_2) \mid r_1 \in F_1 \vee r_2 \in F_2\}$, scelto tale in quanto permette di simulare gli stati accettanti di entrambi gli automi, e vanno prese tutte le coppie che vedono almeno uno dei due stati come accettanti poiché altrimenti non si accetterebbero delle stringhe accettate da M_1 ed M_2 in partenza.

Allora, poiché M è in grado di simulare M_1 ed M_2 contemporaneamente, per costruzione accetterà ogni stringa di A e di B, dunque riconoscendo $A \cup B$, e di conseguenza $A \cup B$ è regolare per definizione. □

Dimostrazione II. Per definizione, A e B sono linguaggi regolari, dunque per il Teorema 1.3.2.1 esistono due NFA

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

tali da riconoscere rispettivamente A e B. Allora, sia $N = (Q, \Sigma, \delta, q_0, F)$ l'NFA costruito come segue:

- $Q := \{q_0\} \cup Q_1 \cup Q_2$, dove q_0 è un nuovo stato, e Q è scelto tale da includere sia N_1 che N_2;
- $\forall q \in Q, a \in \Sigma_\varepsilon \quad \delta(q, a) := \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \wedge a = \varepsilon \\ \varnothing & q = 0 \wedge a \neq \varepsilon \end{cases}$, scelta tale da poter eseguire contemporaneamente gli NFA N_1 ed N_2, definendo per casi la funzione di transizione; infatti, si noti che si è posta $\delta(q_0, \varepsilon) := \{q_1, q_2\}$ in modo da collegare il nuovo stato q_0 a q_1 e q_2, gli stati iniziali di N_1 ed N_2 rispettivamente;
- q_0 è il nuovo stato, che rappresenta lo stato iniziale di N;
- $F := F_1 \cup F_2$, scelto tale da costruire N in modo che accetti una stringa se e solo se la accetterebbero N_1 o N_2.

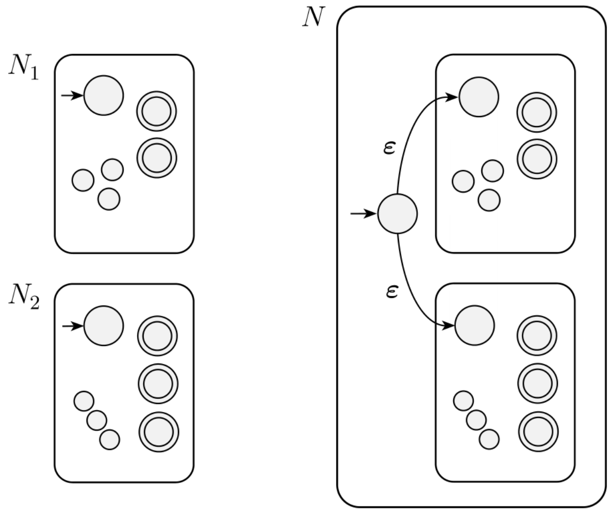


Figura 1.5: Rappresentazione dell'NFA N descritto.

Allora, l'NFA risultante N è in grado di computare contemporaneamente N_1 ed N_2, ed è dunque in grado di riconoscere A e B contemporaneamente; di conseguenza, N riconosce $A \cup B$, che risulta dunque essere regolare per definizione. □

1.4.2 Intersezione

Definizione 1.4.2.1: Intersezione

Siano A e B due linguaggi su un alfabeto Σ; allora, si definisce l'**intersezione** di A e B il seguente linguaggio:

$$A \cap B := \{x \mid x \in A \land x \in B\}$$

Esempio 1.4.2.1 (Intersezione)**.** Sia $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}\}$ l'alfabeto composto da 26 lettere, e siano $A = \{\mathtt{uno}, \mathtt{due}\}$ e $B = \{\mathtt{uno}, \mathtt{tre}\}$ due linguaggi su Σ. Allora, si ha che

$$A \cap B = \{\mathtt{uno}\}$$

Proposizione 1.4.2.1: Chiusura sull'intersezione (REG)

Siano A e B due linguaggi regolari su un alfabeto Σ; allora $A \cap B$ è regolare.

Dimostrazione. Per definizione, A e B sono linguaggi regolari, dunque esistono due DFA

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$
$$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$

tali da riconoscere rispettivamente A e B. Allora, sia $M = (Q, \Sigma, \delta, q_0, F)$ il DFA definito come segue:

- $Q := Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1 \wedge r_2 \in Q_2\}$, scelto tale in quanto permette di avere tutte le possibili combinazioni di stati dei due automi di partenza;
- $\forall (r_1, r_2) \in Q, a \in \Sigma \quad \delta((r_1, r_2), a) := (\delta_1(r_1, a), \delta_2(r_2, a))$, scelta tale in quanto permette di simulare entrambi gli automi di partenza contemporaneamente, mandando ogni stato di M_1 ed M_2 dove sarebbe andato nei rispettivi automi di appartenenza;
- $q_0 := (q_1, q_2)$, scelto tale in quanto deve essere lo stato in cui entrambe gli automi in ipotesi iniziavano;
- $F := F_1 \times F_2 = \{(r_1, r_2) \mid r_1 \in F_1 \wedge r_2 \in F_2\}$, scelto tale in quanto vanno prese tutte e sole le coppie composte da 2 stati entrambe accettanti negli automi di partenza.

Allora, poiché M è in grado di simulare M_1 ed M_2 contemporaneamente, ma accetta solo quando accettavano entrambe gli automi di partenza, per costruzione accetterà ogni stringa di $A \cap B$, che di conseguenza risulta essere regolare per definizione. □

1.4.3 Concatenazione

Definizione 1.4.3.1: Concatenazione

Siano A e B due linguaggi su un alfabeto Σ; allora, si definisce la **concatenazione** di A e B il seguente linguaggio:

$$A \circ B = \{xy \mid x \in A \wedge y \in B\}$$

Si noti che, per ogni linguaggio L, è vero che

- $\emptyset \circ L = L \circ \emptyset = \emptyset$
- $\{\varepsilon\} \circ L = L \circ \{\varepsilon\} = L$.

Inoltre, il simbolo $\circ$ può essere talvolta omesso.

Esempio 1.4.3.1 (Concatenazione)**.** Sia $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}\}$ l'alfabeto composto da 26 lettere, e siano $A = \{\mathtt{uno}, \mathtt{due}\}$ e $B = \{\mathtt{tre}, \mathtt{quattro}\}$ due linguaggi su Σ. Allora, si ha che

$$A \circ B := \{\mathtt{unotre}, \mathtt{unoquattro}, \mathtt{duetre}, \mathtt{duequattro}\}$$

Proposizione 1.4.3.1: Chiusura sulla concatenazione (REG)

Siano A e B due linguaggi regolari su un alfabeto Σ; allora $A \circ B$ è regolare.

Dimostrazione. Per definizione, A e B sono linguaggi regolari, dunque per il Teorema 1.3.2.1 esistono due NFA

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

tali da riconoscere rispettivamente A e B. Allora, sia $N = (Q, \Sigma, \delta, q_0, F)$ l'NFA costruito come segue:

- $Q := Q_1 \cup Q_2$, scelto tale da includere entrambe gli automi N_1 ed N_2 di partenza;
- $q_0 := q_1$, scelto tale da far iniziare l'esecuzione dell'automa su N_1;
- $F := F_2$, scelto tale da far terminare l'esecuzione dell'automa su N_2;
- $\forall q \in Q, a \in \Sigma_\varepsilon \quad \delta(q,a) := \begin{cases} \delta_1(q,a) & q \in Q_1 - F_1 \vee (q \in F_1 \wedge a \neq \varepsilon) \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \wedge a = \varepsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$,
  scelta tale da anteporre l'esecuzione di N_1 a quella di N_2; infatti, se $q \in Q_1 - F_1$ (è uno stato non accettante di N_1), oppure $q \in F_1$ ma $a \neq \varepsilon$, l'esecuzione di N_1 non viene alterata; diversamente, se invece $q \in F_1$ e $a = \varepsilon$, all'insieme di stati $\delta_1(q, \varepsilon)$ viene aggiunto q_2, ovvero lo stato iniziale di N_2, in modo da effettuare la concatenazione tra i due NFA non deterministicamente.

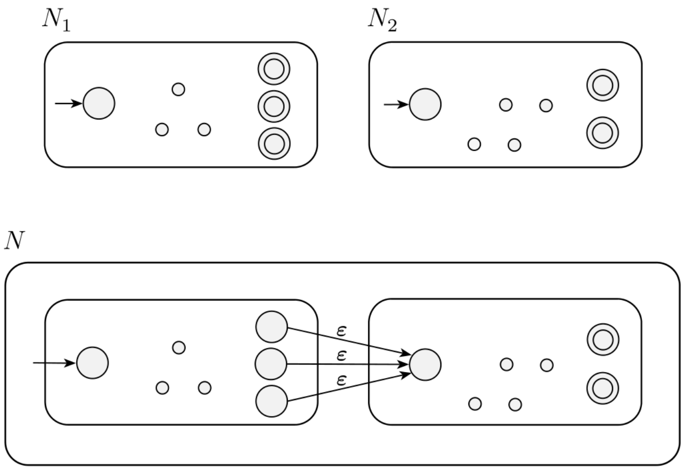


Figura 1.6: Rappresentazione dell'NFA N descritto.

Allora, l'NFA N costruito computa inizialmente N_1, e se vengono raggiunti suoi stati accettanti, l'esecuzione prosegue attraverso N_2, al fine di realizzare la concatenazione tra le stringhe. Di conseguenza, l'automa è in grado di riconoscere $A \circ B$ per costruzione, e dunque $A \circ B$ è regolare per definizione. □

1.4.4 Elevamento a potenza

Definizione 1.4.4.1: Elevamento a potenza

Sia A un linguaggio; allora, si definisce **elevamento a potenza** di A il seguente linguaggio:

$$A^n := \underbrace{A \circ \cdots \circ A}_{n \text{ volte}} = \begin{cases} A^0 := \{\varepsilon\} \\ A^n = A^{n-1} \circ A & n \geq 1 \end{cases}$$

Esempio 1.4.4.1 (Elevamento a potenza)**.** Sia $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}\}$ l'alfabeto composto da 26 lettere, e sia $A = \{\mathtt{uno}, \mathtt{due}\}$ un linguaggio su Σ. Allora, si ha che

$$A^2 = \{\varepsilon, \mathtt{uno}, \mathtt{due}, \mathtt{unouno}, \mathtt{unodue}, \mathtt{dueuno}, \mathtt{duedue}\}$$

Proposizione 1.4.4.1: Chiusura sull'elevamento (REG)

Sia A un linguaggio regolare su un alfabeto Σ, ed $n \in \mathbb{N}$; allora A^n è regolare.

Dimostrazione. La dimostrazione procede per induzione su $n \in \mathbb{N}$.

Caso base. Per $n = 0$, si ha che $A^0 = \{\varepsilon\}$, il quale è regolare poiché ad esempio il DFA

$$M_\varepsilon = (\{q\}, \Sigma, \delta, q, \{q\})$$

è in grado di riconoscerlo.

Ipotesi induttiva. Per $n \in \mathbb{N}$, si assume che A^n è regolare.

Passo induttivo. Per il caso $n+1$, si ha che $A^{n+1} = A^n \circ A$ per definizione dell'elevamento a potenza; inoltre, per ipotesi induttiva A^n è regolare, A è regolare per ipotesi, e poiché REG è chiuso rispetto alla concatenazione per la Proposizione 1.4.3.1, si ha che $A^{n+1} = A^n \circ A$ è regolare.

$\square$

1.4.5 Star

Definizione 1.4.5.1: Star

Sia A un linguaggio; allora, si definisce l'operazione unaria **star** che definisce il seguente linguaggio:

$$A^* := \{x_1 \cdots x_k \mid k \geq 0 \land \forall i \in [1, k] \quad x_i \in A\} = \bigcup_{n \in \mathbb{N}} L^n = \{\varepsilon\} \cup L \cup L^2 \cup \ldots$$

Si noti che $k = 0 \implies \varepsilon \in A^*$ per ogni linguaggio A.

Esempio 1.4.5.1 (Star)**.** Sia $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}\}$ l'alfabeto composto da 26 lettere, e sia $A = \{\mathtt{uno}, \mathtt{due}\}$ un linguaggio su Σ. Allora, si ha che

$$A^* := \{\varepsilon, \mathtt{uno}, \mathtt{due}, \mathtt{unouno}, \mathtt{unodue}, \mathtt{dueuno}, \mathtt{duedue}, \ldots\}$$

Esempio 1.4.5.2 (Stringhe binarie)**.** Si noti che nel caso di $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, si ha che Σ^* è l'insieme di ogni stringa binaria, di arbitraria lunghezza.

Proposizione 1.4.5.1: Chiusura sull'operazione star (REG)

Sia A un linguaggio regolare su un alfabeto Σ; allora A^* è regolare.

Dimostrazione. Per definizione, A è un linguaggio regolare, dunque per il Teorema 1.3.2.1 esiste un NFA

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

tale da riconoscere A. Allora, sia $N = (Q, \Sigma, \delta, q_0, F)$ l'NFA costruito come segue:

- $Q := \{q_0\} \cup Q_1$, dove q_0 è un nuovo stato, posto prima di N_1;
- q_0 è il nuovo stato iniziale;
- $F := \{q_0\} \cup F_1$, poiché q_0 deve essere accettante, in modo tale da accettare ε (si noti che $\varepsilon \in A^*$ per ogni linguaggio A);
- $\forall q \in Q, a \in \Sigma_\varepsilon \quad \delta(q, a) := \begin{cases} \delta_1(q, a) & q \in Q_1 - F_1 \vee (q \in F_1 \wedge a \neq \varepsilon) \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1 \wedge a = \varepsilon \\ \{q_1\} & q = q_0 \wedge a = \varepsilon \\ \varnothing & q = q_0 \wedge a \neq \varepsilon \end{cases},$
  scelta tale da ricominciare l'esecuzione dell'automa ogni volta che viene raggiunto uno stato accettante in N_1; infatti, se $q \in Q_1 - F_1$ (è uno stato non accettante di N_1), oppure q è accettante e $a \neq \varepsilon$, l'esecuzione procede normalmente con $\delta_1(q, a)$; differentemente, se $q \in F_1$ ma $a = \varepsilon$, allora l'esecuzione deve ricominciare da capo per poter effettuare la concatenazione multipla delle stringhe in A che caratterizzano l'operazione star, e dunque a $\delta_1(q, a)$ viene aggiunto q_1 (lo stato inziale di N_1); infine, ponendo $\delta(q_0, \varepsilon) := \{q_1\}$ si realizza l'ε-arco iniziale che collega q_0 (il nuovo stato) a q_1, al fine di rendere N in grado di accettare ε.

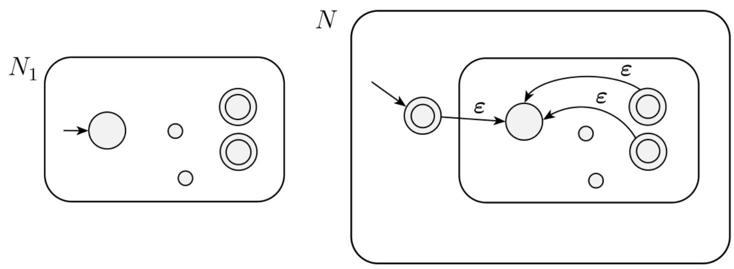


Figura 1.7: Rappresentazione dell'NFA N descritto.

Allora, poichè l'NFA N è in grado di ricominciare l'esecuzione ogni volta che questa sarebbe terminata in N_1, è in grado di accettare molteplici copie concatenate delle stringhe in A, in maniera non deterministica, e dunque per definizione N riconosce A^*, il quale risulta allora essere regolare per definizione. □

1.4.6 Complemento

Definizione 1.4.6.1: Complemento

Sia A un linguaggio definito su un alfabeto Σ; allora si definisce il **complemento** di A il seguente linguaggio:

$$\neg A := \{x \mid x \notin A\}$$

Esempio 1.4.6.1 (Complemento)**.** Sia $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ l'alfabeto binario, e sia $A = \{\mathtt{00}, \mathtt{1}\}$ un linguaggio definito su di esso. Allora il suo complemento è il seguente:

$$\neg A = \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{01}, \mathtt{10}, \mathtt{000}, \ldots\}$$

Proposizione 1.4.6.1: Chiusura sul complemento (REG)

Sia A un linguaggio regolare su un alfabeto Σ; allora $\neg A$ è regolare.

Dimostrazione. In ipotesi A è un linguaggio regolare, dunque per definizione esiste un DFA

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

che lo riconosce; allora, è sufficiente considerare il DFA

$$M' := (Q, \Sigma, \delta, q_0, Q - F)$$

per ottenere un DFA tale da riconoscere $\neg A$; allora, $\neg A$ è regolare per definizione. □

Proposizione 1.4.6.2: Leggi di De Morgan

Siano A e B definiti su un alfabeto Σ; allora, sono vere le seguenti, che prendono il nome di **leggi di De Morgan**:

i) $\neg(A \cup B) = \neg A \cap \neg B$

ii) $\neg(A \cap B) = \neg A \cup \neg B$

Dimostrazione. Omessa. □

1.5 Espressioni regolari

1.5.1 Definizioni

Definizione 1.5.1.1: Espressione regolare

Sia Σ un alfabeto; allora, R si definisce **espressione regolare** se soddisfa una delle seguenti caratteristiche:

- $R = \emptyset$
- $R = \varepsilon$
- $R \in \Sigma$

Un'espressione regolare, dunque, è un modo compatto di definire un linguaggio. Si noti che le definizioni successive sono in grado di espandere la definizione appena fornita. Dato un alfabeto Σ, la classe delle espressioni regolari definite su di esso è denotata con $\mathrm{re}(\Sigma)$. La classe di tutte le espressioni regolari è denotata con REX.

Data un'espressione regolare R, con $L(R)$ si intende il linguaggio che R descrive, ovvero l'insieme di stringhe che R rappresenta. Dunque, sono vere le seguenti:

- $L(\emptyset) = \emptyset$
- $L(\varepsilon) = \{\varepsilon\}$
- $\forall a \in \Sigma \quad L(a) = \{a\}$

Definizione 1.5.1.2: Unione

Siano R_1 ed R_2 due espressioni regolari su un alfabeto Σ; allora, si definisce l'**unione** di R_1 ed R_2 la seguente espressione regolare:

$$(R_1 \cup R_2)$$

e rappresenta uno qualsiasi dei caratteri di R_1 o di R_2. Dunque, si ha che

$$\forall R \in \mathrm{re}(\Sigma) \mid \exists R_1, R_2 \in \mathrm{re}(\Sigma) : R = R_1 \cup R_2 \quad L(R) = L(R_1) \cup L(R_2)$$

Si noti che, per ogni espressione regolare R, è vero che $\emptyset \cup R = R \cup \emptyset = \emptyset$.

Esempio 1.5.1.1 (Unione)**.** Sia $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ un alfabeto; un esempio di espressione regolare di unione su Σ è il seguente:

$$R = (\mathtt{a} \cup \mathtt{c})$$

ed il valore di questa espressione equivale da $\mathtt{a}$ oppure $\mathtt{c}$, e dunque $L(R) = \{\mathtt{a}, \mathtt{c}\}$.

Esempio 1.5.1.2 (Espressioni regolari particolari)**.** Sia $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ un alfabeto; l'espressione regolare $(\mathtt{0} \cup \mathtt{1})$ rappresenta il linguaggio che consiste di tutte le stringhe di lunghezza 1 sull'alfabeto Σ, e dunque l'espressione regolare descritta si abbrevia talvolta con il simbolo Σ stesso.

Definizione 1.5.1.3: Concatenazione

Siano R_1 ed R_2 due espressioni regolari; allora, si definisce la **concatenazione** di R_1 ed R_2 la seguente espressione regolare:

$$(R_1 \circ R_2)$$

e rappresenta le stringhe che iniziano per R_1 e terminano con R_2. Dunque, si ha che

$$\forall R \in \mathrm{re}(\Sigma) \mid \exists R_1, R_2 \in \mathrm{re}(\Sigma) : R = R_1 \circ R_2 \quad L(R) = L(R_1) \circ L(R_2)$$

Per indicare la concatenazione di R con sé stessa, si usa la seguente notazione

$$R^k := \underbrace{R \circ \cdots \circ R}_{k \text{ volte}}$$

Si noti che, per ogni espressione regolare R, è vero che:

- $\emptyset \circ R = R \circ \emptyset = \emptyset$
- $\varepsilon \circ R = R \circ \varepsilon = R$

Inoltre, il simbolo $\circ$ può essere talvolta omesso.

Esempio 1.5.1.3 (Concatenazione)**.** Sia $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ un alfabeto; un esempio di espressione regolare di concatenazione su Σ è il seguente:

$$R = (\mathtt{a} \circ \mathtt{c})$$

che può essere scritto equivalentemente come $\mathtt{ac}$, e dunque $L(R) = \{\mathtt{ac}\}$.

Definizione 1.5.1.4: Star

Sia R un'espressione regolare; allora, si definisce l'operazione unaria **star** su R la seguente espressione regolare:

$$(R^*)$$

e tutte rappresenta le stringhe che si possono ottenere concatenando un qualsiasi numero di caratteri di R, e descrive dunque il linguaggio che consiste di tutte le stringhe dell'alfabeto di R. Dunque, si ha che

$$\forall R \in \mathrm{re}(\Sigma) \mid \exists R_1 \in \mathrm{re}(\Sigma) : R = R_1^* \quad L(R) = L(R_1)^*$$

Si noti che R^* comprende ε per qualsiasi espressione regolare R.

Spesso si usa la notazione $R^+ := RR^*$, ovvero le stringhe che si ottengono attraverso la concatenazione di 1 o più stringhe di R. Di conseguenza, si ha che $R^+ \cup \varepsilon = R^*$.

Esempio 1.5.1.4 (Star)**.** Sia $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ un alfabeto; un esempio di espressione regolare di star su Σ è il seguente:

$$R = (\Sigma^*)$$

che descrive il linguaggio che consiste di tutte le stringhe possibili sull'alfabeto, e dunque

$$L(R) = \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{aa}, \mathtt{bb}, \mathtt{cc}, \ldots\}$$

Esempio 1.5.1.5 (Espressioni regolari)**.** Sia $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ un alfabeto; i seguenti sono esempi di espressioni regolari su Σ:

- $L(\mathtt{0}^*\mathtt{1}\mathtt{0}^*) = \{w \mid w$ contiene un solo $\mathtt{1}\}$;
- $L(\Sigma^*\mathtt{001}\Sigma^*) = \{w \mid w$ contiene la stringa $\mathtt{001}$ come sottostringa$\}$;
- $L(\mathtt{0}\Sigma^*\mathtt{0} \cup \mathtt{1}\Sigma^*\mathtt{1} \cup \mathtt{0} \cup \mathtt{1}) = \{w \mid w$ inizia e termina con lo stesso carattere$\}$, poiché si noti che l'ordine delle operazioni, a meno di parentesi, è (i) star, (ii) concatenazione, ed infine (iii) unione;
- $L((\mathtt{0} \cup \varepsilon)(\mathtt{1} \cup \varepsilon)) = \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{01}\}$;
- $L(\emptyset^*) = \{\varepsilon\}$, poiché $\emptyset$ rappresenta il linguaggio vuoto, e dunque l'unica stringa che si può ottenere concatenando un qualsiasi numero di volte elementi del linguaggio vuoto, è ε.

Definizione 1.5.1.5: Linguaggi descritti da espressioni regolari

Dato un alfabeto Σ, si definisce **classe dei linguaggi di Σ descritti da espressioni regolari** il seguente insieme:

$$\mathcal{L}(\mathrm{re}(\Sigma)) := \{L \subseteq \Sigma^* \mid \exists R \in \mathrm{re}(\Sigma) : L = L(R)\}$$

1.6 Configurazioni

1.6.1 Configurazioni di DFA

Definizione 1.6.1.1: Estensione di δ (DFA)

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA; è possibile estendere la definizione della funzione di transizione δ, utilizzando la notazione dell'operazione star, mediante la seguente definizione ricorsiva:

$$\delta^* : Q \times \Sigma^* \to Q : (q, x) \mapsto \begin{cases} \delta^*(q, \varepsilon) = \delta(q, \varepsilon) \\ \delta^*(q, by) = \delta^*(\delta(q, b), y) \quad b \in \Sigma, y \in \Sigma^* \mid x = by \end{cases}$$

Si noti che tale funzione prende in input uno stato ed una stringa, e restituisce lo stato in cui il DFA si troverà al termine della lettura dell'intera stringa di input. La notazione δ^* è coerente con la definizione dell'operazione star, poiché viene calcolata la transizione di stati attraverso δ fintanto che l'input non è stato esaurito.

Definizione 1.6.1.2: Configurazione (DFA)

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA; una tupla $(q, x) \in Q \times \Sigma^*$ è detta **configurazione** di M se q è pari allo stato attuale della computazione di un certo input, ed x è la porzione di input rimanente da leggere.

Definizione 1.6.1.3: Relazione tra configurazioni (DFA)

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA, e siano $(p, x), (q, y) \in Q \times \Sigma^*$ due sue configurazioni durante la computazione di un certo input; allora, tali due configurazioni si dicono essere **in relazione** se e solo se dall'una è possibile passare all'altra. In simboli:

$$(p, x) \vdash_M (q, y) \iff \begin{cases} p, q \in Q \\ x, y \in \Sigma^* \\ \exists a \in \Sigma \mid x = ay \land \delta(p, a) = q \end{cases}$$

Osservazione 1.6.1.1: Chiusura transitiva di $\vdash$

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA; si noti che la chiusura transitiva della relazione tra configurazioni $\vdash$, ovvero $\vdash^*$, equivale a calcolare gli input attraverso la funzione di transizione estesa δ^* definita nella Definizione 1.6.1.1.

Esempio 1.6.1.1 (Chiusura transitiva di $\vdash$)**.** Sia $M = (Q, \sigma, \delta, q_0, F)$ un DFA, e sia $(p, x) \in Q \times \Sigma^*$ una sua configurazione durante la computazione di un certo input; inoltre, siano $a, b, c \in \Sigma$ tali che $x = abc$. Inoltre, siano vere le seguenti:

- $(p, abc) \vdash_M (p_1, bc)$
- $(p_1, bc) \vdash_M (p_2, c)$
- $(p_2, c) \vdash_M (q, \varepsilon)$

per certi $p, p_1, p_2, q \in Q$; allora, si ha che $(p, x) \vdash_M^* (q, \varepsilon)$, poiché è possibile raggiungere lo stato q, partendo da p, attraverso l'input $x = abc$.

Osservazione 1.6.1.2: Determinismo

Dato un DFA $M = (Q, \Sigma, \delta, q_0, F)$, è possibile fornire una definizione di *determinismo* alternativa, attraverso la relazione tra configurazioni definita nella Definizione 1.6.1.3, come segue:

$$\forall q \in Q, a \in \Sigma, x \in \Sigma^* \quad \exists! p \in Q \mid (q, ax) \vdash_M (p, x)$$

1.6.2 Configurazioni di NFA

TODO

Osservazione 1.6.2.1: Linguaggio di un automa

Dato un automa M, la Definizione 1.2.1.3 si può riscrivere in simboli come segue:

$$L(M) := \{x \in \Sigma^* \mid \delta^*(q_0, x) \in F \iff \exists q \in F \mid (q_0, x) \vdash_M^* (q, \varepsilon)\}$$

Si noti che, nella definizione, verrà presa in considerazione la funzione δ^* corrispondente alla classe dell'automa M in questione.

1.7 Non determinismo generalizzato

1.7.1 Definizioni

Definizione 1.7.1.1: GNFA

Un **GNFA** (*Generalized Nondeterministic Finite Automaton*) è una versione generalizzata di un NFA, in cui gli archi delle transizioni sono espressioni regolari sull'alfabeto dato.

Un GNFA legge blocchi di simboli dall'input, e si muove lungo gli archi che sono etichettati da espressioni regolari che possono descrivere il blocco di simboli letto.

Inoltre, essendo una versione generalizzata di un NFA, un GNFA può avere diversi modi di elaborare la stessa stringa di input, e accetta quest'ultima se la sua elaborazione può far si che il GNFA si trovi in uno stato accettante al suo termine.

Nota: All'interno di questi appunti, a meno di specifica, si assume che ogni GNFA preso in considerazione abbia:

- un solo stato di inizio, privo di archi entranti, connesso con ogni altro stato, ma non con sé stesso;
- un solo stato accettante, privo di archi uscenti, non connesso con altri archi;
- ogni altro stato collegato con ogni altro stato, a meno di quello iniziale, anche con sé stessi.

Formalmente, dato un alfabeto Σ, un GNFA del tipo appena descritto è una quintupla $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ definita come segue:

- Q è l'**insieme degli stati** dell'automa, un insieme *finito*
- Σ è l'**alfabeto dell'automa**, un insieme *finito*
- $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to \text{re}(\Sigma)$ è la **funzione di transizione**, che definisce la relazione tra gli stati; si noti che δ ha come dominio il prodotto cartesiano tra gli stati, e come codominio $\text{re}(\Sigma)$, poiché a differenza di un normale DFA o NFA, un GNFA prende come input 2 stati (che non possono essere né quello iniziale né quello accettante) e restituisce un'espressione regolare; inoltre, il dominio di δ è $(Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\})$ per evitare di includere l'arco $(q_{\text{accept}}, q_{\text{start}})$
- $q_{\text{start}} \in Q$ è lo **stato iniziale**
- $q_{\text{accept} \in Q}$ è lo **stato accettante**

Esempio 1.7.1.1 (GNFA)**.** Il seguente è il digramma di un GNFA sull'alfabeto $\Sigma = \{\texttt{a}, \texttt{b}\}$.

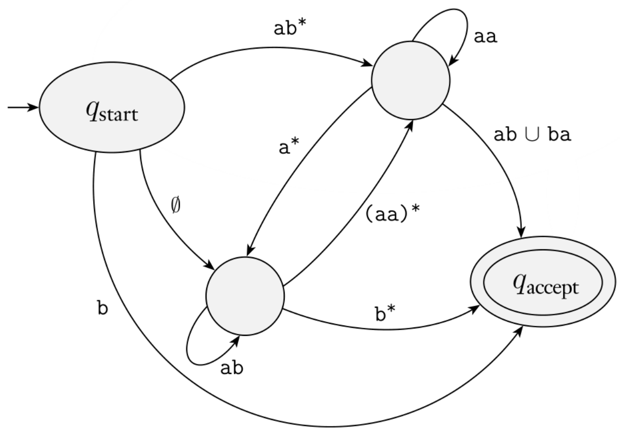


Figura 1.8: Un GNFA.

Definizione 1.7.1.2: Stringhe accettate (GNFA)

Sia $G = (Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ un GNFA, e sia $w = w_1 \cdots w_n$ una stringa tale per cui $\forall i \in [1, n] \quad w_i \in \Sigma^*$; allora, G **accetta** w se esiste una sequenza di stati $q_0, \ldots, q_n \in Q$ tali per cui

- $q_0 = q_{\text{start}}$
- $\forall i \in [0, n-1] \quad w_i \in L(\delta(q_i, q_{i+1}))$, ovvero, w_i deve far parte del linguaggio rappresentato dall'espressione regolare sull'arco da q_i a q_{i+1}
- $q_n = q_{\text{accept}}$

1.7.2 Equivalenze

Metodo 1.7.2.1: GNFA di DFA

Sia M un DFA; allora per costruire un GNFA ad esso equivalente, è sufficiente:

- aggiungere un nuovo stato iniziale, con un ε-arco entrante sul vecchio stato iniziale;
- aggiungere un nuovo stato accettante, con ε-archi entranti provenienti dai vecchi stati accettanti;
- sostiuire gli archi con etichette multiple, con archi aventi come etichetta l'unione delle etichette;
- aggiungere archi etichettati con $\emptyset$ tra gli stati non collegati (si noti che questa operazione non varia l'automa di partenza, poiché un arco etichettato con $\emptyset$ non potrà mai essere utilizzato)

Esempio 1.7.2.1 (GNFA di DFA)**.** Si consideri il seguente DFA

$$D = (\{b, c\}, \{0, 1, 2\}, \delta, b, \{c\})$$

rappresentato come segue:

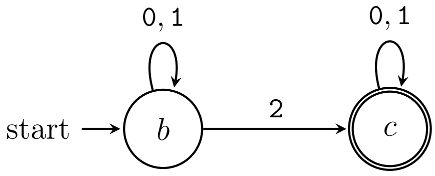


Figura 1.9: Il DFA D.

Allora, il suo GNFA equivalente è il seguente:

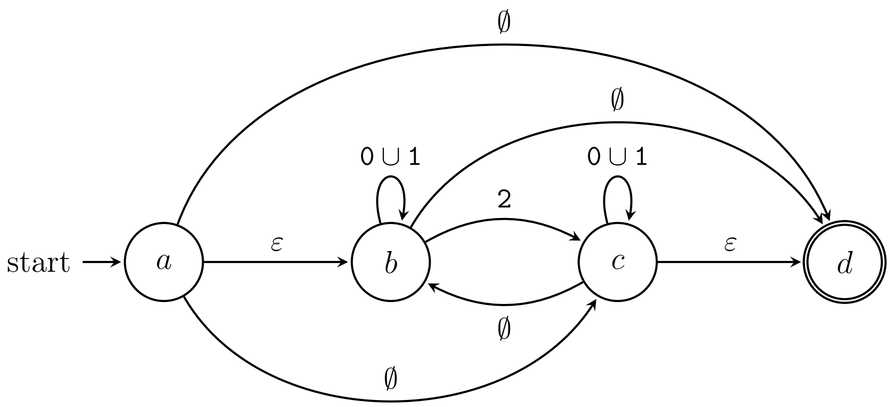


Figura 1.10: Il GNFA di D.

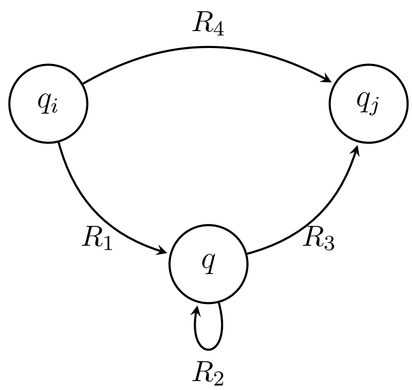
Algoritmo 1.7.2.1: Espressione regolare di un GNFA

Dato un GNFA $G = (Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, l'algoritmo restituisce un'espressione regolare equivalente a G.

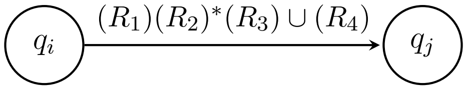
1: **function** CONVERTGNFATOREGEX(G)
2:     **if** $|Q| == 2$ **then**
3:         **return** $\delta(q_{\text{start}}, q_{\text{accept}})$
4:     **else if** $|Q| > 2$ **then**
5:         $q \in Q - \{q_{\text{start}}, q_{\text{accept}}\}$
6:         $Q' := Q - \{q\}$
7:         **for** $q_i \in Q' - \{q_{\text{accept}}\}$ **do**
8:             **for** $q_j \in Q' - \{q_{\text{start}}\}$ **do**
9:                 $\delta'(q_i, q_j) := \delta(q_i, q)\delta(q, q)^*\delta(q, q_j) \cup \delta(q_i, q_j)$
10:             **end for**
11:         **end for**
12:         $G' := (Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$
13:         **return** `convertGNFAtoRegEx`(G')
14:     **end if**
15: **end function**

Idea. L'algoritmo inizia prendendo in input un GNFA G, ed inizialmente viene controllato il numero di stati di G:

- se $|Q| = 2$, allora sicuramente $Q = \{q_{\text{start}}, q_{\text{accept}}\}$, e poiché si vuole resituire l'espressione regolare equivalente a G, di fatto quest'ultimo è costituito esclusivamente dall'espressione regolare posta sull'arco tra q_{start} e q_{accept}, dunque è sufficiente restituirla in output (riga 3);
- se $|Q| > 2$, allora viene costruito un GNFA G', avente uno stato in meno, ovvero q (scelto alla riga 5), naturalmente diverso da q_{start} e da q_{accept}; successivamente, per ogni coppia di stati (q_i, q_j), viene definita $\delta'(q_i, q_j)$ in modo tale da accorpare tutte le possibili configurazioni di stati; ad esempio, prendendo in esame il seguente GNFA



dove R_1, R_2, R_3 ed R_4 sono espressioni regolari, è possibile vedere che il seguente GNFA è ad esso equivalente



poiché l'arco che q ha su sé stesso è stato descritto attraverso $(R_2)^*$, gli archi (q_i, q) e (q, q_j) sono stati inseriti per concatenazione, ed infine è stato unito l'altro possibile cammino verso q_j tramite unione;

- inoltre, si noti che tale espressione regolare — contenuta nella riga 9 — tiene in considerazione tutte le possibili configurazioni di archi tra stati di un GNFA, per come è stato definito il GNFA all'interno della Definizione 1.7.1.1; infatti, per $|Q| > 2$, tra due stati q_i e q_j, oltre ad avere la garanzia che esista l'arco (q_i, q_j), esiste sicuramente un terzo stato q intermedio tale per cui esistano archi (q_i, q) e (q, q_j), ed esiste anche l'arco (q, q);
- infine, sia per la Definizione 1.7.1.1, sia per come la riga 5 dell'algoritmo opera, per ogni GNFA si ha che $|Q| \geq 2$, dunque non è necessario gestire ulteriori casi.

Allora, l'algoritmo è in grado di restituire l'espressione regolare equivalente a G.

Dimostrazione. La dimostrazione procede per induzione su k, il numero di stati del GNFA.

Caso base. Se $k = 2$, ovvero se G ha solo 2 stati, necessariamente $Q = \{q_{\text{start}}, q_{\text{accept}}\}$, e dunque $\texttt{convertGNFAtoRegEx}(G) = \delta(q_{\text{start}}, q_{\text{accept}})$, come descritto nella riga 3.

Ipotesi induttiva. Se G è un GNFA con $k - 1$ stati, allora $\texttt{convertGNFAtoRegEx}(G)$ è un'espressione regolare equivalente a G.

Passo induttivo. È necessario dimostrare che, se G è un GNFA con k stati, allora $\texttt{convertGNFAtoRegEx}(G)$ è un'espressione regolare equivalente a G. In primo luogo, è necessario dimostrare che G e G' sono equivalenti, e dunque sia w una stringa accettata da G; allora, in un ramo accettante della computazione, G entra in una sequenza di stati

$$q_{\text{start}}, q_1, q_2, \ldots, q_{\text{accept}}$$

dunque, se il ramo non contiene lo stato rimosso q, allora sicuramente G' accetta w; viceversa, se q è contenuto all'interno di tale ramo, allora per quanto discusso all'interno dell'idea di dimostrazione, l'espressione regolare inserita al posto dello stato q rimosso è in grado di tenere in considerazione ogni possibile configurazione di archi, e dunque G' accetta sicuramente w; infine, è possibile applicare la stessa osservazione per mostrare che una stringa accettata da G' deve essere necessariamente accettata anche da G. Allora, poiché G e G' sono equivalenti, e G ha k stati, allora necessariamente al termine del k-esimo passo dell'algoritmo, G' avrà $k - 1$ stati, e su di esso è possibile applicare l'ipotesi induttiva.

□

Esempio 1.7.2.2 (Espressioni regolari di GNFA)**.** Sia G il GNFA dell'Esempio 1.7.2.1; allora, la sua espressione regolare equivalente è ottenibile attraverso i seguenti passaggi:

- viene inizialmente rimosso lo stato b, ottenendo l'automa G':

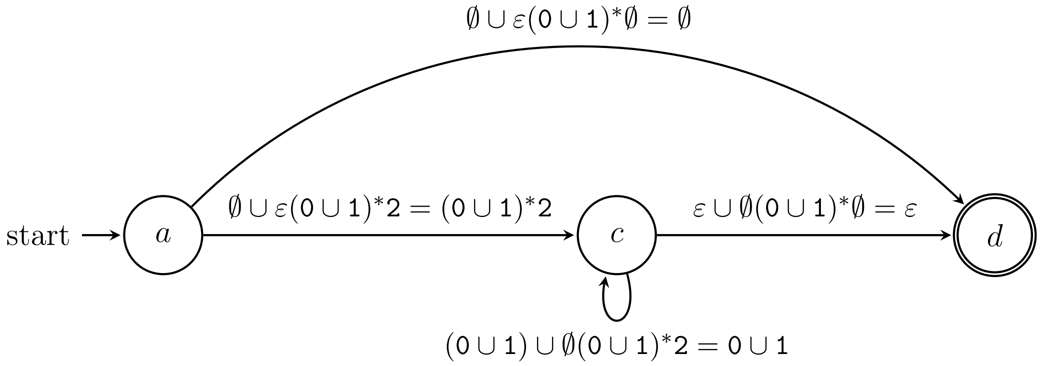


Figura 1.11: G', ovvero il GNFA G, dopo aver rimosso b.

- successivamente, viene rimosso lo stato c, ottenendo l'automa G'':

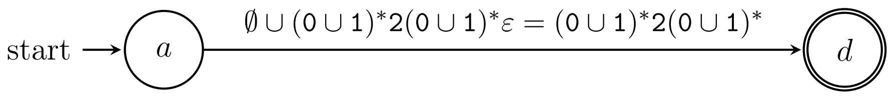


Figura 1.12: G'', ovvero il GNFA G', dopo aver rimosso c.

- allora, poiché G'' ha 2 stati, l'espressione regolare cercata — equivalente a G — è posta sull'arco (a, d), ed è

$$(0 \cup 1)^*2(0 \cup 1)^*$$

Teorema 1.7.2.1: Linguaggi ed espressioni regolari

Un linguaggio è regolare se e solo se esiste un'espressione regolare che lo descrive; dunque, tutti e soli i linguaggi regolari sono descritti da espressioni regolari. In simboli, per un certo alfabeto Σ si ha che

$$\mathsf{REG} = \mathcal{L}(\mathsf{REX})$$

Dimostrazione.

Prima implicazione. Sia A un linguaggio regolare; allora, per definizione, esiste un DFA che lo riconosce, e sia questo M. Utilizzando il Metodo 1.7.2.1, è possibile costruire un GNFA che sia equivalente ad M; sia quest'ultimo G. Allora, è sufficiente applicare l'Algoritmo 1.7.2.1 su G per ottenere l'espressione regolare ad esso equivalente; dunque, segue la tesi.

Seconda implicazione. Sia A un linguaggio su un alfabeto Σ, descritto da un'espressione regolare R. Allora, la dimostrazione procede costruendo degli NFA, per casi, come segue:

- $R = \emptyset \implies L(R) = \emptyset$; allora, il seguente NFA è in grado di riconoscere $L(R)$:

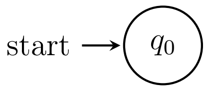


Figura 1.13: Un NFA in grado di riconoscere $L(\emptyset)$.

L'automa mostrato è descritto dalla quintupla $N = (\{q_0\}, \Sigma, \delta, q_0, \varnothing)$, dove $\forall a \in \Sigma \quad \delta(q_0, a) = \emptyset$.

- $R = \varepsilon \implies L(R) = \{\varepsilon\}$; allora, il seguente NFA è in grado di riconoscere $L(R)$:

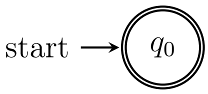


Figura 1.14: Un NFA in grado di riconoscere $L(\varepsilon)$.

L'automa mostrato è descritto dalla quintupla $N = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$, dove $\forall a \in \Sigma \quad \delta(q_0, a) = \emptyset$.

- $R \in \Sigma \implies L(R) = \{a\}$ per qualche $a \in \Sigma$; allora, il seguente NFA è in grado di riconoscere $L(R)$:

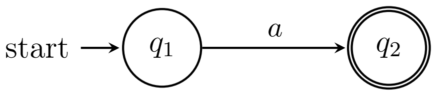


Figura 1.15: Un NFA in grado di riconoscere $L(a)$.

L'automa mostrato è descritto dalla quintupla $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, dove $\forall q \in \{q_1, q_2\}, x \in \Sigma \quad \delta(q, x) = \begin{cases} \{q_2\} & q = q_1 \land x = a \\ \emptyset & q \neq q_1 \lor x \neq a \end{cases}$.

- si noti che se esistono due espressioni regolari R_1 ed R_2 tali che $R = R_1 \cup R_2$, oppure $R = R_1 \circ R_2$, o ancora $R = R_1^*$, è sufficiente costruire gli NFA che sono stati costruiti nelle dimostrazioni della Proposizione 1.4.1.1, della Proposizione 1.4.3.1 e della Proposizione 1.4.5.1.

Allora, per qualsiasi espressione regolare R, tale da descrivere un certo linguaggio A, è possibile costruire un NFA che riconosce il linguaggio che R descrive. Allora, per il Corollario 1.3.2.1, A è regolare.

$\square$

Osservazione 1.7.2.1: Equivalenze tra classi di automi

Si noti che, dal Teorema 1.7.2.1, dall'Algoritmo 1.7.2.1, e dal Metodo 1.7.2.1, si ha che

$$\mathsf{REG} = \mathcal{L}(\mathsf{REX}) \supseteq \mathcal{L}(\mathsf{GNFA}) \supseteq \mathsf{REG}$$

dunque, segue l'importante risultato:

$$\mathsf{REG} := \mathcal{L}(\mathsf{DFA}) = \mathcal{L}(\mathsf{NFA}) = \mathcal{L}(\mathsf{GNFA}) = \mathcal{L}(\mathsf{REX})$$

1.8 Linguaggi non regolari

1.8.1 Pumping lemma

Principio 1.8.1.1: Principio della piccionaia

Siano A e B due insiemi finiti, tali che $|B| < |A|$; allora, non esiste alcuna funzione iniettiva $f : A \to B$.

In altri termini, avendo una piccionaia con m caselle, non è possibile inserire più di m piccioni al suo interno: alcuni volatili dovranno necessariamente condividere la propria casella.

Definizione 1.8.1.1: Linguaggio non regolare

Un linguaggio A si definisce **non regolare** se non esiste un DFA in grado di riconoscerlo. In simboli

$$A \notin \mathsf{REG}$$

Lemma 1.8.1.1: Pumping lemma (REG)

Sia A un linguaggio regolare; allora, esiste un $p \in \mathbb{N}$, detto **lunghezza del pumping**, tale che per ogni stringa $s \in A$ tale per cui $|s| \geq p$, esistono 3 stringhe $x, y, z \mid s = xyz$, soddisfacenti le seguenti condizioni:

- $\forall i \geq 0 \quad xy^iz \in A$
- $|y| > 0$ (o, equivalentemente, $y \neq \varepsilon$)
- $|xy| \leq p$

Dimostrazione. Poiché A è un linguaggio regolare in ipotesi, per definizione esiste un DFA $M = (Q, \Sigma, \delta, q_1, F)$ in grado di riconoscerlo. Allora, sia $p := |Q|$, e sia $s \in A \mid s = s_1 s_2 \cdots s_n$ tale che $\forall i \in [1, n] \quad s_i \in \Sigma$, e $n \geq p$. Inoltre, sia

$$\forall i \in [1, n] \quad r_{i+1} := \delta(r_i, s_i)$$

la sequenza di stati attraversati da M mentre elabora s; dunque, si ha che $r_{n+1} \in F$. Si noti che la sequenza di stati ha dunque cardinalità

$$|\{r_1, \dots, r_{n+1}\}| = n + 1$$

in quanto devono essere attraversati n archi, e dunque $n+1$ stati. Inoltre, $n \geq p \iff n + 1 \geq p + 1$, e dunque per il Principio 1.8.1.1, poiché $p := |Q|$, due dei primi $p + 1$ stati della sequenza devono necessariamente essere lo stesso stato; siano r_j il primo ed r_l il secondo, con $j \neq l$. Allora, sicuramente $l \leq p+1$, poiché r_l è uno stato tra i primi $p+1$.

Si pongano dunque

$$\begin{cases} x := s_1 \cdots s_{j-1} \\ y := s_j \cdots s_{l-1} \\ z := s_l \cdots s_n \end{cases}$$

Allora, si ha che:

- x porta M da r_1 ad r_j, y porta M da $r_j = r_l$ ad r_j — dunque y porta M da r_j ad r_j stesso — ed infine z porta M da r_j ad r_{n+1}, e poiché $r_{n+1} \in F$, M accetta sicuramente xy^iz per ogni $i \geq 0$;
- $j \neq l \implies |y| > 0$;
- $l \leq p + 1 \implies |xy| \leq p$, poiché r_l è lo stato che viene dopo aver letto $s_{l-1} \in y$, e dunque nel caso limite si ha che $l = p + 1 \implies |xy| = p$.

Allora, sono soddisfatte tutte le condizioni della tesi. $\square$

Esempio 1.8.1.1 (Dimostrazione del pumping lemma)**.** La seguente rappresentazione raffigura un automa definito come segue:

$$M = (\{q_1, \dots, q_9, \dots, q_{13}\}, \Sigma, \delta, q_1, \{q_{13}\})$$

dove $|Q| = 13$, e $r_j = r_l = q_9$ è lo stato che si ripete all'interno dei primi $p+1$ stati della sequenza presa in esame nella dimostrazione del Lemma 1.8.1.1.

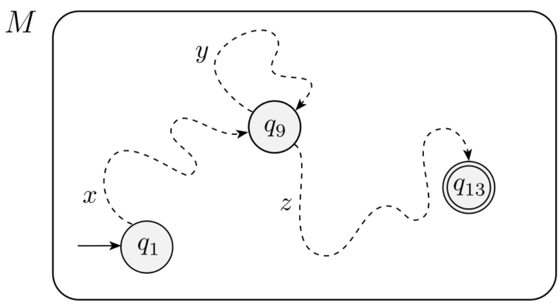


Figura 1.16: L'automa M descritto

Esempio 1.8.1.2 (Pumping lemma in REG)**.** Si consideri il linguaggio

$$L := \{\mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N}\}$$

e per assurdo, sia $L \in \mathsf{REG}$, e dunque per esso vale il Lemma 1.8.1.1. Allora, sia $p \in \mathbb{N}$ la lunghezza del pumping di L, e si consideri la seguente stringa

$$s := \mathtt{0}^p\mathtt{1}^p \implies |s| = 2p > p$$

avente lunghezza sicuramente maggiore di p. Siano inoltre x, y, z tali da soddisfare il pumping lemma, ed in particolare $|xy| \le p$, ma poiché $s := \mathtt{0}^p\mathtt{1}^p = xy^iz$ per ogni $i \in \mathbb{N}$, allora necessariamente la stringa xy è composta da soli $\mathtt{0}$, ed inoltre $|y| > 0$ implica che y ha almeno uno $\mathtt{0}$. Siano allora

- $k := |x| \implies x = \mathtt{0}^k$, poiché x è composta solo da $\mathtt{0}$
- $m := |y| > 0 \implies y = \mathtt{0}^m$, poiché y è composta solo da $\mathtt{0}$ (ed almeno uno)
- $|xy| \le p \implies k + m \le p \implies z = \mathtt{0}^{p-m-k}\mathtt{1}^p$ per gli $\mathtt{0}$ restanti

e, ponendo ad esempio $i = 2$, si ottiene che

$$xy^2z = \mathtt{0}^k\mathtt{0}^{2m}\mathtt{0}^{p-m-k}\mathtt{1}^p$$

ma il numero di $\mathtt{1}$ di questa stringa è pari a p, mentre il numero di $\mathtt{0}$ è pari a

$$k + 2m + (p - m - k) = m + p$$

e poiché $m > 0$, si ha che $m + p > p$. Dunque $xy^2z \notin L$ poiché non è una stringa della forma $\mathtt{0}^n\mathtt{1}^n$ $\lightning$.

Osservazione 1.8.1.1: Condizioni del pumping lemma (REG)

Si noti che la seconda condizione del Lemma 1.8.1.1 stabilisce che $y \neq \varepsilon$, poiché altrimenti il teorema sarebbe trivialmente verificato; infatti, ammettendo $y = \varepsilon$, dato un linguaggio $A \in \mathsf{REG}$, e presa una sua stringa $s \in A$, ponendo $p := |x|$ esistono sicuramente stringhe $x, z \mid s = xz$ tali da verificare le restanti condizioni del lemma, infatti:

- $\forall i \ge 0 \quad xy^iz = x\varepsilon^iz = xz =: s \in A$
- $|xy| = |x\varepsilon| = |x| \le p := |x|$

e non sarebbe necessaria l'ipotesi per cui A debba essere regolare; dunque, non sarebbe possibile utilizzare tale lemma per determinare la classe di un dato linguaggio.

2

Linguaggi e grammatiche context-free

2.1 Grammatiche context-free

2.1.1 Definizioni

Definizione 2.1.1.1: Grammatica

Una grammatica è un insieme di regole di sostituzione di stringhe, in grado di produrre quest'ultime a partire da una variabile iniziale, mediante una sequenza di scambi. Le regole sono scritte nella forma

$$\alpha A\beta \to \alpha\gamma\beta$$

Definizione 2.1.1.2: Acontestualità

Si definisce **acontestualità** la condizione per cui il lato sinistro delle regole di una grammatica è composto sempre da un solo simbolo.

Definizione 2.1.1.3: CFG

Una **grammatica context-free** — detta anche **acontestuale** —, o CFG (*Context-Free Grammar*) è una quadrupla (V, Σ, R, S), dove

- V è l'insieme delle **variabili**, un insieme *finito*
- Σ è l'insieme dei **terminali**, un insieme *finito*, dove $\Sigma \cap V = \varnothing$
- R è l'insieme delle **regole** o **produzioni**, un insieme *finito*
- $S \in V$ è la **variabile iniziale**, ed è generalmente il simbolo alla sinistra della prima regola della grammatica

Le CFG si scrivono nella forma

$$X \to Y$$

dove $X \in V$ e $X \to Y \in R$ è una regola della CFG. Due regole $X \to Y, X \to Z \in R$ possono essere accorpate con il simbolismo $X \to Y \mid Z$.

Siano u, v, w stringhe, e $A \to w \in R$; allora si dice che uAv **produce** uwv, denotato con

$$uAv \Rightarrow uwv$$

Date due stringhe u, v, se $u = v$, oppure esistono stringhe $u_1, \ldots, u_k$ con $k \geq 0$ tali che

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

si dice che u **deriva** v, ed è denotato con $u \overset{*}{\Rightarrow} v$.

Esempio 2.1.1.1 (CFG)**.** Un esempio di CFG è il seguente:

$$\begin{gathered} A \to \mathtt{0}A\mathtt{1} \\ A \to B \\ B \to \mathtt{\#} \end{gathered}$$

In essa, si hanno:

$$\begin{gathered} V := \{A, B\} \\ \Sigma := \{\mathtt{0}, \mathtt{1}, \mathtt{\#}\} \\ S := A \in V \end{gathered}$$

Da essa, è possibile ottenere ad esempio la stringa `000#111` attraverso le seguenti sostituzioni:

$$A \implies \mathtt{0}A\mathtt{1} \implies \mathtt{000}A\mathtt{111} \implies \mathtt{000}B\mathtt{111} \implies \mathtt{000\#111}$$

Definizione 2.1.1.4: Linguaggio di una grammatica

Data una grammatica G, il suo **linguaggio** è l'insieme delle stringhe che la grammatica G è in grado di generare, ed è denotato con $L(G)$. In simboli, data una grammatica G, si ha che

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

Definizione 2.1.1.5: CFL

Se G è una CFG, allora $L(G)$ è detto **linguaggio context-free**, o CFL (*Context-Free Language*).

Esempio 2.1.1.2 (CFL)**.** Si prenda in esame la CFG dell'Esempio 2.1.1.1, e sia essa G. Allora, in tale grammatica il linguaggio risulta essere

$$L(G) = \{\mathtt{0}^n\#\mathtt{1}^n \mid n \geq 0\}$$

Esempio 2.1.1.3 (Grammatiche e linguaggi)**.** I seguenti sono esempi di linguaggi, e corrispondenti grammatiche che li descrivono:

- il linguaggio $L_1 := \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w \text{ contiene almeno tre } \mathtt{1}\}$ è descritto dalla seguente grammatica:
$$\begin{gathered} S_1 \to X\mathtt{1}X\mathtt{1}X \\ X \to \varepsilon \mid \mathtt{0}X \mid \mathtt{1}X \end{gathered}$$
- il linguaggio $L_2 := \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w = w^{\mathcal{R}} \land |w| \equiv 0 \pmod 2\}$ è descritto dalla seguente grammatica:
$$S \to \mathtt{0}S\mathtt{0} \mid \mathtt{1}S\mathtt{1} \mid \varepsilon$$
- il linguaggio $L_3 := \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^{i+j} \mid i, j \geq 0\}$ è descritto dalla seguente grammatica:
$$\begin{gathered} S \to \mathtt{a}S\mathtt{c} \mid X \\ X \to \mathtt{b}S\mathtt{c} \mid \varepsilon \end{gathered}$$

Metodo 2.1.1.1: CFG di DFA

Dato un DFA $D = (Q, \Sigma, \delta, q_0, F)$, per ottenere una grammatica $G = (V, \Sigma, R, S)$ tale che $L(D) = L(G)$, è necessario:

- associare una variabile V_i, per ogni stato $q_i \in Q$, con $i \in [1, n]$ (assumendo $|Q| = n$);
- porre $S = V_0$;
- introdurre una regola $V_i \to aV_j$ se vale $\delta(q_i, a) = q_j$ per qualche coppia di stati $q_i, q_j \in Q$;
- aggiungere la regola $V_i \to \varepsilon$ se $q_i \in F$.

2.1.2 Ambiguità

Definizione 2.1.2.1: Derivazione a sinistra

Sia G una grammatica; una stringa si dice essere **derivata a sinistra** se è stata ottenuta applicando regole di G sulle variabili più a sinistra disponibili.

Esempio 2.1.2.1 (Derivazione a sinistra)**.** TODO

Definizione 2.1.2.2: Ambiguità

Sia G una grammatica; se esistono due stringhe u, v con $u \neq v$, tali che esiste una terza stringa z *derivata a sinistra* sia da u che da v attraverso le regole di G, si dice che G genera z **ambiguamente**.

Simmetricamente, G si dice essere **ambigua** se genera stringhe ambiguamente.

Esempio 2.1.2.2 (Stringhe generate ambiguamente)**.** Si consideri la seguente grammatica:

$$E \to E\texttt{+}E \mid E\texttt{*}E \mid \texttt{a}$$

Attraverso essa, è possibile ottenere ad esempio la seguente stringa, applicando la derivazione a sinistra:

$$E \Rightarrow E\texttt{*}E \Rightarrow E\texttt{+}E\texttt{*}E \Rightarrow \texttt{a+}E\texttt{*}E \Rightarrow \texttt{a+a*}E \Rightarrow \texttt{a+a*a}$$

Si noti però che è possibile ottenere questa stringa applicando anche la seguente derivazione a sinistra:

$$E \Rightarrow E\texttt{+}E \Rightarrow \texttt{a+}E \Rightarrow \texttt{a+}E\texttt{*}E \Rightarrow \texttt{a+a*}E \Rightarrow \texttt{a+a*a}$$

Dunque, la grammatica descritta risulta essere ambigua, poiché esistono due derivazioni a sinistra per la stringa `a+a*a`.

Definizione 2.1.2.3: Linguaggi inerentemente ambigui

Un linguaggio si dice essere **inerentemente ambiguo** se non esistono grammatiche non ambigue che lo possano generare.

2.1.3 Forma normale di Chomsky

Definizione 2.1.3.1: Forma normale di Chomsky

Una CFG (V, Σ, R, S) è in **forma normale di Chomsky**, o CNF (*Chomsky Normal Form*), se ogni regola è della forma

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

dove $A, B, C \in V$, $a \in \Sigma$, $S := A$, e la regola $S \to \varepsilon \in R$ è sempre ammessa, la quale prende il nome di **ε-regola**.

Dunque, la CNF non permette di

i) avere **regole unitarie**, ovvero regole della forma $X \to B \in R$, con $X, B \in V$;

ii) avere regole della forma $X \to u \in R$, con $u \in (V \cup \Sigma)^* - \Sigma$;

iii) avere regole della forma $X \to S \in R$, con $X \in V$.

Metodo 2.1.3.1: CNF di CFG

Sia $G = (V, \Sigma, R, S)$ una CFG; allora, è possibile rendere G in CNF attraverso i seguenti passaggi:

- vengono aggiunte la variabile $S_0 \in V$, e la regola $S_0 \to S \in R$, in modo da non avere S alla destra di nessuna regola in R;
- ogni regola $A \to \varepsilon \in R$ — con $A \in V - \{S\}$ — viene rimossa da R, e successivamente per ogni regola della forma $X \to uAv \in R$ — per qualche $X \in V$ e $u, v \in (V \cup \Sigma)^*$ — viene aggiunta $X \to uv \in R$; inoltre, se $X \to A \in R$, allora viene aggiunta $X \to \varepsilon \in R$, solo se quest'ultima non era stata precedentemente rimossa;
- ogni regola unitaria $A \to B \in R$ viene rimossa, e per ogni regola $B \to u \in R$ — con $u \in (V \cup \Sigma)^*$ — viene aggiunta $A \to u \in R$, solo se questa non era una regola unitaria precedentemente rimossa;
- ogni regola restante $A \to u_1 \ldots u_k$ con $k \geq 3$ — dove $u_1, \ldots, u_k \in (V \cup \Sigma)^*$ — viene rimpiazzata con le regole

$$\begin{aligned} A &\to u_1 A_1 \\ A_1 &\to u_2 A_2 \\ A_2 &\to u_3 A_3 \\ &\vdots \\ A_{k-2} &\to u_{k-1} u_k \end{aligned}$$

  dove $A_1, \ldots, A_{k-2} \in V$, creando dunque una *catena di sostituzioni*, al fine di avere al più 2 variabili o terminali alla destra delle regole;
- infine, ogni $u_i \in \Sigma$ (dunque terminale) — per ogni $i \in [1, k]$ — viene rimpiazzato con una nuova varibile $U_i \in V$, e viene aggiunta la regola $U_i \to u_i \in R$.

Lemma 2.1.3.1: CFL generati da CFG in CNF

Ogni CFL è generato da una CFG in forma normale di Chomsky.

Dimostrazione. Sia L un CFL generato da una CFG G; allora, attraverso il Metodo 2.1.3.1, è possibile rendere G in forma normale di Chomsky, e dunque segue la tesi. $\square$

Esempio 2.1.3.1 (Convertire CFG in CNF)**.** TODO

2.2 Automi a pila

2.2.1 Definizioni

Definizione 2.2.1.1: PDA

Un **PDA** (*Pushdown Automaton*) è un NFA dotato di uno **stack** illimitato, che gli consente di riconoscere alcuni linguaggi non regolari, poiché in esso è in grado di porre i simboli che legge dalla stringa di input, di fatto implementando un sistema di *memoria*.

Formalmente, un PDA è una sestupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, dove

- Q è l'**insieme degli stati**, un insieme *finito*
- Σ è l'**alfabeto dell'automa**, un insieme *finito*
- Γ è l'**alfabeto dello stack** (o *pila*), un insieme *finito*
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ è la **funzione di transizione**, che definisce la relazione tra gli stati
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'**insieme degli stati accettanti**

dove $\Gamma_\varepsilon := \Gamma \cup \{\varepsilon\}$.

Si noti che la macchina può usare differenti alfabeti per il suo input e la sua pila, infatti la definizione formale vede due alfabeti distinti, Σ e Γ rispettivamente. Inoltre, Γ_ε compare nel prodotto cartesiano del dominio di δ, poiché il simbolo in cima allo stack del PDA è in grado di determinare anch'esso la mossa seguente dell'automa; a tal proposito, $\varepsilon \in \Gamma$ permette di ignorare il primo elemento della pila. In aggiunta, Γ_ε compare anche all'interno dell'insieme potenza (si noti che un PDA è non deterministico) del codominio di δ, al fine di decidere se salvare il simbolo letto all'interno dello stack (tramite $\varepsilon \in \Gamma$ stesso). Dunque, un'operazione di un PDA sul suo stack può essere un *push*, un *pop*, o entrambe — ottenendo l'effetto di rimpiazzare l'elemento in cima allo stack.

Esempio 2.2.1.1 (PDA)**.** Un esempio di PDA $P = (Q, \Sigma, \Gamma, \delta, q_1, F)$ è il seguente:

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

e δ è data dalla seguente tabella di transizione:

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

Dunque, il suo diagramma di stato è il seguente:

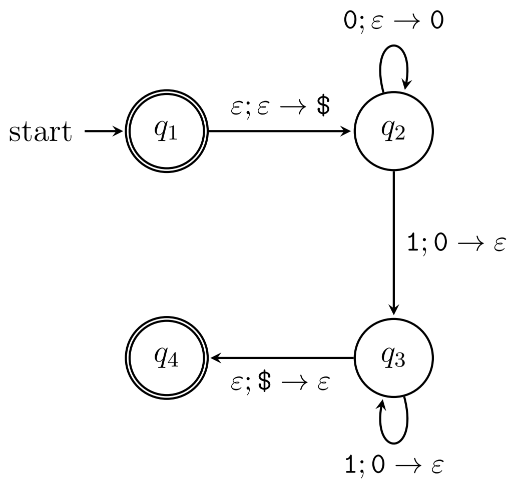


Figura 2.1: Il PDA P.

La notazione $a; b \to c$ presente sugli archi di questo diagramma sta ad indicare che se viene letto il simbolo a dall'input, M può sostituire b, se in cima al suo stack (attraverso un'operazione di *pop*) con c (mediante un'operazione di *push*). Si noti inoltre che ognuno dei simboli può essere ε, e dunque

- $\varepsilon; b \to c$ indica che il ramo viene eseguito senza attendere alcun simbolo di input (si noti l'Esempio 1.3.1.1 per il non determinismo)
- $a; \varepsilon \to c$ indica che, alla lettura di a, viene effettuato solamente il *push* di c nello stack
- $a; b \to \varepsilon$ indica che, alla lettura di a, viene effettuato solamente il *pop* di b dallo stack

Definizione 2.2.1.2: Stringhe accettate (PDA)

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ un PDA, e sia $w = w_1 \cdots w_n$ una stringa tale per cui $\forall i \in [1, n] \quad w_i \in \Sigma_\varepsilon$; allora, P **accetta** w se esistono una sequenza di stati $r_0, \ldots, r_n \in Q$ e una sequenza di stringhe $s_0, \ldots, s_n \in \Gamma^*$ tali per cui

- $r_0 = q_0$
- $s_0 = \varepsilon$, ovvero, lo stack è inizialmente vuoto
- $\forall i \in [0, n-1] \quad \exists a, b \in \Gamma_\varepsilon, t \in \Gamma^* \mid (r_{i+1}, b) \in \delta(r_i, w_{i+1}, a) \land \begin{cases} s_i = at \\ s_{i+1} = bt \end{cases}$, ovvero, M si muove correttamente in base allo stato, al simbolo nello stack, ed al prossimo simbolo di input; si noti che le stringhe $s_0, \ldots, s_n$ rappresentano di fatto il contenuto dello stack che M ha su un ramo accettante della computazione, infatti $s_i = at$ diventa $s_{i+1} = bt$ con l'iterazione successiva, dunque a è stato sostituito con b in cima alla pila
- $r_n \in F$

Osservazione 2.2.1.1: Stack vuoto

Un PDA non è in grado di controllare se il suo stack è vuoto, ma è possibile ottenere questo effetto come segue: si prenda in esame il PDA M dell'Esempio 2.2.1.1; è importante notare che esso utilizza il simbolo \$ per capire se lo stack è vuoto o meno, poiché viene inserito sin dall'inizio (si osservi l'etichetta $\varepsilon; \varepsilon \to \$$ sull'arco (q_1, q_2)), e dunque se viene letto \$ in cima allo stack, M sa che quello è l'unico elemento contenuto al suo interno, e lo stack è di fatto vuoto.

Osservazione 2.2.1.2: Fine dell'input

Un PDA non è in grado di controllare se è stata raggiunga la fine della stringa di input, ma è possibile ottenere questo effetto come segue: si prenda in esame il PDA M dell'Esempio 2.2.1.1; è importante notare che esso può sapere se è stata raggiunta la fine della stringa di input, poiché lo stato accettante q_4 può essere eventualmente raggiunto solamente alla lettura di ε e nel momento in cui è possibile rimuovere \$ dallo stack (si noti l'Osservazione 2.2.1.1).

Osservazione 2.2.1.3: Linguaggi non regolari e PDA

Si consideri il PDA dell'Esempio 2.2.1.1; esso opera come segue:

- viene posto $ all'interno dello stack;
- fintanto che viene letto 0, viene *pushato* 0 nello stack;
- non appena viene letto 1, e fintanto che viene letto 1, viene rimosso 0 dallo stack;
- se all'interno dello stack è presente $, allora l'automa accetta.

Dunque, di fatto, il PDA sta *contando* il numero di 0 e di 1 presenti all'interno della stringa, poiché la computazione avanza solamente se, per ognuno degli 1 letti, è possibile rimuovere uno 0 dallo stack. Infine, lo stato q_1 è accettante per poter riconoscere la stringa ε. Allora, l'automa riconosce il linguaggio

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

il quale non è regolare per l'Esempio 1.8.1.2. In simboli, si ha che

$$\mathsf{REG} \subsetneq \mathcal{L}(\mathsf{PDA})$$

Esempio 2.2.1.2 (PDA)**.** Si consideri il seguente PDA:

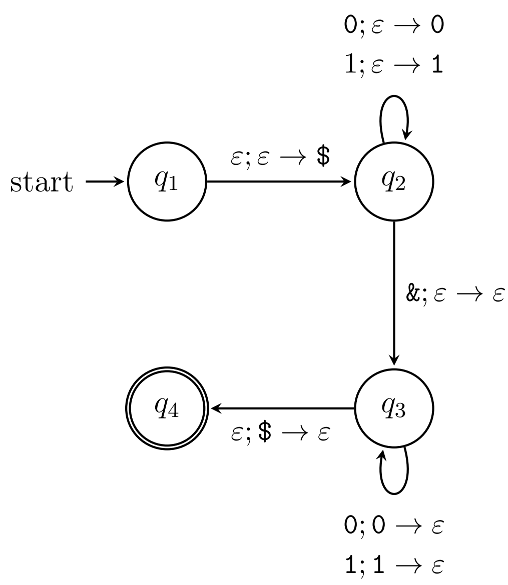


Figura 2.2: Un PDA.

Esso è in grado di riconoscere il linguaggio

$$L = \{w \in \{0, 1\}^* \mid w\&w^{\mathcal{R}}\}$$

poiché:

- pone `$` all'interno dello stack, per sapere quando quest'ultimo diventa vuoto;
- pone all'interno dello stack qualsiasi simbolo diverso da `&`;
- una volta letto il simbolo `&`, l'automa cambia stato lasciando lo stack intatto, ed inizia a rimuovere da questo ogni simbolo che viene letto, *solo se coincide con il simbolo posto sulla sua cima*; di fatto, questa tecnica controlla che i simboli che vengono letti dopo `&` siano l'*opposto* di come sono stati letti inizialmente;
- se trova `$` nello stack — e dunque quest'ultimo è vuoto — accetta.

2.2.2 Equivalenze

Definizione 2.2.2.1: Inserimento di stringhe nello stack

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ un PDA; è possibile introdurre una notazione per effettuare l'**inserimento di una stringa nello stack** di P, come segue: dati due stati $p, q \in Q$, per inserire una stringa $u_1 \cdots u_k \in \Gamma^*$ all'interno dello stack di P, devono esistere stati $r_1, \ldots, r_{k-1} \in Q$ tali per cui

$$\begin{gathered} \delta(p, a, u_1 \cdots u_k) \ni (r_1, u_k) \\ \delta(q_1, \varepsilon, \varepsilon) = \{(r_2, u_{k-1})\} \\ \vdots \\ \delta(r_{n-1}, \varepsilon, \varepsilon) = \{(q, u_1)\} \end{gathered}$$

In simboli, per indicare l'inserimento di una stringa all'interno dello stack di un PDA verrà utilizzato il seguente simbolismo

$$a; s \to xyz$$

per certi $a \in \Sigma_\varepsilon, s \in \Gamma, xyz \in \Gamma^*$.

Esempio 2.2.2.1 (Inserimenti di stringhe nello stack)**.** Si consideri il seguente PDA:

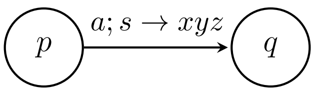


Figura 2.3: Un PDA che inserisce una stringa nello stack.

per certi $a \in \Sigma_\varepsilon, s \in \Gamma, xyz \in \Gamma^*$; dunque, con la notazione presente sull'arco (p, q) verrà sottointesa la seguente successione di stati — per certi stati $r_1, r_2 \in Q$:

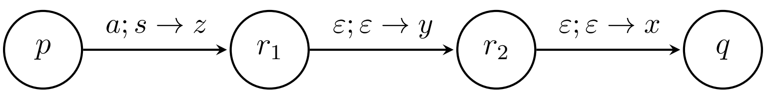


Figura 2.4: La formalizzazione del PDA precedente.

Metodo 2.2.2.1: PDA di CFG

Sia $G = (V, \Sigma, R, S)$ una CFG, e sia E l'insieme di stati tale da permettere di utilizzare la notazione descritta nella Definizione 2.2.2.1; per costruire un PDA in grado di riconoscere $L(G)$, che avrà la forma

$$P := (\{q_0, q', q\} \cup E, \Sigma, V \cup \Sigma, \delta, q_0, \{q\})$$

si definiscono i seguenti:

- $\delta(q_0, \varepsilon, \varepsilon) := \{(q', S\$)\}$, ovvero, viene inserito $\$$ come marcatore nello stack di P (si noti l'Osservazione 2.2.1.1), e successivamente la stringa iniziale S di G;
- $\forall A \in V \quad \delta(q', \varepsilon, A) := \{(q', w) \mid A \to w \in R, w \in (V \cup \Sigma)^*\}$ poiché, se viene incontrata una variabile A sulla cima dello stack di P, viene scelta una delle regole di G in grado di sostituire A (si noti che la sostituzione verrà effettuata non deterministicamente);
- $\forall a \in \Sigma \quad \delta(q', a, a) := \{(q', \varepsilon)\}$ per rimuovere i terminali dallo stack, evitando che possano essere ulteriormente rimpiazzati;
- $\delta(q', \varepsilon, \$) := \{(q, \varepsilon)\}$ per sfruttare il marcatore $\$$.

Il seguente è un diagramma che raffigura P (a meno degli stati in E):

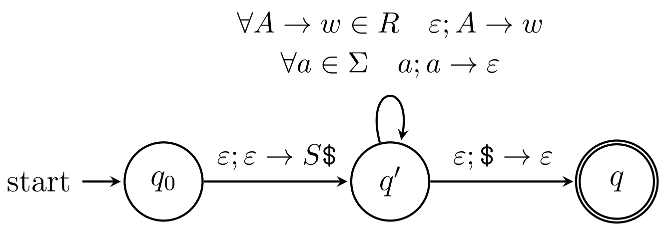


Figura 2.5: Il PDA P appena costruito.

Esempio 2.2.2.2 (PDA di CFG)**.** Si consideri la seguente CFG:

$$G: \begin{array}{c} S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b} \\ T \to T\mathtt{a} \mid \varepsilon \end{array}$$

Attraverso il Metodo 2.2.2.1, si ottiene il seguente PDA, in grado di riconoscere $L(G)$:

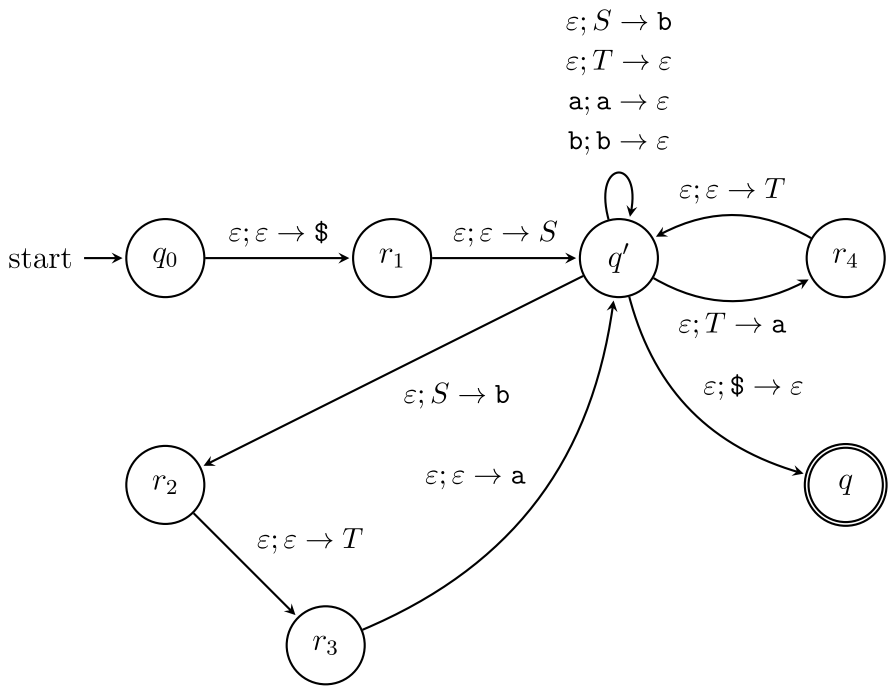


Figura 2.6: Un PDA in grado di riconoscere $L(G)$.

Teorema 2.2.2.1: CFL e PDA

Un linguaggio è context-free se e solo se esiste un PDA che lo riconosce; in simboli

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

Dimostrazione.

Prima implicazione. Sia L un CFL, e dunque per definizione esiste una CFG G tale che $L(G) = L$; allora, utilizzando il Metodo 2.2.2.1, è possibile trasformare G in un PDA ad essa equivalente — ovvero, tale da riconoscere $L(G)$ — dunque segue la tesi.

Seconda implicazione. Sia L un linguaggio riconosciuto da un PDA definito come $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ — dunque $L = L(P)$ — e si consideri il seguente PDA

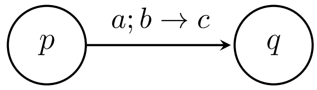
$$P' := (Q', \Sigma, \Gamma, \delta', q_0, F')$$

tale che:

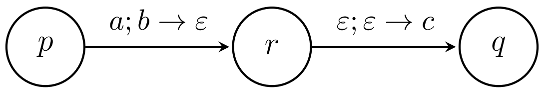
- ogni transizione di stati possa effettuare solamente un'operazione di *push* o *pop*, *non simultaneamente*, dunque avente δ' definita come segue:

  $$\forall p, q \in Q, a \in \Sigma, b, c \in \Gamma^* \quad (q, c) \in \delta(p, a, b) \implies \exists r \in Q' \mid \begin{cases} (r, \varepsilon) \in \delta'(p, a, b) \\ (q, c) \in \delta'(r, \varepsilon, \varepsilon) \end{cases}$$

  al fine di trasformare la seguente transizione



come segue:



e sia $Q_{\delta'}$ l'insieme di stati tali da permettere la non simultaneità appena descritta

- $Q' := Q \cup Q_{\delta'} \cup \{q_{\text{accept}}\}$
- $F' := \{q_{\text{accept}}\}$ sia costituito da un solo stato accettante, e dunque

$$\forall q \in F \quad (q_{\text{accept}}, \varepsilon) \in \delta'(q, \varepsilon, \varepsilon)$$

in modo da far terminare la computazione di ogni stringa accettata in q_{accept}

- venga svuotato lo stack prima di accettare qualsiasi stringa, di conseguenza

$$\forall q \in F, a \in \Sigma \quad (q, \varepsilon) \in \delta'(q, \varepsilon, a)$$

Si noti che, per costruzione di P', si ha che $L(P) = L(P')$, poiché P' è una versione di P in cui le transizioni possono effettuare solamente un'operazione per volta sullo stack, è presente un solo stato accettante e viene svuotato lo stack prima di accettare le stringhe.

Si consideri ora la CFG $G = (V, \Sigma, R, S)$ definita come segue:

- $V := \{A_{p,q} \mid p, q \in Q'\}$
- $S := A_{q_0, q_{\text{accept}}}$
- l'insieme di regole R è composto dall'unione dei seguenti:

$$\begin{gathered} R := \{A_{p,p} \to \varepsilon \mid p \in Q'\} \cup \\ \cup \{A_{p,q} \to aA_{r,s}b \mid p, q, r, s \in Q', a, b \in \Sigma_\varepsilon, \exists u \in \Gamma : (r, u) \in \delta'(p, a, \varepsilon), (q, \varepsilon) \in \delta(s, b, u)\} \cup \\ \cup \{A_{p,q} \to A_{p,r}A_{r,q} \in R \mid p, q, r \in Q'\} \end{gathered}$$

Dunque, il linguaggio di G è il seguente:

$$L(G) = \{w \in \Sigma^* \mid A_{q_0, q_{\text{accept}}} \overset{*}{\Rightarrow} w\}$$

La dimostrazione procederà ora provando che $L(G) = L(P')$. A tal scopo, sarebbe dunque sufficiente dimostrare che una stringa è accettata da P' se e solo se è derivabile, attraverso le regole di G, a partire da $A_{q_0, q_{\text{accept}}}$, ma è possibile generalizzare questa proposizione dimostrando che, per ogni coppia di stati $p, q \in Q'$, $A_{p,q}$ è in grado di derivare una stringa w se e solo se w porta P' dallo stato p — avendo lo stack vuoto — allo stato q — avendo ancora lo stack vuoto.

Prima implicazione. La dimostrazione procede per induzione sul numero di passaggi nella derivazione di w, a partire da $A_{p,q}$, attraverso le regole in R.

Caso base. Se la derivazione è composta solamente da 1 passaggio, allora è stata utilizzata una regola della forma $A_{p,q} \to u_1 \dots u_k$ dove $u_1, \dots u_k \in \Sigma^*$, dunque esclusivamente terminali. Si noti però che le uniche regole della grammatica che vedono solamente terminali sulla loro destra sono le regole del primo insieme di definizione di R, ovvero forma $A_{p,p} \to \varepsilon$ dunque, se P' si trova nello stato p, avendo lo stack vuoto, sicuramente ε è in grado di portare P' in p stesso, mantentendo vuoto lo stack.

Ipotesi induttiva forte. Dati due stati $p, q \in Q'$, se $A_{p,q}$ è in grado di derivare una stringa w, applicando al più k passaggi (con $k \geq 1$) allora w porta P' dallo stato p — avendo lo stack vuoto — allo stato q — avendo ancora lo stack vuoto.

Passo induttivo. È necessario dimostare che l'ipotesi induttiva sia ancora verificata per derivazioni costituite da $k+1$ passaggi. Si consideri dunque una stringa w tale che $A_{p,q} \overset{*}{\Rightarrow} w$ con $k+1$ passaggi; per costruzione di R, il primo passaggio può essere $A_{p,q} \Rightarrow aA_{r,s}b \overset{*}{\Rightarrow} x$ per certi stati $r, s \in Q'$ e terminali $a, b \in \Sigma_\varepsilon$, oppure $A_{p,q} \Rightarrow A_{p,r}A_{r,q} \overset{*}{\Rightarrow} x$ per un qualche stato $r \in Q'$, e dunque:

- nel primo caso, sia y la sottostringa di w generata da $A_{r,s}$, dunque $A_{r,s} \overset{*}{\Rightarrow} y \implies w = ayb$, e si noti che y è stata derivata con k passaggi; allora, per ipotesi induttiva forte, y porta P' dallo stato r — avendo lo stack vuoto — allo stato s — avendo ancora lo stack vuoto; inoltre, per costruzione di R, si ha che

  $$A_{p,q} \to aA_{r,s}b \in R \iff \exists u \in \Gamma \mid \begin{cases} (r, u) \in \delta'(p, a, \varepsilon) \\ (q, \varepsilon) \in \delta'(s, b, u) \end{cases}$$

  e dunque, assumendo che P' abbia lo stack vuoto:

  – se si trova nello stato p, leggendo a, P' può andare nello stato r, inserendo u nello stack;

  – se si trova nello stato r, leggendo y, P' può andare nello stato s, lasciando lo stack invariato — per ipotesi induttiva forte;

  – se si trova nello stato s, leggendo b, P' può andare nello stato q, rimuovendo u dallo stack.

  allora w è in grado di portare P' dallo stato p allo stato q, lasciando invariato lo stack.

- differentemente, nel secondo caso, siano y e z sottostringhe di w generate rispettivamente da $A_{p,r}$ ed $A_{r,q}$ dunque $\begin{cases} A_{p,r} \overset{*}{\Rightarrow} y \\ A_{r,q} \overset{*}{\Rightarrow} z \end{cases} \implies w = yz$, e si noti che y e z sono state derivate con $k+1$ passaggi; allora, per ipotesi induttiva forte, y e z portano, rispettivamente, P' dagli stati p

ed r — avendo lo stack vuoto — agli stati r e q — avendo ancora lo stack vuoto; allora, w è in grado di portare P' dallo stato p allo stato q, lasciando invariato lo stack;

Seconda implicazione. La dimostrazione procede per induzione sul numero di passaggi della computazione di w, da parte di P', tra gli stati p e q.

Caso base. Se la computazione è composta da 0 passaggi, allora inizia e finisce nello stesso stato, dunque inizia e termina in p stesso; inoltre, in 0 passaggi P' non può leggere nessun carattere, dunque $w = \varepsilon$; allora, per costruzione di R, si ha che $A_{p,p} \to \varepsilon \in R \implies A_{p,p} \overset{*}{\Rightarrow} w$.

Ipoesi induttiva forte. Dati due stati $p, q \in Q'$, se una stringa w porta P' dallo stato p — avendo lo stack vuoto — allo stato q — avendo ancora lo stack vuoto — attraverso al più k passaggi (con $k \geq 0$), allora $A_{p,q}$ è in grado di derivare w.

Passo induttivo. È necessario dimostrare che l'ipotesi induttiva sia ancora verificata per computazioni costituite da $k+1$ passaggi. Se lo stack deve essere vuoto sia all'inizio della computazione (dunque su p) sia al termine (dunuqe su q), allora o lo stack non si è mai svuotato durante l'intera computazione, oppure esistono alcuni passaggi in cui lo stack si svuota e riempie nuovamente, e dunque:

- nel primo caso, il primo simbolo che viene inserito all'interno dello stack deve necessariamente coincidere con l'ultimo che viene rimosso, e sia questo $u \in \Gamma$; siano inoltre:

  - $a \in \Sigma_\varepsilon$ il primo input letto, partendo da p;
  - $r \in Q'$ lo stato del secondo passaggio;
  - $s \in Q'$ lo stato del penultimo passaggio;
  - $b \in \Sigma_\varepsilon$ l'ultimo input letto, terminando in q;

  allora, segue che $(r, u) \in \delta'(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta'(s, b, u)$; si consideri ora la regola $A_{p,q} \to aA_{r,s}b \in R$ presente tra le regole di G, e sia y la sottostringa di w tale che $w = ayb$ (e dunque $A_{r,s} \overset{*}{\Rightarrow} y$); si noti che, se $w = ayb$, allora y non ha necessità di rimuovere u dallo stack, ed è dunque in grado di portare P' dallo stato r allo stato s lasciando lo stack invariato; allora, poiché la porzione di computazione di y è composta da $(k+1) - 2 = k - 1$ passaggi, è possibile applicare su essa l'ipotesi induttiva forte, per la quale $A_{r,s} \overset{*}{\Rightarrow} y$; allora si ha che

  $$A_{p,q} \Rightarrow aA_{r,s}b \overset{*}{\Rightarrow} ayb = w \implies Ap, q \overset{*}{\Rightarrow} w$$

- differentemente, nel secondo caso si assuma esistauno stato $r \in Q'$ tale per cui la computazione veda vuoto lo stack di P' in r ; allora, poiché la computazione da p a q è composta da $k+1$ passaggi, le computazioni da p ad r, e da r a q possono entrambe contenere al massimo k passaggi;

allora, chiamate rispettivamente $y, z \in \Sigma_\varepsilon^*$ gli input letti durante le due porzioni di computazione (si noti allora che $w = yz$), si ha che per ipotesi induttiva forte $A_{p,r} \overset{*}{\Rightarrow} y$ e $A_{r,q} \overset{*}{\Rightarrow} z$; dunque, poiché la regola $A_{p,q} \to A_{p,r}A_{r,q} \in R$ è presente tra le regole di G, si ha che

$$A_{p,q} \Rightarrow Ap, rA_{r,q} \overset{*}{\Rightarrow} yz = w$$

In particolare, $A_{q_0,q_{\text{accept}}}$ è in grado di derivare una stringa w se e solo se w porta P' dallo stato q_0 — avendo lo stack vuoto — allo stato q_{accept} — avendo ancora lo stack vuoto — o equivalentemente, P' accetta w. Allora, segue che

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

e dunque, poiché $L(G) = L(P') = L(P) = L$, segue la tesi.

$\square$

2.3 Linguaggi non context-free

2.3.1 Pumping lemma

Proposizione 2.3.1.1: Altezza di derivazioni su CFG in CNF

Sia $G = (V, \Sigma, R, S)$ una CFG in CNF, e $x \in L(G)$ una sua stringa; allora, se h è l'altezza all'albero di derivazione di x, si ha che

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

Dimostrazione. La dimostrazione procede per induzione sull'altezza h dell'albero di derivazione di x.

Caso base. Se $h = 1$, e dunque la derivazione è costituita da un solo passaggio, poiché G è in CNF in ipotesi, allora la regola applicata deve necessariamente essere della forma $S \to a \in R$ con $a \in \Sigma$, e dunque x è costituita esclusivamente da un singolo simbolo (terminale); allora, poiché

$$|x| \leq 2^{h-1} = 2^{1-1} = 2^0 = 1$$

la tesi è verificata.

Ipotesi induttiva forte. Data una stringa $x \in L(G)$ il cui albero di derivazione abbia altezza al più h, è vero che $|x| \leq 2^{h+1}$

Passo induttivo. È necessario dimostrare che la tesi è verificata per ogni stringa $x \in L(G)$ il cui albero di derivazione abbia altezza $h + 1$. Si noti che, poiché G è in CNF, il primo passaggio dell'albero di derivazione di x deve necessariamente essere stato ottenuto attraverso una regola della forma $S \to AB$ per qualche $aA, B \in V$, e dunque devono esistere y, z sottostringhe di $x = yz$ — e dunque $|x| = |y| + |z|$ —

tali che $A \overset{*}{\Rightarrow} y$ e $B \overset{*}{\Rightarrow} z$. Poiché la derivazione $S \overset{*}{\Rightarrow} x$ ha altezza $h+1$, allora le derivazioni di y e di z devono avere altezza h, e dunque per essi è possibile applicare l'ipotesi induttiva forte, e dunque si verifica che

$$\begin{cases} |y| \leq 2^{h-1} \\ |z| \leq 2^{h-1} \end{cases} \implies |x| = |y| + |z| \leq 2^{h-1} + 2^{h-1} = 2^h = 2^{(h+1)-1}$$

allora segue la tesi.

□

Lemma 2.3.1.1: Pumping lemma (CFL)

Sia A un CFL; allora, esiste un $p \in \mathbb{N}$, detto **lunghezza del pumping**, tale che per ogni stringa $s \in A$ tale per cui $|s| \geq p$, esistono 5 stringhe $u, v, x, y, z \mid s = uvxyz$ soddisfacenti le seguenti condizioni:

- $\forall i \geq 0 \quad uv^ixy^iz \in A$
- $|vy| > 0$ (o, equivalentemente, $v \neq \varepsilon \vee y \neq \varepsilon$)
- $|vxy| \leq p$

Dimostrazione. Poiché A è un CFL, per definizione esiste una CFG $G = (V, \Sigma, R, S)$ che lo genera, e si assuma — senza perdita di generalità, per il Metodo 2.1.3.1 — che G sia in CNF; sia allora $p := 2^{|V|}$. Inoltre, sia $s \in A \mid |s| \geq p$, e dunque per la Proposizione 2.3.1.1, poiché G è in CNF, si ha che

$$p := 2^{|V|} \leq |s| \leq 2^{h-1} \implies 2^{|V|} \leq 2^{h-1} \iff |V| + 1 \leq h$$

dove h è l'altezza dell'albero di derivazione di s.

TODO DA FINIRE □

Esempio 2.3.1.1 (Pumping lemma in CFL)**.** Si consideri il linguaggio

$$L := \{0^n 1^n 2^n \mid n \in \mathbb{N}\}$$

e per assurdo, sia $L \in$ CFL, e dunque per esso vale il Lemma 2.3.1.1. Allora, sia $p \in \mathbb{N}$ la lunghezza del pumping di L, e si consideri la seguente stringa

$$w := 0^p 1^p 2^p \implies |w| = 3p > p$$

avente lunghezza sicuramente maggiore di p. Siano inoltre u, v, x, y, z tali da soddisfare il pumping lemma, ed in particolare $|vxy| \leq p$, ma poiché $w := 0^p 1^p 2^p = uv^ixy^iz$ per ogni $i \in \mathbb{N}$, allora si può verificare solo uno dei seguenti:

- vxy contiene solamente 0, dunque v^0xy^0 contiene solamente 0
- vxy contiene solamente 1, dunque v^0xy^0 contiene solamente 1
- vxy contiene solamente 2, dunque v^0xy^0 contiene solamente 2

- vxy contiene soltanto $\mathtt{0}$ e $\mathtt{1}$, dunque v^0xy^0 può contenere solo $\mathtt{0}$, solo $\mathtt{1}$, o sia $\mathtt{0}$ che $\mathtt{1}$
- vxy contiene soltanto $\mathtt{1}$ e $\mathtt{2}$, dunque v^0xy^0 può contenere solo $\mathtt{1}$, solo $\mathtt{2}$, o sia $\mathtt{1}$ che $\mathtt{2}$

e dunque, indipendentemente dalla scelta di v, x, y TODO DA FINIRE IL FINALE NON HA SENSO

Esempio 2.3.1.2 (Pumping lemma in CFL)**.** Si consideri il linguaggio

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

e per assurdo, sia $L \in \mathsf{CFL}$, e dunque per esso vale il Lemma 2.3.1.1. Allora, sia $p \in \mathbb{N}$ la lunghezza del pumping di L, e si consideri la seguente stringa

$$w := \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p\mathtt{1}^p \implies |w| = 4p > p$$

avente sicuramente lunghezza maggiore di p. Siano inoltre u, v, x, y, z tali da soddisfare il pumping lemma, ed in particolare $|vxy| \le p$, ma poiché $w := \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p\mathtt{1}^p = uv^ixy^iz$ per ogni $i \in \mathbb{N}$, allora si può verificare solo uno dei seguenti:

- vxy contiene solamente $\mathtt{0}$ o solamente $\mathtt{1}$, e dunque assumendo che ad esempio si trovi nella prima porzione di $\mathtt{0}$ di w, per ogni $i > 1$ si ha che la stringa diventa della forma
$$w := \mathtt{0}^{p+k}\mathtt{101}$$
per qualche k, e dunque non può essere della forma $w'w'$ con $w' \in \{\mathtt{0}, \mathtt{1}\}^*$;
- vxy contiene sia $\mathtt{0}$ che $\mathtt{1}$, e dunque vxy si trova a cavallo tra due sequenze di $\mathtt{0}$ ed $\mathtt{1}$; allora, assumendo che ad esempio si trovi nella prima regione di w che comprende sia $\mathtt{0}$ che $\mathtt{1}$, per ogni $i > 1$, si ha che la stringa diventa della forma
$$w := \mathtt{0}^{p+j}\mathtt{1}^{p+h}\mathtt{0}^p\mathtt{1}^p$$
per qualche j ed h, e dunque non può essere della forma $w'w'$ con $w' \in \{\mathtt{0}, \mathtt{1}\}^*$.

Allora, in nessuno dei casi possibili $w := uv^ixy^iz \in L$ $\lightning$.

Osservazione 2.3.1.1: Condizioni del pumping lemma (CFL)

TODO

2.4 Operazioni context-free

2.4.1 Unione

Proposizione 2.4.1.1: Chiusura sull'unione (CFL)

Siano $L_1, \ldots, L_n$ dei CFL; allora $\displaystyle\bigcup_{i=1}^{n} L_i$ è un CFL.

Dimostrazione. Poiché $L_1, \ldots, L_n$ sono CFL, allora esistono delle CFG $G_1, \ldots, G_n$ tali da generare rispettivamente ogni linguaggio in ipotesi; siano queste grammatiche definite come

$$\forall i \in [1, n] \quad G_i = (V_i, \Sigma_i, R_i, S_i)$$

Per dimostrare la tesi, è necessario dimostrare che esiste una CFG G tale che

$$L(G) = \bigcup_{i=1}^{n} L(G_i)$$

poiché $L(G) \in \mathsf{CFL}$ per definizione; allora, si consideri la seguente CFG:

$$G = \left(\bigcup_{i=1}^{n} V_i \cup \{S\}, \bigcup_{i=1}^{n} \Sigma_i, \bigcup_{i=1}^{n} R_i \cup \{S \to S_1 \mid \ldots \mid S_n\}, S \right)$$

e si noti che

$$w \in \bigcup_{i=1}^{n} L(G_i) \iff \exists j \in [1, n] \mid \begin{cases} w \in L(G_j) \iff S_j \overset{*}{\Rightarrow} w \\ S \to S_j \in R \end{cases} \iff S \overset{*}{\Rightarrow} w \iff w \in L(G)$$

Allora, segue la tesi. □

Esempio 2.4.1.1 (Unione di CFL)**.** Siano L_1 ed L_2 due CFL generati rispettivamente dalle seguenti CFG G_1 e G_2:

$$\begin{aligned} G_1 &: S_1 \to \mathtt{0} S_1 \mathtt{1} \mid \varepsilon \\ G_2 &: S_2 \to \mathtt{1} S_2 \mathtt{0} \mid \varepsilon \end{aligned}$$

Allora, la seguente CFG G:

$$G: \quad \begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to \mathtt{0} S_1 \mathtt{1} \mid \varepsilon \\ S_2 &\to \mathtt{1} S_2 \mathtt{0} \mid \varepsilon \end{aligned}$$

è in grado di generare $L_1 \cup L_2$.

2.4.2 Concatenazione

Proposizione 2.4.2.1: Chiusura sulla concatenazione (CFL)

TODO

Dimostrazione. Omessa. □

2.4.3 Star

Proposizione 2.4.3.1: Chiusura sull'operazione star (CFL)

TODO.

Dimostrazione. Omessa. □

2.4.4 Intersezione

Proposizione 2.4.4.1: CFL non è chiuso sull'intersezione

Siano L_1 ed L_2 due CFL; allora $L_1 \cap L_2$ non è necessariamente un CFL.

Dimostrazione. Siano G_1 e G_2 le due seguenti CFG:

$$G_1 : \begin{array}{l} S \to TU \\ T \to 0T1 \mid \varepsilon \\ U \to 2U \mid \varepsilon \end{array} \qquad G_2 : \begin{array}{l} S \to UT \\ T \to 1T2 \mid \varepsilon \\ U \to 0U \mid \varepsilon \end{array}$$

Si noti che la prima grammatica, partendo da S, compone stringhe costituite da TU, dove T è solo in grado di diventare una combinazione di $0T1$ o terminare — e dunque produce stringhe della forma 0^n1^n — mentre U è solo in grado di diventare una combinazione di $2U$ o terminare — e dunque produce stringhe della forma 2^k; allora, il suo linguaggio è

$$L(G_1) = \{0^n1^n2^k \mid n, k \in \mathbb{N}\}$$

Per ragionamento analogo, è possibile verificare che le stringhe che genera G_2 appartengono al linguaggio

$$L(G_2) = \{0^k1^n2^n \mid n, k \in \mathbb{N}\}$$

Infine, si noti che

$$L(G_1) \cap L(G_2) = \{0^n1^n2^n\}$$

poiché il primo linguaggio contiene le stringhe aventi stesso numero di 0 e di 1, ma arbitrario numero di 2, mentre il secondo contiene le stringhe aventi stesso numero di 1 e 2, ma arbitrario numero di 0, dunque la loro intersezione deve necessariamente essere il linguaggio composto da stringhe aventi stesso numero di 0, 1 e 2. Si noti però che, come dimostrato nell'Esempio 2.3.1.1, $\{0^n1^n2^n\} \notin$ CFL. Allora, poiché $L(G_1), L(G_2) \in$ CFL per loro stessa definizione, segue la tesi. □

2.4.5 Complemento

Proposizione 2.4.5.1: CFL non è chiuso sul complemento

Sia L un CFL; allora $\neg L$ non è necessariamente un CFL.

Dimostrazione. Sia L il linguaggio del Esempio 2.3.1.2, e si consideri il suo complemento

$$L := \{ww \mid w \in \{0,1\}^*\} \iff \neg L = \{0,1\}^* - \{ww \mid w \in \{0,1\}^*\}$$

e si consideri la seguente grammatica G:

$$G : \begin{array}{l} S \to A \mid B \mid AB \mid BA \\ A \to 0 \mid 0A0 \mid 0A1 \mid 1A0 \mid 1A1 \\ B \to 1 \mid 0B0 \mid 0B1 \mid 1B0 \mid 1B1 \end{array}$$

e si noti che le stringhe derivate da A sono tutte le stringhe aventi lunghezza dispari ed uno $\mathtt{0}$ al centro, mentre le stringhe derivate da B sono tutte le stringhe aventi lunghezza dispari ed un $\mathtt{1}$ al centro.

Dunque, si ha che:

- presa una stringa $x \in \neg L$, e ponendo $n := |x|$, se n è dispari allora
  - se x ha $\mathtt{0}$ al centro, allora $A \overset{*}{\Rightarrow} x$
  - se ha un $\mathtt{1}$ al centro, allora $B \overset{*}{\Rightarrow} x$

  diversamente, se n è pari, allora sia i tale che $x_i \neq x_{\frac{n}{2}+i}$ — si noti che i deve necessariamente esistere, poiché se non esistesse allora vorrebbe dire che $\forall i \in [1, n] \quad x_i = x_{\frac{n}{2}+1} \iff \exists w \in \{\mathtt{0}, \mathtt{1}\}^* \mid x = ww \iff x \in L$; siano inoltre

  $$\begin{aligned} u := x_1 \dots x_{2i-1} &\implies |u| = 2i - 1 - 1 + 1 \\ v := x_{2i} \dots x_n &\implies |v| = n - 2i + 1 \end{aligned}$$

  due sottostringhe di $x = uv$ di lunghezza dispari — si noti che $n - 2i + 1$ è dispari poiché n è pari in ipotesi; allora, i due caratteri centrali di u e v saranno rispettivamente

  $$\begin{aligned} m(u) &= x_{\frac{2i-1+1}{2}} = x_i \\ m(v) &= x_{\frac{n+2i}{2}} = x_{\frac{n}{2}+1} \end{aligned}$$

  allora, poiché $x_i \neq x_{\frac{n}{2}+1}$, e questi sono proprio i centri di due sottostringhe di x, aventi lunghezza dispari, si ha che x può essere generata attraverso le regole di G, poiché in essa sono presenti le regole $S \to AB \mid BA$; questo dimostra che $x \in \neg L \implies S \overset{*}{\Rightarrow} x$;

- presa una stringa x tale che $S \overset{*}{\Rightarrow} x$, e ponendo $n := |x|$, se n è dispari allora sicuramente $x \notin L \iff x \in \neg L$; differentemente, se n è pari, allora la stringa è stata generata attraverso una delle regole $S \to AB \mid BA$ necessariamente, e dunque — senza perdita di generalità — assumendo che x sia stata ottenuta attraverso la regola $S \to AB$, devono esistere $u, v \in \{\mathtt{0}, \mathtt{1}\}^* \mid x = uv \land A \overset{*}{\Rightarrow} u \land B \overset{*}{\Rightarrow} v$, dove u e v sono due stringhe aventi lunghezza dispari; siano allora $l := |u| \implies |v| = n - l$, ottenendo che

  $$m(u) = x_{\frac{l+1}{2}} = u_{\frac{l+1}{2}} = \mathtt{0} \neq \mathtt{1} = v_{\frac{n-l+1}{2}} = x_{\frac{n-l+1}{2}+l} = x_{\frac{n+l+1}{2}} = m(v)$$

  allora, poiché $m(u)$ ed $m(v)$ devono necessariamente trovarsi l'uno nella prima metà di x, l'altro nella seconda metà (indipendentemente dalla scelta di u e v) si ha che la stringa x è necessariamente costituita da due stringhe $w, w' \in \{\mathtt{0}, \mathtt{1}\}^* \mid w \neq w'$, poiché differiscono per almeno un carattere, ovvero $m(u) \neq m(v)$, di conseguenza $x \notin \iff x \in \neg L$; questo dimostra che $S \overset{*}{\Rightarrow} x \implies x \in \neg L$.

Dunque, poiché $S \overset{*}{\Rightarrow} x \iff x \in \neg L$, si ha che

$$L(G) := \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid S \overset{*}{\Rightarrow} x\} = \neg L \implies \neg L \in \mathsf{CFL}$$

e dunque segue la tesi. $\square$

3

Decidibilità

3.1 Macchine di Turing

3.1.1 Definizioni

Definizione 3.1.1.1: TM

Una TM (*Turing Machine*) è un automa fornito di un nastro (o *tape*) illimitato costituito da celle sovrascrivibili — sul quale viene posto anche l'input stesso — e di una testina che punta alla cella corrente della computazione, dove quest'ultima è in grado di muoversi liberamente sia verso destra che verso sinistra; inoltre, l'automa è caratterizzato da un solo stato di accettazione, ed un solo stato di rifiuto, che hanno *effetto immediato* (dunque la computazione termina non appena viene raggiunto lo stato di accettazione o di rifiuto; si noti che le macchine di Turing sono in grado di sovrascrivere ogni cella, anche quelle dell'input fornito, dunque questa caratteristica degli stati di accettazione e rifiuto ha significato).

Formalmente, una TM è una 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$ dove

- Q è l'**insieme degli stati**, un insieme *finito*
- Σ è l'**alfabeto dell'automa**, un insieme *finito*, tale che $\sqcup \notin \Sigma$
- Γ è l'**alfabeto del nastro**, un insieme *finito* tale che $\Sigma \subseteq \Gamma$ e $\sqcup \in \Gamma$
- $\delta : (Q - \{q_{\mathrm{accept}}, q_{\mathrm{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ è la **funzione di transizione**, che definisce la relazione tra gli stati
- $q_0 \in Q$ è lo **stato iniziale**
- $q_{\mathrm{accept}} \in Q$ è lo **stato di accettazione**
- $q_{\mathrm{reject}} \in Q$ è lo **stato di rifiuto**, tale che $q_{\mathrm{accept}} \neq q_{\mathrm{reject}}$

Si noti che il nastro di una TM può essere limitato a destra, limitato a sinistra, o illimitato da entrambe le estremità, poiché è possibile dimostrare l'equivalenza delle 3 tipologie di nastri descritti; dunque, all'interno di questi appunti, a meno di specifica, si assume che la TM in questione è costituita da un nastro limitato a sinistra.

Inizialmente, il nastro di una macchina di Turing contiene solamente la stringa di input, e tutto il resto è vuoto, dunque la TM riceve una stringa di input $w = w_1 \dots w_n \in \Sigma^*$ sulle n celle più a sinistra del nastro, e le celle restanti contengono il simbolo $\sqcup$ (letto "*blank*").

Per la computazione delle TM, si noti la segnatura della funzione di transizione δ: essa considera lo stato corrente ed il carattere che si trova sulla cella puntata dalla testina, e restituisce il prossimo stato, il carattere con cui sovrascrivere la cella corrente, e una direzione da prendere, indicata con L (*left*, e dunque la testina si sposterà a sinistra) o R (*right*, e dunque la testina si sposterà a destra). Se la testina si trova sulla prima cella — quella più a sinistra, nell'assunzione considerata — e prova a spostarsi a sinistra, essa non effettuerà alcuno spostamento. Si noti che, se la TM non raggiunge mai lo stato di accettazione o di rifiuto, la macchina non può terminare e la computazione proseguirà per sempre (*looping*).

Esempio 3.1.1.1 (TM). Un esempio di TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ è il seguente:

- $Q = \{q_{\text{accept}}, q_{\text{reject}}, q_0, q_1, q_2, q_3\}$
- $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
- $\Gamma = \{\sqcup, \mathtt{0}, \mathtt{1}, \mathtt{x}, \mathtt{y}\}$

e δ segue dal suo diagramma:

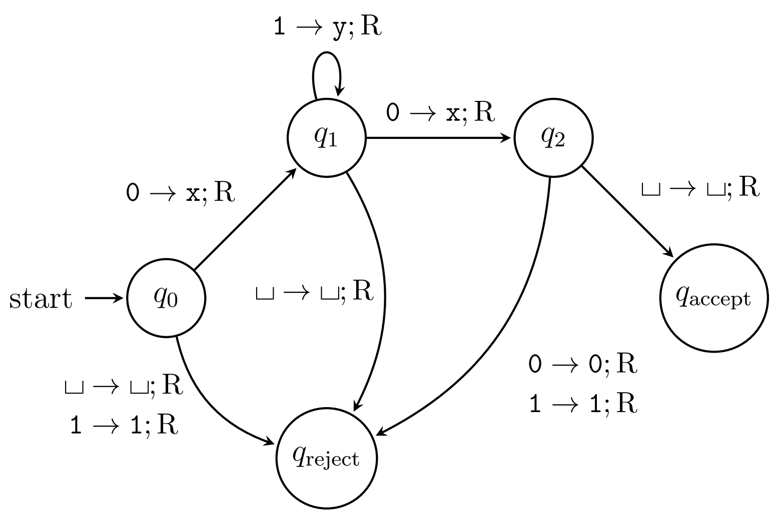


Figura 3.1: La TM M.

La notazione $a \to b; c$ presente sugli archi di questo diagramma sta ad indicare che viene letto il simbolo $a \in \Gamma$ sulla cella puntata correntemente dalla testina, questo viene rimpiazzato col simbolo $b \in \Gamma$, e la testina si sposta verso $c \in \{\mathrm{L}, \mathrm{R}\}$. Dunque, per non sovrascrivere la cella corrente, è sufficiente porre $a \to a; c$ come etichetta dell'arco in questione.

Si noti che, all'interno dei diagrammi delle TM, lo stato di rifiuto, e gli archi in esso entranti, sono generalmente omessi per brevità, implicitando che ogni stato q che non presenta transizioni esplicitamente rappresentate descrive in realtà archi (q, q_{reject}).

3.1.2 Configurazioni di TM

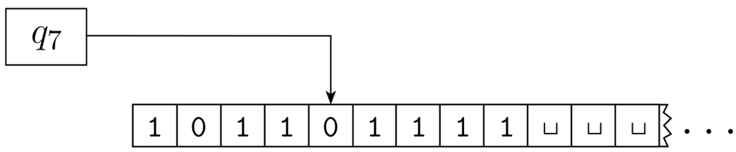
Definizione 3.1.2.1: Configurazione (TM)

Sia $M = (Q, \Sigma, \Gamma, q_0, q_{\text{accept}}, q_{\text{reject}})$ una TM; con il simbolismo

$$u \; q \; v$$

per certi $u, v \in \Gamma^*, q \in Q$ si denota una **configurazione** di M, dove q rappresenta lo stato attuale della computazione di un certo input, uv è la stringa che descrive il nastro (si assume che, dopo l'ultimo simbolo di v, il nastro contenga solo $\sqcup$), e la posizione della testina è sul primo simbolo di v.

Esempio 3.1.2.1 (Configurazioni di TM)**.** Data la seguente TM



si ha che la sua configurazione è

$$\texttt{1011} \; q_7 \; \texttt{01111}$$

Definizione 3.1.2.2: Produzione di configurazioni

Siano C_1 e C_2 due configurazioni di una certa macchina di Turing descritta dalla 7-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$; si dice che C_1 **produce** C_2 se e solo se M può passare da C_1 a C_2 in un unico passo. In simboli

$$\begin{array}{c} ua \; q_i \; bv \quad \text{produce} \quad u \; q_j \; acv \iff \delta(q_i, b) = (q_j, c, \mathrm{L}) \\ ua \; q_i \; bv \quad \text{produce} \quad uac \; q_j v \iff \delta(q_i, b) = (q_j, c, \mathrm{R}) \end{array}$$

per certi $u, v \in \Gamma^*$, $a, b, c \in \Gamma$ e $q_i, q_j \in Q$.

Osservazione 3.1.2.1: Configurazioni particolari

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ una TM; se la TM si trova sull'estremità sinistra del nastro, dunque sul primo carattere, la configurazione corrente è $q_i\ bv$ per certi $q_i \in Q$ e $b \in \Gamma, v \in \Gamma^*$; dunque, se si verifica una transizione che comporta una mossa a sinistra ed un rimpiazzo sel simbolo b con $c \in \Gamma$, per quando detto nella Definizione 3.1.1.1, si ha che la prossima configurazione di M sarà $q_j\ cv$ per qualche $q_j \in Q$. Se invece la TM si trova sull'estremità destra — dunque al termine dell'input fornito sul nastro — la configurazione corrente è $ua\ q_i\ \sqcup$ per certi $q_i \in Q$ e $a \in \Gamma, u \in \Gamma^*$, ma per brevità il simbolo $\sqcup$ verrà omesso, assumendo che i simboli $\sqcup$ seguano la parte del nastro rappresentata dalla configurazione.

Dunque, dato un input w, posto sul nastro, la *configurazione iniziale* di M è denotata con $q_0\ w$, mentre le *configurazioni di accettazione e rifiuto* — dette anche *configurazioni di arresto*, poiché non producono ulteriori configurazioni — sono descritte rispettivamente con q_{accept} e q_{reject}.

Definizione 3.1.2.3: Stringhe accettate (TM)

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ una TM, e sia $w = w_1 \cdots w_n$ una stringa tale per cui $\forall i \in [1, n] \quad w_i \in \Sigma$; allora, M **accetta** w se esiste una sequenza di configurazioni $C_0, \ldots, C_n$ tali per cui

- $C_0 = q_0\ w$
- $\forall i \in [0, n-1] \quad C_i$ produce C_{i+1}
- $C_n = q_{\text{accept}}$

3.1.3 Turing-riconoscibilità

Definizione 3.1.3.1: Turing-riconoscibilità

Un linguaggio è detto **Turing-riconoscibile** (o **ricorsivamente enumerabile**) se esiste una macchina di Turing che lo riconosce. La classe dei linguaggi Turing-riconoscibili è denotata con REC, dunque in simboli

$$L \in \mathsf{REC} \iff \exists M \in \mathsf{TM} \mid L(M) = L$$

Esempio 3.1.3.1 (Linguaggi Turing-riconoscibili)**.** Si consideri la TM M descritta all'interno dell'Esempio 3.1.1.1; si noti che essa, per sua costruzione, è in grado di riconoscere il linguaggio L descritto dall'espressione regolare `01*0`, poiché si comporta come segue:

- partendo dallo stato iniziale q_0, se viene letto uno `0`, la TM lo sovrascrive con il carattere `x`, e si sposta a destra, andando nello stato q_1;
- se ora viene letto un `1`, la TM lo sovrascrive con il carattere `y`, e si sposta ancora a destra, ma rimane nello stato q_1;

- se ora viene letto uno `0`, la TM lo sovrascrive con il carattere `x`, e si sposta ancora a destra, andando nello stato q_2;
- infine, se viene letto $\sqcup$, la macchina non sovrascrive il carattere, si sposta a destra (la direzione è irrilevante in questo caso) e termina accettando l'input — dunque andando nello stato q_{accept};

In ogni altro caso, la macchina rifiuta immediatamente.

Allora, poiché esiste una TM M tale per cui $L(M) = L$, si ha che $L \in \mathsf{REC}$.

Esempio 3.1.3.2 (Linguaggi Turing-riconoscibili)**.** TODO

3.1.4 Turing-decidibilità

Definizione 3.1.4.1: Decisore

Una macchina di Turing è detta **decisore** se termina sempre, dunque se accetta o rifiuta per qualsiasi input, non andando mai in *loop*.

Osservazione 3.1.4.1: Decisori

Data una TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, e ponendo

$$L(M) := \{w \in \Sigma^* \mid M \text{ accetta } w\}$$
$$R(M) := \{w \in \Sigma^* \mid M \text{ rifiuta } w\}$$

si ha che, in generale

$$L(M) \cup R(M) \subseteq \Sigma^*$$

poiché M potrebbe andare in loop, e dunque

$$M \text{ decisore } \iff L(M) \cup R(M) = \Sigma^*$$

Definizione 3.1.4.2: Turing-decidibilità

Un linguaggio è detto **Turing-decidibile** (o **ricorsivo**, o semplicemente **decidibile**) se esiste un decisore che lo riconosce. La classe dei linguaggi Turing-decidibili è denotata con DEC.

Simmetricamente, se un linguaggio L è Turing-decidibile per qualche decisore M, allora si dice che M **decide** L.

Osservazione 3.1.4.2: Turing-decidibilità

Si noti che $\mathsf{DEC} \subseteq \mathsf{REC}$, poiché un linguaggio Turing-decidibile è sicuramente Turing-riconoscibile, per definizione stessa.

3.2 Varianti di macchine di Turing

3.2.1 Macchine di Turing con testina ferma

Definizione 3.2.1.1: Macchina di Turing con testina ferma

Una **macchina di Turing con testina ferma** è una 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ dove ogni elemento della 7-tupla rimane invariato rispetto alla Definizione 3.1.1.1, a meno della funzione di transizione δ, definita come segue:

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$$

in cui il simbolo S (*stay*) indica che la testina della TM non si muove.

Proposizione 3.2.1.1: Equivalenza delle TM con testina ferma

Sia M una macchina di Turing con testina ferma; allora esiste una TM M' ad essa equivalente.

Dimostrazione.

Prima implicazione. Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ una macchina di Turing con testina ferma; di conseguenza, nel prodotto cartesiano del codominio di δ è presente l'insieme $\{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$, e dunque per costruire una macchina di Turing $M' = (Q', \Sigma, \Gamma, \delta', q'_0, q'_{\text{accept}}, q'_{\text{reject}})$ equivalente ad M, è sufficiente inserire nuovi stati in Q' tali che

$$\forall p, q \in Q, a, b \in \Gamma \quad \delta(p, a) = (q, b, \mathrm{S}) \implies \exists r \in Q' \mid \begin{cases} \delta'(p, a) = (r, b, \mathrm{R}) \\ \delta'(r, \gamma) = (p, \gamma, \mathrm{L}) \end{cases}$$

per un certo $\gamma \in \Gamma$, dove $r \in Q'$ è il nuovo stato, dunque facendo spostare la testina in una direzione — in questo caso, verso destra — e poi facendola spostare nuovamente nella direzione opposta — dunque in questo caso, verso sinistra.

Seconda implicazione. Sia $M' = (Q', \Sigma, \Gamma, \delta', q'_0, q'_{\text{accept}}, q'_{\text{reject}})$ una TM; allora, la segnatura della funzione di transizione di M' segue dalla Definizione 3.1.1.1, dunque nel prodotto cartesiano del suo codominio è presente l'insieme $\{\mathrm{L}, \mathrm{R}\}$, e poiché

$$\{\mathrm{L}, \mathrm{R}\} \subseteq \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$$

allora è possibile considerare M' stessa come macchina di Turing con testina ferma, che però non presenta mai il simbolo S all'interno delle transizioni.

□

3.2.2 Macchine di Turing multinastro

Definizione 3.2.2.1: Macchina di Turing multinastro

Una **macchina di Turing multinastro** è una 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ dove ogni elemento della 7-tupla rimane invariato rispetto alla Definizione 3.1.1.1, a meno della funzione di transizione δ, definita come segue:

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{\text{L}, \text{R}, \text{S}\}^k$$

dove $k \geq 1$ è il numero di nastri — e di testine — della macchina.

Osservazione 3.2.2.1: Operazione di *shift* sul nastro

Le macchine di Turing sono in grado di effettuare operazioni di *shift* di 1 cella a destra o a sinistra dell'intero nastro, semplicemente scansionando tutto quest'ultimo, e rimpiazzando ogni carattere con quello letto precedentemente.

Osservazione 3.2.2.2: Inizio del nastro

Una TM non è in grado di sapere se la sua testina si trova all'inizio dell'input, ma è possibile ottenere questo comportamento effettuando uno *shift* a destra dell'intero nastro (si noti l'Osservazione 3.2.2.1), e successivamente ponendo un simbolo sentinella — dunque che non sia già presente nell'alfabeto del nastro — nella prima cella.

Proposizione 3.2.2.1: Equivalenza delle TM multinastro

Sia M una macchina di Turing multinastro; allora esiste una TM M' ad essa equivalente.

Dimostrazione.

Prima implicazione. Sia M una macchina di Turing multinastro; si vuole dunque costruire una macchina di Turing M' tale da simulare M. Poiché M' è una TM, essa è costituita da un solo nastro, e qunque per simulare M è necessario suddividere il nastro di M'" in regioni, che saranno delimitate da un nuovo carattere #, e per simulare le testine di ogni nastro, verrà utilizzata una versione marcata dei caratteri in Γ, al fine di segnare "virtualmente" la posizione di ogni testina in ogni nastro. Dunque, dato un input $w = w_1 \cdots w_n$ sul nastro di M', si ha che:

- inizialmente M' trasforma il nastro nel formato descritto, e dunque da

  $$w_1 w_2 \cdots w_n \sqcup \ldots$$

  viene effetuato uno *shift* a destra del nastro (si veda l'Osservazione 3.2.2.1), anteposto il carattere # al suo inizio (si veda l'Osservazione 3.2.2.2), e successivamente inseriti i seguenti caratteri marcati per segnare le testine virtuali:

$$\# \ \dot{w}_1 \ w_2 \cdots w_n \ \# \ \dot{\sqcup} \ \# \ \dot{\sqcup} \ \# \ \ldots \ \#$$

- per simulare una singola mossa di M, la testina di M' scansiona tutto il suo nastro, aggiornando oppurtunamente i simboli marcati con $\bullet$ per simulare lo spostamento di ogni testina su ogni nastro; successivamente, viene effettuaa una seconda scansione, per simulare i rimpiazzi dei simboli sulle celle, descritti dalla δ;
- se una qualsiasi testina virtuale effettua uno spostamento a destra, e si trova su un # — dunque a cavallo tra 2 nastri simulati — allora M' deve simulare il fatto che, nel nastro reale, la reale testina in M starebbe leggendo una porzione di nastro vuota, non letta precedentemente; allora, viene effettuato uno *shift* verso destra di tutto il nastro di M', dalla cella corrente in poi, e sulla cella corrente viene posto il carattere $\sqcup$;

Di conseguenza, M' risulta essere in grado di simulare correttamente il comportamento di ogni nastro della macchina di Turing multinastro M, e dunque segue la tesi.

Seconda implicazione. Per $k = 1$, si ha che la funzione di transizione δ descritta nella Definizione 3.2.2.1 diventa

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

che rappresenta la funzione di transizione delle macchine di Turing con testina ferma (si veda la Definizione 3.2.1.1); allora, per la Proposizione 3.2.1.1, si ha che ogni macchina di Turing M', poiché è anche una macchina di Turing con testina ferma, è una macchina di Turing multinastro avente $k = 1$ nella funzione δ.

□

3.2.3 Macchine di Turing non deterministiche

Definizione 3.2.3.1: NTM

Una **NTM** (*Nondeterministic Turing Machine*) è una 7-tupla $(Q, \Sigma, \gamma, \delta, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$ dove ogni elemento della 7-tupla rimane invariato rispetto alla Definizione 3.1.1.1, a meno della funzione di transizione δ, definita come segue:

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

poiché una NTM computa non deterministicamente.

Proposizione 3.2.3.1: Equivalenza delle NTM

Sia N una NTM; allora esiste una TM M ad essa equivalente.

Dimostrazione.

Prima implicazione. TODO

Seconda implicazione. TODO

□

3.2.4 Enumeratori

Definizione 3.2.4.1: Enumeratore

Un **enumeratore** è una TM descritta da una 7-tupla $(Q, \Sigma, \gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ — dove ogni elemento della 7-tupla rimane invariato rispetto alla Definizione 3.1.1.1 — connessa ad una stampante (ad esempio un nastro secondario), la quale stampa le stringhe di TODO DA FINIRE

Proposizione 3.2.4.1: Equivalenza degli enumeratori

Sia E un enumeratore; allora esiste una TM M ad esso equivalente.

Dimostrazione.

Prima implicazione. TODO

Seconda implicazione. TODO

□

3.3 Linguaggi Turing-riconoscibili e Turing-decidibili

3.3.1 Tesi di Church-Turing

Definizione 3.3.1.1: Algoritmo

Un **algoritmo** è un insieme di istruzioni semplici per l'esecuzione di un certo compito.

Definizione 3.3.1.2: Tesi di Church-Turing

La **tesi di Church-Turing** è la seguente: "*la classe delle funzioni calcolabili coincide con la classe delle funzioni calcolabili da una macchina di Turing*".

Informalmente, la tesi di Church-Turing afferma che se un problema è umanamente calcolabile, allora esiste una macchina di Turing in grado di risolverlo, ovvero di calcolarlo.

La tesi di Church-Turing, sebbene ormai universalmente accettata, non può essere dimostrata.

Osservazione 3.3.1.1: Algoritmi e macchine di Turing

Poiché si assume la tesi di Church-Turing, si ha che dato un *algoritmo*, esiste una *macchina di Turing* che può eseguirlo. Dunque, all'interno di questi appunti verrà assunta l'intercambiabilità tra gli algoritmi (o metodi) descritti, e le macchine di Turing.

3.3.2 Linguaggi non Turing-riconoscibili

Osservazione 3.3.2.1: Codifiche

Sia O un oggetto; poichè una TM può prendere in input qualsiasi oggetto, ammesso che sia codificato attraverso una quaclhe codifica predeterminata, all'interno di questi appunti si utilizzerà il simbolismo $\langle O \rangle$ per sottointendere una qualche codifica del dato oggetto O, in modo che sia possibile rappresentare O sul nastro di una TM.

Esempio 3.3.2.1 (Codifiche di δ)**.** Sia δ la funzione di transizione di un DFA; allora, una sua possibile codifica potrebbe essere scrivere ogni riga della rappresentazione tabellare di δ, separata da un #.

Definizione 3.3.2.1: Numerabilità

Un insieme A è detto **numerabile** se e solo se esiste una biezione $f : \mathbb{N} \to A$.

Esempio 3.3.2.2 (Numeri pari)**.** L'insieme

$$P := \{2n \mid n \in \mathbb{N}\}$$

dei numeri pari è numerabile, poiché la funzione

$$f : \mathbb{N} \to P : n \mapsto 2n$$

è biettiva; infatti è iniettiva poiché

$$\forall x, y \in \mathbb{N} \quad f(x) = f(y) \iff 2x = 2y \implies x = y$$

ed è suriettiva poiché

$$\forall p \in P \quad \exists x \in \mathbb{N} \mid p = 2x$$

(altrimenti p non sarebbe pari).

Lemma 3.3.2.1: Numerabilità delle macchine di Turing

L'insieme delle macchine di Turing è numerabile; dunque, esiste una biezione $f : \mathbb{N} \to$ TM.

Dimostrazione. Sia Σ un alfabeto contenente dei caratteri che rendano possibile codificare ogni macchina di Turing in una codifica $\langle M \rangle$ per ogni $M \in$ TM; sia dunque $\prec$ una relazione d'ordine su Σ^* definita come segue:

$$\forall x, y \in \Sigma^* \quad x \prec y \iff \begin{cases} |x| < |y| & |x| \neq |y| \\ x \prec_l y & l := |x| = |y| \end{cases}$$

dove $\prec_l$ è la relazione d'ordine dell'ordinamento lessicografico. Si noti che, per sua stessa definizione, la relazione d'ordine $\prec$ risulta essere totale; di conseguenza, è possibile ordinare *totalmente* le stringhe presenti in Σ^*, ed è dunque possibile definire una biezione $f : \mathbb{N} \to \Sigma^*$ (è sufficiente associare $i \in \mathbb{N}$ con l'i-esima stringa ottenutta dall'ordinamento descritto). Questo dimostra che Σ^* è numerabile, per qualsiasi alfabeto Σ (una volta stabilito un ordinamento lessicografico per caratteri esterni alle lettere dell'alfabeto canonico, se presenti).

Dunque, poiché con l'alfabeto Σ scelto è possibile codificare ogni macchina di Turing TM, questo dimostra che è possibile numerare anche le macchine di Turing. Allora TM è numerabile. $\square$

Proposizione 3.3.2.1: Non-numerabilità di $\mathcal{B}$

Sia $\mathcal{B}$ l'insieme delle stringhe binarie di lunghezza infinita; in simboli

$$\mathcal{B} := \{w \in \{\mathsf{0}, \mathsf{1}\}^* \mid |w| = +\infty\}$$

Allora $\mathcal{B}$ non è numerabile.

Dimostrazione. La tecnica che verrà utilizzata è nota in letteratura come Argomento diagonale di Cantor, attraverso la quale Cantor ha dimostrato che $\mathbb{R}$ non è numerabile.

Per assurdo, sia $\mathcal{B}$ numerabile; allora, per definizione, esiste una biezione $f : \mathbb{N} \to \mathcal{B}$. Si assuma ad esempio che f sia la seguente mappa biunivoca:

$$\begin{array}{ccccccccc} f(0) & = & \mathsf{0} & \mathsf{1} & \mathsf{0} & \mathsf{0} & \mathsf{1} & \ldots \\ f(1) & = & \mathsf{1} & \mathsf{0} & \mathsf{0} & \mathsf{1} & \mathsf{0} & \ldots \\ f(2) & = & \mathsf{1} & \mathsf{0} & \mathsf{1} & \mathsf{0} & \mathsf{0} & \ldots \\ f(3) & = & \mathsf{0} & \mathsf{1} & \mathsf{0} & \mathsf{1} & \mathsf{1} & \ldots \\ f(4) & = & \mathsf{1} & \mathsf{0} & \mathsf{1} & \mathsf{0} & \mathsf{1} & \ldots \\ \vdots & & & & \vdots & & & \end{array}$$

e si consideri la stringa binaria descritta dall'opposto delle cifre sulla *diagonale* di tale matrice, ovvero

$$b := \mathsf{11000} \cdots$$

Essa dunque è definita come segue:

$$b \in \{\mathsf{0}, \mathsf{1}\}^* \mid b_i = \neg f(i)_i$$

estendendo la definizione di i-esimo carattere anche allo 0; allora per sua stessa definizione b non può essere presente nella mappa di f, poiché se esistesse un $k \in \mathbb{N}$ tale che $f(k) = b$, allora la k-esima cifra di b (ovvero b_k) dovrebbe essere il negato di $f(k)_k$ stesso, ma $f(k)_k = b_k$ poiché $f(k) = b$ $\lightning$. $\square$

Proposizione 3.3.2.2: Biezioni tra insiemi

Siano A e B due insiemi; allora esiste una biezione $f : A \to B$ se e solo se $|A| = |B|$.

Dimostrazione.

Prima implicazione. Siano A e B insiemi, sui quali è definita una biezione $f : A \to B$; allora:

- f è biettiva, ed in particolare è iniettiva, e dunque

$$\forall x_1, x_2 \in A \quad x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$$

  il che implica necessariamente che $|A| \subseteq |B|$;

- f è biettiva, ed in particolare è suriettiva, e dunque

$$\forall y \in B \quad \exists x \in A \mid f(x) = y$$

  il che implica necessariamente che $|B| \subseteq |A|$.

Allora $|A| = |B|$, come volevasi dimostrare.

Seconda implicazione. Se A e B sono due insiemi tali che $|A| = |B|$, allora è sicuramente possibile definire una mappa *uno-ad-uno* che associa gli elementi di A con gli elementi di B, poiché non possono essere presenti elementi in eccesso o in difetto in nessuno dei due insiemi in quanto hanno la stessa cardinalità; dunque, segue la tesi.

□

Lemma 3.3.2.2: Linguaggi non Turing-riconiscibili

Esistono linguaggi non Turing-riconiscibili; in simboli, dato un alfabeto Σ si ha che

$$\exists L \in \mathcal{P}(\Sigma^*) \mid L \notin \mathsf{REC}$$

Dimostrazione. Dato un linguaggio L definito su un qualche alfabeto Σ, ordinato secondo l'ordinamento $\prec$ definito all'interno della dimostrazione del Lemma 3.3.2.1, si definisce *sequenza caratteristica* di L una stringa binaria χ_L in cui l'i-esimo carattere di χ_L è $\mathtt{1}$ se e solo se l'i-esima stringa di L (rispetto a $\prec$) è presente in Σ^*. Ad esempio, dato

$$L := \{\mathtt{1}, \mathtt{01}, \mathtt{010}\}$$

definito sull'alfabeto binario (il quale è ordinabile su $\prec$), si ha che χ_L è ottenuta come segue:

$$\begin{array}{ccccccccccccccc}
\Sigma^* & = & \{ & \varepsilon, & \mathtt{0}, & \mathtt{1}, & \mathtt{00}, & \mathtt{01}, & \mathtt{10}, & \mathtt{11}, & \mathtt{000}, & \mathtt{001}, & \mathtt{010}, & \mathtt{011}, & \ldots \} \\
L & := & \{ & & & \mathtt{1}, & & \mathtt{01}, & & & & & \mathtt{010} & & \} \\
\chi_L & := & & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \cdots
\end{array}$$

Si consideri dunque la funzione

$$f : \mathcal{P}(\Sigma^*) \to \mathcal{B} : L \mapsto \chi_L$$

dove $\mathcal{B}$ è l'insieme delle stringhe binarie infinite; f dunque mappa ogni linguaggio nella sua stringa caratteristica. Si noti dunque che

$$\forall \chi_A, \chi_B \in \mathcal{B} \quad \chi_A = \chi_B \implies A = B$$

poiché le due stringhe descrivono lo stesso linguaggio; allora f è iniettiva. Inoltre, si verifica che

$$\forall \chi_L \in \mathcal{B} \quad \exists L \in \mathcal{P}(\Sigma^*) \mid f(L) = \chi_L$$

poiché ogni stringa binaria deve necessariamente descrivere un linguaggio, per definizione di χ_L; allora f è suriettiva. Di conseguenza, f è biettiva, e quindi per la Proposizione 3.3.2.2 si ha che

$$|\mathcal{P}(\Sigma^*)| = |\mathcal{B}|$$

Si noti però che, per la Proposizione 3.3.2.1, $\mathcal{B}$ non è numerabile, e di conseguenza neanche $\mathcal{P}(\Sigma^*)$ lo è. Questo prova che l'insieme dei linguaggi definiti su Σ non è numerabile.

Poiché l'insieme TM è numerabile per il Lemma 3.3.2.1, anche ogni sottoinsieme di TM deve necessariamente esserlo; allora, fissato un alfabeto Σ, si consideri l'insieme delle macchine di Turing definite su Σ; si noti che, per quanto dimostrato precedentemente, l'insieme dei linguaggi definiti su Σ non è numerabile. Allora, deve necessariamente esistere qualche linguaggio, definito su Σ, che non può essere riconosciuto da nessuna macchina di Turing definita su Σ stesso; di conseguenza, segue la tesi. □

3.3.3 Problema dell'accettazione

Definizione 3.3.3.1: Linguaggi di accettazione

Si definiscono **linguaggi di accettazione** i linguaggi definiti come segue

$$A_{\mathcal{C}} := \{\langle B, w\rangle \mid B \in \mathcal{C}, w \in L(B)\}$$

Osservazione 3.3.3.1: Decidibilità dei linguaggi $A_{\mathcal{C}}$

Si noti che, dimostrare che un linguaggio di accettazione è decidibile per una certa classe di oggetti $\mathcal{C}$, equivale a dimostrare che "stabilire se un certo oggetto in $\mathcal{C}$ accetta/descrive un certo input w" è un problema *computabile*, e dunque per l'Osservazione 3.3.1.1 di fatto esiste una macchina di Turing che può stabilirlo *senza andare in loop*; infatti, per definizione stessa di $A_{\mathcal{C}}$, controllare che $B \in \mathcal{C}$ accetti/descriva w equivale a controllare che $\langle B, w\rangle$ sia in $A_{\mathcal{C}}$.

Teorema 3.3.3.1: Decidibilità di A_{DFA}

A_{DFA} è decidibile; in simboli

$$A_{\mathsf{DFA}} := \{\langle B, w\rangle \mid B \in \mathsf{DFA}, w \in L(B)\} \in \mathsf{DEC}$$

Dimostrazione. Sia M una macchina di Turing; stabilita una qualche codifica per $\langle B, w\rangle$, per prima cosa, M controlla che sul nastro sia descritta una codifica valida per $\langle B, w\rangle$, ed in caso contrario rifiuta. Successivamente, M deve simulare la transizione di stati definita dalla δ di B, e può farlo servendosi delle infinite delle del suo nastro, aggiornando opportunamente lo stato corrente. Allora, M accetta se e solo se B avrebbe accettato, e rifiuta se e solo se B avrebbe rifiutato; allora M non va mai in loop, e dunque è un decisore. □

Teorema 3.3.3.2: Decidibilità di A_{NFA}

A_{NFA} è decidibile; in simboli

$$A_{\mathsf{NFA}} := \{\langle B, w\rangle \mid B \in \mathsf{NFA}, w \in L(B)\} \in \mathsf{DEC}$$

Dimostrazione I. Una volta controllata la validità della codifica in input, è possibile utilizzare una NTM M per simulare l'NFA posto sul suo nastro, e dunque M accetterebbe se e solo se l'NFA accetterebbe, e rifiuterebbe se e solo se l'NFA rifiuterebbe; allora, poiché M non può andare in loop, ed è dunque un decisore, per la Proposizione 3.2.3.1 segue la tesi. □

Dimostrazione II. Sia M una macchina di Turing; una volta controllata la validità della codifica in input, M può utilizzare l'algoritmo di conversione presentato all'interno della dimostrazione del Teorema 1.3.2.1 per convertire l'NFA fornito in input in un DFA (grazie all'Osservazione 3.3.1.1); allora, per il Teorema 3.3.3.1, segue la tesi. □

Teorema 3.3.3.3: Decidibilità di A_{REX}

A_{REX} è decidibile; in simboli

$$A_{\mathsf{REX}} := \{\langle R, w\rangle \mid R \in \mathsf{REX}, w \in L(R)\} \in \mathsf{DEC}$$

Dimostrazione. Sia M una macchina di Turing; una volta controllata la validità della codifica in input, M può utilizzare l'algoritmo di conversione presentato all'interno della dimostrazione del Teorema 1.7.2.1 per convertire l'espressione regolare fornita in input in un NFA (grazie all'Osservazione 3.3.1.1); allora, per il Teorema 3.3.3.2, segue la tesi. □

Teorema 3.3.3.4: Decidibilità di A_{CFG}

A_{CFG} è decidibile; in simboli

$$A_{\mathsf{CFG}} := \{\langle G, w\rangle \mid G \in \mathsf{CFG}, w \in L(G)\} \in \mathsf{DEC}$$

Dimostrazione. Se una grammatica G è in CNF, allora ogni stringa w, avente lunghezza $n := |w| \geq 1$, derivabile attraverso le regole di G, richiede esattamente $2n - 1$ passaggi. La dimostrazione procede per induzione sulla lunghezza della stirnga w.

Caso base. Per $n = 1$, poiché G è in CNF, si ha che l'unica regola che può aver prodotto la stringa w è della forma $S \to a$ dove $a \in \Sigma$, ed infatti il numero di passaggi è pari a

$$2n - 1 = 2 \cdot 1 - 1 = 2 - 1 = 1$$

Ipotesi induttiva forte. Data una grammatica G in CNF, ogni stringa non vuota w lunga al più n derivabile attraverso le sue regole, richiede esattamente $2n - 1$ passaggi.

Passo induttivo. È necessario dimostrare la tesi per una stringa w tale che $|w| = n+1$. Poiché G è in CNF, le sue regole possono essere della forma $A \to BC$, oppure $D \to a$ con $a \in \Sigma$; allora, assumendo che w abbia lunghezza $n + 1$ con $n > 1$, il numero di passaggi per derivare w deve essere maggiore 1 (come dimostrato nel caso base), e dunque deve esistere una regola della forma $A \to BC$ che possa aver prodotto w; dunque, esisteranno due sottostringhe y e z di w, tali che $B \overset{*}{\Rightarrow} y$ e $C \overset{*}{\Rightarrow} z$, e sicuramente non vuote, poiché sono state generate da una regola della forma $A \to BC$. Dunque, poiché non sono vuote, e sono sottostringhe di w, hanno lunghezza al più n, e di conseguenza su di esse è possibile applicare l'ipotesi induttiva forte. Allora, ponendo $k := |y|$, e di conseguenza $|z| = |w| - |y| = n + 1 - k$, si ha che y è stata derivata attraverso

$$2k - 1$$

passaggi, mentre w attraverso

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

passaggi. Allora, w deve essere stata derivata attraverso i passaggi per derivare y, i passaggi per derivare z, ed il passaggio $A \to BC$ stesso, e dunque sono

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

passaggi. Allora segue la tesi, poiché

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

Sia dunque M una macchina di Turing; una volta controllata la validità della codifica in input, M può convertire la CFG— fornita in input — in CNF, attraverso il Metodo 2.1.3.1 (grazie all'Osservazione 3.3.1.1), e successivamente

- se la stringa w ha lunghezza 0, M controlla che la regola $S \to \varepsilon$ sia presente in G, altrimenti rifiuta;
- se la stringa w ha lunghezza $n \geq 1$, M lista tutte le derivazioni di G, resa in CNF, aventi lunghezza $2n-1$ (sufficiente per quanto dimostrato precedentemente), le quali sono in numero *finito* — dunque M non va in loop — e se w è presente in una di queste, M accetta, altrimenti rifiuta.

Allora M non può andare in loop, quindi è un decisore, e dunque segue la tesi. □

Corollario 3.3.3.1: Decidibilità dei CFL

Ogni CFL è decidibile; in simboli

$$\mathsf{CFL} \subseteq \mathsf{DEC}$$

Dimostrazione. Sia $L \in \mathsf{CFL}$, e dunque per definizione esiste una $G \in \mathsf{CFG}$ tale che $L(G) = L$; affinché L sia decidibile, deve esistere una TM in grado di deciderlo, e dunque di riconoscere ogni sua stringa $w \in L$, senza andare mai in loop. Si consideri ora la TM costruita all'interno della dimostrazione del Teorema 3.3.3.4, la quale decide se una data stringa appartiene al linguaggio di una data CFG, e sia questa M_G. Allora, per decidere L è sufficiente costruire una TM M — avente M_G come sua sottoprocedura — tale che, data una stringa $w \in L$ in input, esegua M_G su $\langle G, w\rangle$ per stabilire se $w \in L(G)$. Allora M è un decisore poiché M_G è un decisore, e dunque segue la tesi. □

Definizione 3.3.3.2: Macchina di Turing universale

Si definisce **macchina di Turing universale** una macchina di Turing in grado di simulare qualsiasi altra macchina di Turing.

Teorema 3.3.3.5: Riconoscibilità di A_{TM}

A_{TM} è Turing-riconoscibile; in simboli

$$A_{\mathsf{TM}} := \{\langle M, w\rangle \mid M \in \mathsf{TM}, w \in L(M)\} \in \mathsf{REC}$$

Dimostrazione. Sia M una macchina di Turing multinastro; una volta controllata la validità della codifica in input (presente su uno dei due nastri), sia M tale da computare come segue:

- uno dei due nastri viene utilizzato da M per mantenere le codifiche della macchina di Turing in input, che comprendono:
  - il numero di stati $\langle |Q| \rangle$
  - il numero di simboli dell'alfabeto dell'automa $\langle |\Sigma| \rangle$
  - il numero di simboli dell'alfabeto del nastro $\langle |\Gamma| \rangle$
  - la funzione $\langle \delta \rangle$, descritta attraverso codifica di tuple $\langle \langle q, x\rangle, \langle r, y, z\rangle\rangle$ che costituiscono coppie input-output presenti nella rappresentazione insiemistica della funzione di transizione δ (dunque, si ha che $q, r \in Q, x, y \in \Gamma, z \in \{\mathrm{L}, \mathrm{R}\}$)
- l'altro nastro viene utilizzato da M per mantenere lo stato della configurazione corrente della macchina di Turing che M deve simulare, indicata attraverso la codifica $\langle a, q, b\rangle$ (dove $q \in Q, a, b \in \Gamma^*$)
- allora, M inizialmente scansiona il primo nastro, cercando $\langle \delta \rangle$;

- successivamente, per ogni codifica di regole $\langle\langle q, x\rangle, \langle r, y, z\rangle\rangle$, se $x = b_1$, la codifica della configurazione corrente viene aggiornata in base alla transizione considerata, altrimenti M passa alla regola successiva;
- se la condifica della configurazione corrente, presente sul secondo nastro, contiene q_{accept} o q_{reject}, M accetta o rifiuta rispettivamente.

Allora M è in grado di simulare la macchina di Turing che gli è stata fornita in input; si noti però che se la macchina che M sta simulando va in loop, anche M va in loop necessariamente. Allora M riconosce A_{TM}, ma non lo decide. Inoltre, per la Proposizione 3.2.2.1 si ha che esiste una TM U equivalente alla M descritta, che risulta dunque essere una macchina di Turing universale. Allora, poiché U è equivalente ad M, U riconosce (ma non decide) A_{TM}, e dunque segue la tesi. □

Teorema 3.3.3.6: Indecidibilità di A_{TM}

A_{TM} è indecidibile; in simboli

$$A_{\mathsf{TM}} := \{\langle M, w\rangle \mid M \in \mathsf{TM}, w \in L(M)\} \in \mathsf{REC} - \mathsf{DEC}$$

Dimostrazione. Per assurdo, sia $A_{\mathsf{TM}} \in \mathsf{DEC}$, e dunque per definizione esiste un decisore H tale che $L(H) = A_{\mathsf{TM}}$. Allora, poiché è un decisore, H computa come segue:

$$\forall M \in \mathsf{TM}, w \in L(M) \quad H(\langle M, w\rangle) = \begin{cases} \text{accetta} & M \text{ accetta } w \\ \text{rifiuta} & M \text{ rifiuta } w \end{cases}$$

dove la sintassi $H(\langle M, w\rangle)$ indica che H computa avendo la codifica $\langle M, w\rangle$ sul nastro. Sia ora D una macchina di Turing avente H come sua sottoprocedura, la quale viene chiamata quando l'input di M è la descrizione di M stessa (ovvero $\langle M\rangle$). Inoltre, D computa come segue:

$$\forall M \in \mathsf{TM} \quad D(\langle M\rangle) = \begin{cases} \text{accetta} & H(\langle M, \langle M\rangle\rangle) \text{ rifiuta} \iff M \text{ rifiuta } \langle M\rangle \\ \text{rifiuta} & H(\langle M, \langle M\rangle\rangle) \text{ accetta} \iff M \text{ accetta } \langle M\rangle \end{cases}$$

dunque D inverte il risultato ottenuto computando $H(\langle M, \langle M\rangle\rangle)$. Allora, si ha che se M è pari a D stesso, si ottiene che

$$D(\langle D\rangle) = \begin{cases} \text{accetta} & H(\langle D, \langle D\rangle\rangle) \text{ rifiuta} \iff D \text{ rifiuta } \langle D\rangle \\ \text{rifiuta} & H(\langle D, \langle D\rangle\rangle) \text{ accetta} \iff D \text{ accetta } \langle D\rangle \end{cases}$$

che rappresenta chiaramente un assurdo $\lightning$. □

Definizione 3.3.3.3: CoTuring-riconoscibilità

Un linguaggio è detto **coTuring-riconoscibile** se questo è il complemento di un linguaggio Turing-riconoscibile. La classe dei lingauggi coTuring-riconoscibili è denotata con COREC. Allora, in simboli, dato un linguaggio L, si ha che

$$L \in \mathsf{COREC} \iff \neg L \in \mathsf{REC}$$

Teorema 3.3.3.7: CoTuring-riconoscibilità e decidibilità

Un linguaggio è decidibile se e solo se è Turing-riconoscibile e coTuring-riconoscibile; in simboli

$$\mathsf{DEC} = \mathsf{REC} \cap \mathsf{COREC}$$

Dimostrazione.

Prima implicazione. Sia $L \in \mathsf{DEC}$; per l'Osservazione 3.1.4.2 si ha che $\mathsf{DEC} \subseteq \mathsf{REC}$, dunque sicuramente $L \in \mathsf{REC}$. Inoltre, poiché $L \in \mathsf{DEC}$, per definizione esiste un decisore $M \in \mathsf{TM}$ tale che $L(M) = L$; allora, se si invertono gli stati di accettazione e di rifiuto di M, si ottiene una TM M' che riconosce $\neg L$, poiché accetta se e solo se M rifiuta, e rifiuta se e solo se M accetta; di conseguenza, M' non può andare in loop, e dunque è un decisore. Allora, poiché esiste un decisore $M' \in \mathsf{TM}$ tale che $L(M') = \neg L$, si ha che $\neg L \in \mathsf{DEC}$ per definizione stessa, ed in particolare

$$\neg L \in \mathsf{DEC} \implies \neg L \in \mathsf{REC} \iff L \in \mathsf{COREC}$$

Seconda implicazione. Sia L tale che $L \in \mathsf{REC}$ e $L \in \mathsf{COREC}$, e dunque siano M_1 ed M_2 rispettivamente le TM tali da riconoscere L e $\neg L$; inoltre, sia Σ l'alfabeto su cui sono definiti L ed $\neg L$, e si consideri una stringa $w \in \Sigma$. Si noti che, per definizione stessa di complemento, si ha che

$$L \cup \neg L = \Sigma^*$$
$$L \cap \neg L = \varnothing$$

e dunque $w \in L \iff w \notin \neg L$. Allora, fornendo in input w alle TM M_1 ed M_2, si verifica che una delle due macchine deve necessariamente accettare w, indipendentemente se l'altra rifiuta o va in loop.

Sia allora M una NTM costituita da 2 nastri, tale da simulare *concorrentemente* — dunque un passo alla volta — M_1 ed M_2 sui suoi 2 nastri, e tale che

$$M(w) = \begin{cases} \text{accetta} & M_1 \text{ accetta } w \\ \text{rifiuta} & M_2 \text{ accetta } w \end{cases}$$

Per quanto detto in precedenza, fornendo in input una qualsiasi stringa $w \in \Sigma^*$ ad M, si verifica che una delle due TM simulate accetta w necessariamente. Allora, segue che M termina sempre, e dunque è un decisore; inoltre, poiché accetta se e solo se M_1 accetta — la quale riconosce L — e rifiuta se e solo se M_2 accetta — la quale riconosce $\neg L$ — di fatto M decide L, e dunque $L \in \mathsf{DEC}$.

□

Corollario 3.3.3.2: Non-riconoscibilità di $\neg A_{\mathsf{TM}}$

A_{TM} non è coTuring-riconoscibile; in simboli

$$A_{\mathsf{TM}} \notin \mathsf{COREC}$$

Dimostrazione. Per il Teorema 3.3.3.5 ed il Teorema 3.3.3.6, si ha che

$$A_{\mathsf{TM}} \in \mathsf{REC} - \mathsf{DEC}$$

allora per il Teorema 3.3.3.7, segue che

$$A_{\mathsf{TM}} \in \mathsf{REC} - \mathsf{DEC} \iff A_{\mathsf{TM}} \notin \mathsf{COREC}$$

□

3.3.4 Problema del vuoto

Definizione 3.3.4.1: Linguaggi del vuoto

Si definiscono **linguaggi del vuoto** i linguaggi definiti come segue

$$E_{\mathcal{C}} := \{\langle A \rangle \mid A \in \mathcal{C} : L(A) = \varnothing\}$$

Osservazione 3.3.4.1: Decidibilità dei linguaggi $E_{\mathcal{C}}$

Si noti che, dimostrare che un linguaggio del vuoto è decidibile per una certa classe di oggetti $\mathcal{C}$, equivale a dimostrare che "stabilire se il linguaggio di un certo oggetto in $\mathcal{C}$ è vuoto" è un problema *computabile*, e dunque per l'Osservazione 3.3.1.1 di fatto esiste una macchina di Turing che può stabilirlo *senza andare in loop*; infatti, per definizione stessa di $E_{\mathcal{C}}$, controllare che il linguaggio di $A \in \mathcal{C}$ sia vuoto equivale a controllare che $\langle A \rangle$ sia in $E_{\mathcal{C}}$.

Teorema 3.3.4.1: Decidibilità di E_{DFA}

E_{DFA} è decidibile; in simboli

$$E_{\mathsf{DFA}} := \{\langle A \rangle \mid A \in \mathsf{DFA} : L(A) = \varnothing\} \in \mathsf{DEC}$$

Dimostrazione. Sia M una macchina di Turing; una volta controllata la validità della codifica in input, M può interpretare il DFA fornito in input come un grafo, e dunque controllare che tale DFA non accetti alcuna stringa equivale a controllare che non esista alcun cammino dallo stato iniziale del DFA ad un qualsiasi suo stato accettante; allora, usando ad esempio una visita dei nodi in DFS (*Depth-First Search*) — realizzabile grazie all'Osservazione 3.3.1.1 — M accetta se e solo se non è presente alcun cammino come descritto, e dunque non può andare in loop; allora, segue la tesi. □

Teorema 3.3.4.2: Decidibilità di E_{CFG}

E_{CFG} è decidibile; in simboli

$$E_{\mathsf{CFG}} := \{\langle G \rangle \mid G \in \mathsf{CFG} : L(G) = \varnothing\} \in \mathsf{DEC}$$

Dimostrazione. Sia M una macchina di Turing; una volta controllata la validità della codifica in input, M può utilizzare il seguente algoritmo — grazie all'Osservazione 3.3.1.1 — per stabilire se il linguaggio della grammatica $G = (V, \Sigma, R, S)$ presente sul suo nasstro sia vuoto:

- M marca inizialmente tutti i simboli terminali di G;
- fintanto che sono presenti variabili non marcate, M marca ripetutamente ogni variabile $A \in V$ tale che $A \to U_1 \cdots U_k \in R$, dove $U_1, \ldots, U_k \in (V \cup \Sigma)^*$ sono già stati tutti marcati;
- accetta se e solo se S non è marcata, altrimenti rifiuta.

Di conseguenza, l'algoritmo che sta eseguendo M su G, partendo dai terminali di quest'ultima, controlla che sia possibile ripercorrere gli alberi di derivazione dai terminali fino ad S, attraverso le regole in R, sfruttate "al contrario". Allora, se al termine dell'algoritmo S è marcata, esiste una sequenza di terminali derivabile da S, e dunque $L(G) \neq \varnothing$. □

3.3.5 Problema dell'uguaglianza

Definizione 3.3.5.1: Linguaggi dell'uguaglianza

Si definiscono **linguaggi dell'uguaglianza** i linguaggi definiti come segue:

$$EQ_{\mathcal{C}} := \{\langle A, B\rangle \mid A, B \in \mathcal{C} : L(A) = L(B)\}$$

Osservazione 3.3.5.1: Decidibilità dei linguaggi $EQ_{\mathcal{C}}$

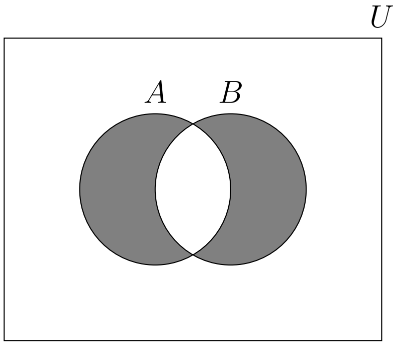
Si noti che, dimostrare che un linguaggio dell'uguaglianza è decidibile per una certa classe di oggetti $\mathcal{C}$, equivale a dimostrare che "stabilire se due oggetti in $\mathcal{C}$ siano equivalenti" è un problema *computabile*, e dunque per l'Osservazione 3.3.1.1 di fatto esiste una macchina di Turing che può stabilirlo *senza andare in loop*; infatti, per definizione stessa di $EQ_{\mathcal{C}}$, controllare che $A \in \mathcal{C}$ sia equivalente a $B \in \mathcal{C}$ corrisponde a controllare che $\langle A, B\rangle$ sia in $EQ_{\mathcal{C}}$.

Definizione 3.3.5.2: Differenza simmetrica

Siano A e B due insiemi; allora, si definisce **differenza simmetrica** tra A e B la seguente:

$$A \mathbin{\Delta} B := (A \cap \neg B) \cup (\neg A \cap B)$$

Graficamente, dati due insiemi A e B all'interno di un insieme universo U, la differenza simmetrica $A \mathbin{\Delta} B$ è colorata in grigio nel seguente diagramma:



Proposizione 3.3.5.1: Differenza simmetrica vuota

Dati due insiemi A e B, si ha che

$$A \mathbin{\Delta} B = \varnothing \iff A = B$$

Dimostrazione.

Prima implicazione. Per assurdo, sia $A \mathbin{\Delta} B = \varnothing$ e $A \neq B$; si noti che $A \neq B$ se e solo se esiste $x \in A - B$, oppure $x \in B - A$, e dunque

- $x \in A - B \implies x \in A \land x \notin B \implies x \in A \cap \neg B \implies A \mathbin{\Delta} B \neq \varnothing$ ↯
- $x \in B - A \implies x \in B \land x \notin A \implies x \in \neg A \cap B \implies A \mathbin{\Delta} B \neq \varnothing$ ↯

Seconda implicazione. Per dimostrare la tesi, è sufficiente considerare che

$$A = B \implies A \cap \neg B = \neg A \cap B = \varnothing \implies A \mathbin{\Delta} B = \varnothing$$

□

Teorema 3.3.5.1: Decidibilità di EQ_{DFA}

EQ_{DFA} è decidibile; in simboli

$$EQ_{\mathsf{DFA}} := \{\langle A, B\rangle \mid A, B \in \mathsf{DFA} : L(A) = L(B)\} \in \mathsf{DEC}$$

Dimostrazione. Sia M una macchina di Turing; una volta controllata la validità della codifica in input, M può convertire i due DFA A e B forniti in input in un terzo

DFA C — utilizzando gli algoritmi presentati all'interno della Proposizione 1.4.1.1, della Proposizione 1.4.2.1 e della Proposizione 1.4.6.1 — tale che

$$L(C) = L(A) \,\Delta\, L(B)$$

e dunque per la Proposizione 3.3.5.1, si ha che

$$L(A) = L(B) \iff L(C) = \varnothing$$

Sia M' la TM costruita all'interno del Teorema 3.3.4.1; allora, M può utilizzare M' sull'input $\langle C \rangle$, poiché se M' accetta, allora $L(C) = \varnothing$, e dunque A è equivalente a B; allora M accetta se e solo se M' accetta, e poiché M' è decisore per la dimostrazione del Teorema 3.3.4.1 stessa, segue la tesi. □

Teorema 3.3.5.2: Indecidibilità di EQ_{CFG}

EQ_{CFG} è indecidibile; in simboli

$$EQ_{\mathsf{CFG}} := \{\langle G, H \rangle \mid G, H \in \mathsf{CFG} : L(G) = L(H)\} \notin \mathsf{DEC}$$

Dimostrazione. Omessa. □

4

Riducibilità

4.1 Riduzione

4.1.1 Definizioni

Definizione 4.1.1.1: Riduzione

Una **riduzione** è un modo di convertire un problema in un altro, in modo tale che una soluzione al secondo possa essere utilizzata per risolvere il primo.

Definizione 4.1.1.2: Funzione calcolabile

Una funzione $f : \Sigma^* \to \Sigma^*$ è detta **funzione calcolabile** se esiste una macchina di Turing che, su qualsiasi input $w \in \Sigma^*$, si ferma avendo solamente $f(w)$ sul nastro.

Definizione 4.1.1.3: Riducibilità mediante funzione

Siano A e B definiti su un alfabeto Σ; A si dice **riducibile mediante funzione** a B, se esiste una funzione calcolabile $f : \Sigma^* \to \Sigma^*$ tale che

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

Tale relazione è indicata attraverso il simbolismo

$$A \leq_{\mathrm{m}} B$$

La funzione f prende il nome di **riduzione** da A a B.

Teorema 4.1.1.1: Decidibilità mediante riduzione

Siano A e B due linguaggi definiti su un alfabeto Σ; se $A \leq_{\mathrm{m}} B$, e B è decidibile, allora A è decidibile. In simboli

$$\begin{cases} A \leq_{\mathrm{m}} B \\ B \in \mathsf{DEC} \end{cases} \implies A \in \mathsf{DEC}$$

Dimostrazione. Poiché $B \in \mathsf{DEC}$ in ipotesi, esiste un decisore M tale che M decide B; inoltre, poichè $A \leq_{\mathrm{m}} B$, esiste una funzione calcolabile $f : \Sigma^* \to \Sigma^*$ tale per cui A sia riducibile a B. Sia dunque N una macchina di Turing che, data in input una stringa $w \in \Sigma^*$, computa come segue:

- computa $f(w)$;
- simula M avente $f(w)$ come input;
- N accetta se e solo se M accetta.

Allora, per costruzione di N, e per riducibilità di A a B, si ha che

$$w \in L(N) \iff f(w) \in L(M) = B \iff w \in A$$

e dunque N decide A. □

Corollario 4.1.1.1: Indecidibilità mediante riduzione

Siano A e B due linguaggi definiti su un alfabeto Σ; se $A \leq_{\mathrm{m}} B$, ed A è indecidibile, allora B è indecidibile. In simboli

$$\begin{cases} A \leq_{\mathrm{m}} B \\ A \notin \mathsf{DEC} \end{cases} \implies B \notin \mathsf{DEC}$$

Dimostrazione. Per assurdo, sia $B \in \mathsf{DEC}$; allora, poiché $A \leq_{\mathrm{m}} B$, per il Teorema 4.1.1.1, si ha che $A \in \mathsf{DEC}$ ↯. □

4.2 Linguaggi non Turing-decidibili

4.2.1 Problema della terminazione

Definizione 4.2.1.1: Linguaggi di terminazione

Si definiscono **linguaggi di terminazione** i linguaggi definiti come segue

$$HALT_{\mathcal{C}} := \{\langle A, w\rangle \mid A \in \mathcal{C}, A \text{ si ferma su input } w\}$$

Dimostrazione. TODO □

Osservazione 4.2.1.1: Decidibilità dei linguaggi $HALT_{\mathcal{C}}$

Si noti che, dimostrare che un linguaggio di terminazione è decidibile per una certa classe di oggetti $\mathcal{C}$, equivale a dimostrare che "stabilire se un certo oggetto in $\mathcal{C}$ si ferma su input w" è un problema *computabile*, e dunque per l'Osservazione 3.3.1.1 di fatto esiste una macchina di Turing che può stabilirlo *senza andare in loop*; infatti, per definizione stessa di $HALT_{\mathcal{C}}$, controllare che $A \in \mathcal{C}$ si fermi su input w equivale a controllare che $\langle A, w \rangle$ sia in $HALT_{\mathcal{C}}$.

Teorema 4.2.1.1: Indecidibilità di $HALT_{\mathsf{TM}}$

$HALT_{\mathsf{TM}}$ è indecidibile; in simboli

$$HALT_{\mathsf{TM}} := \{\langle M, w \rangle \mid M \in \mathsf{TM}, M \text{ si ferma su input } w\} \notin \mathsf{DEC}$$

Dimostrazione. Per assurdo, sia $HALT_{\mathsf{TM}}$ decidibile, e dunque per esso esiste un decisore H; di conseguenza $H(\langle M, w \rangle)$ accetta se e solo se M si ferma avendo w come input. Si consideri ora una macchina di Turing D — avente H come sua sottoprocedura — che, data in input una codifica $\langle M, w \rangle$, computa come segue:

- esegue $H(\langle M, w \rangle)$;
- se H rifiuta, allora M va in loop avendo w come input, e dunque rifiuta;
- se H accetta, allora M si ferma avendo w come input, e dunque procede a simulare M stessa;
  - se M accetta, D accetta
  - se M rifiuta, D rifiuta.

Di conseguenza, D è in grado di stabilire, data una macchina di Turing M ed un input w, se $w \in L(M)$; inoltre, si noti che D non può andare in loop, e dunque D è un decisore. L'esistenza di D dimostrerebbe dunque che $A_{\mathsf{TM}} \in \mathsf{DEC}$, ma questo è assurdo per il Teorema 3.3.3.6 $\lightning$. □

4.2.2 Problema del vuoto

Teorema 4.2.2.1: Indecidibilità di E_{TM}

E_{TM} è indecidibile; in simboli

$$E_{\mathsf{TM}} := \{\langle M \rangle \mid M \in \mathsf{TM} : L(M) = \varnothing\} \notin \mathsf{DEC}$$

Dimostrazione. Per assurdo, sia E_{TM} decidibile, e dunque per esso esiste un decisore H; di conseguenza $H(\langle M \rangle)$ accetta se e solo se $L(M) = \varnothing$. Si consideri ora una macchina di Turing D — avente H come sua sottoprocedura — che, data in input una codifica $\langle M, w \rangle$, computa come segue:

- modifica la macchina M input in modo che, data in input una stringa x, computi come segue:
  - se l'input x di M è proprio w, computa normalmente;
  - se l'input x di M non è w, rifiuta;

  e sia M' la versione modificata di M (si noti che questa modifica è realizzabile da parte di D, poiché è sufficiente aggiungere degli stati ad M che controllino l'input fornito, e computare come descritto);
- esegue $H(\langle M' \rangle)$;
- se H accetta, allora $L(M') = \varnothing$, e dunque rifiuta;
- se H rifiuta, allora $L(M') \neq \varnothing$, e dunque accetta.

Di conseguenza, D è in grado di stabilire, data una macchina di Turing M ed un input w, se $w \in L(M)$, poiché accetta se e solo se H rifiuta, ovvero se e solo se $L(M') \neq \varnothing$, e per costruzione di M' si ha che $L(M') \neq \varnothing \iff L(M') = \{w\}$; inoltre, si noti che D non può andare in loop, e dunque D è un decisore. L'esistenza di D dimostrerebbe dunque che $A_{\mathsf{TM}} \in \mathsf{DEC}$, ma questo non è possibile per il Teorema 3.3.3.6 $\lightning$. $\square$

4.2.3 Problema della regolarità

Definizione 4.2.3.1: Linguaggi di regolarità

Si definiscono **linguaggi di regolarità** i linguaggi definiti come segue

$$REG_{\mathcal{C}} := \{\langle A \rangle \mid A \in \mathcal{C}, L(A) \in \mathsf{REG}\}$$

Osservazione 4.2.3.1: Decidibilità dei linguaggi $REG_{\mathcal{C}}$

Si noti che, dimostrare che un linguaggio di regolarità è decidibile per una certa classe di oggetti $\mathcal{C}$, equivale a dimostrare che "stabilire se il linguaggio di un certo oggetto in $\mathcal{C}$ sia regolare" è un problema *computabile*, e dunque per l'Osservazione 3.3.1.1 di fatto esiste una macchina di Turing che può stabilirlo *senza andare in loop*; infatti, per definizione stessa di $REG_{\mathcal{C}}$, controllare che il linguaggio di $A \in \mathcal{C}$ sia regolare equivale a controllare che $\langle A \rangle$ sia in $REG_{\mathcal{C}}$.

Teorema 4.2.3.1: Indecidibilità di REG_{TM}

REG_{TM} è indecidibile; in simboli

$$REG_{\mathsf{TM}} := \{\langle M \rangle \mid M \in \mathsf{TM}, L(M) \in \mathsf{REG}\} \notin \mathsf{DEC}$$

Dimostrazione. Per assurdo, sia REG_{TM} decidibile, e dunque per esso esiste un decisore H; di conseguenza $H(\langle M \rangle)$ accetta se e solo se $L(M) \in \mathsf{REG}$. Si consideri ora una macchina di Turing D — avente H come sua sottoprocedura — che, data in input una codifica $\langle M, w\rangle$, computa come segue:

- costruisce una macchina di Turing M' che, data in input una stringa x, computa come segue:
  - se l'input x di M' ha la forma $\mathtt{0}^n\mathtt{1}^n$ (per qualche $n \in \mathbb{N}$), accetta;
  - se l'input x di M' non ha tale forma, accetta se $M(w)$ accetta (dunque ignorando x);
- esegue $H(\langle M'\rangle)$;
- se H accetta, allora $L(M') \in \mathsf{REG}$, e dunque accetta;
- se H rifiuta, allora $L(M') \in \mathsf{REG}$, e dunque rifiuta.

Di conseguenza, D è in grado di stabilire, data una macchina di Turing M ed un input w, se $w \in L(M)$, poiché accetta se e solo se H accetta, ovvero se $L(M') \in \mathsf{REG}$, e per costruzione di M' si ha che $L(M') \in \mathsf{REG} \iff \begin{cases} x \notin \{\mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N}\} \\ M(w) \text{ accetta} \end{cases}$ TODO DA FINIRE NON HO CAPITO □

4.2.4 Problema dell'uguaglianza

Teorema 4.2.4.1: Indecidibilità di EQ_{TM}

EQ_{TM} è indecidibile; in simboli

$$EQ_{\mathsf{TM}} := \{\langle M_1, M_2\rangle \mid M_1, M_2 \in \mathsf{TM}, L(M_1) = L(M_2)\} \notin \mathsf{DEC}$$

Dimostrazione. Per assurdo, sia EQ_{TM} decidibile, e dunque per esso esiste un decisore H; di conseguenza $H(\langle M_1\rangle, M_2)$ accetta se e solo se $L(M_1) = L(M_2)$. Si consideri ora una macchina di Turing D — avente H come sua sottoprocedura — che, data in input una codifica $\langle M\rangle$, computa come segue:

- esegue $H(\langle M, M_{\text{reject}}\rangle)$, dove M_{reject} è una macchina di Turing che rifiuta ogni input, e dunque $L(M_{\text{reject}}) = \varnothing$;
- se H accetta, allora $L(M) = L(M_{\text{reject}}) = \varnothing$, e dunque accetta;
- se H rifiuta, allora $L(M) \neq L(M_{\text{reject}}) = \varnothing$, e dunque rifiuta.

Di conseguenza, D è in grado di stabilire, data una macchina di Turing M, se $L(M) = \varnothing$, poiché attraverso H controlla che il linguaggio di M coincida con $L(M_{\text{reject}}) = \varnothing$; inoltre, si noti che D non può andare in loop, e dunque D è un decisore. L'esistenza stessa di D dimostrerebbe dunque che $E_{\mathsf{TM}} \in \mathsf{DEC}$, ma questo non è possibile per il Teorema 4.2.2.1 $\lightning$. □